ISBN 978-1-939133-32-8

usenix ASSOCIATION

Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST '23) *Santa Clara, CA, USA February 21–23, 2023*

conference

proceedings

21st USENIX Conference on File and Storage Technologies (FAST '23)

Santa Clara, CA, USA

February 21–23, 2023

Sponsored by

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In cooperation with ACM SIGOPS

FAST '23 Sponsors

Gold Sponsors

amazon

FUTUREWEI Technologies

NSF

Silver Sponsor

Google

Bronze Sponsors

IBM

Meta

SAMSUNG

SNIA

Open Access Sponsor

NetApp

USENIX Supporters

USENIX Patrons

Amazon • Futurewei
Google • Meta • NetApp

USENIX Benefactors

Bloomberg • Goldman Sachs

USENIX Partner

Thinkst Canary

Open Access Supporter

Google

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the 21st USENIX Conference on File and Storage Technologies

February 21–23, 2023
Santa Clara, CA, USA

ISBN 978-1-939133-32-8

Conference Organizers

Program Co-Chairs

Ashvin Goel, *University of Toronto*

Dalit Naor, *The Academic College of Tel Aviv–Yaffo*

Program Committee

Nitin Agrawal, *Google*

Deniz Altinbüken, *Google Research*

Lakshmi N. Bairavasundaram, *VMware, Inc.*

Randal Burns, *Johns Hopkins University*

Ali R. Butt, *Virginia Tech*

Rong Chen, *Shanghai Jiao Tong University*

Sangyeun Cho, *Samsung Electronics Co.*

Young-ri Choi, *UNIST (Ulsan National Institute of Science and Technology)*

Alex Conway, *VMware Research*

Peter Desnoyers, *Northeastern University and Red Hat*

Yu Hua, *Huazhong University of Science and Technology*

Sudarsun Kannan, *Rutgers University*

Sanidhya Kashyap, *EPFL*

Kimberly Keeton, *Google*

Geoff Kuenning, *Harvey Mudd College*

Patrick P. C. Lee, *The Chinese University of Hong Kong*

Youyou Lu, *Tsinghua University*

Xiaosong Ma, *Qatar Computer Research Institute, HBKU*

Peter Macko, *MongoDB*

Ethan Miller, *University of California, Santa Cruz, and Pure Storage*

Hyungon Moon, *UNIST (Ulsan National Institute of Science and Technology)*

Adam Morrison, *Tel Aviv University*

Beomseok Nam, *Sungkyunkwan University (SKKU)*

Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology) and Virginia Tech*

Raju Rangaswami, *Florida International University*

Rob Ross, *Argonne National Laboratory*

Jiri Schindler, *Tranquil Data*

Philip Shilane, *Dell Technologies*

Liuba Shrira, *Brandeis University*

Keith A. Smith, *MongoDB*

Haris Volos, *University of Cyprus*

Carl Waldspurger, *Carl Waldspurger Consulting*

Avani Wildani, *Emory University*

Youjip Won, *Korea Advanced Institute of Science and Technology (KAIST)*

Gala Yadgar, *Technion—Israel Institute of Technology*

Work-in-Progress/Posters Co-Chairs

Ram Alagappan, *University of Illinois at Urbana–Champaign*

Aishwarya Ganesan, *University of Illinois at Urbana–Champaign*

Test of Time Awards Committee

Nitin Agrawal, *Google*

Jiri Schindler, *Tranquil Data*

Steering Committee

Nitin Agrawal, *Google*

Marcos K. Aguilera, *VMware Research*

Casey Henderson, *USENIX Association*

Dean Hildebrand, *Google*

Kimberly Keeton, *Google*

Geoff Kuenning, *Harvey Mudd College*

Arif Merchant, *Google*

Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology) and Virginia Tech*

Don Porter, *The University of North Carolina at Chapel Hill*

Raju Rangaswami, *Florida International University*

Erik Riedel

Jiri Schindler, *Tranquil Data*

Bianca Schroeder, *University of Toronto*

Keith A. Smith, *MongoDB*

Eno Thereska, *Alcion*

Carl Waldspurger, *Carl Waldspurger Consulting*

Hakim Weatherspoon, *Cornell University*

Brent Welch, *Google*

Ric Wheeler, *Facebook*

Gala Yadgar, *Technion—Israel Institute of Technology*

Erez Zadok, *Stony Brook University*

Message from the FAST ’23 Program Co-Chairs

Welcome to the 21st USENIX Conference on File and Storage Technologies (FAST ’23).

This year’s conference 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 has adapted with the progression of the COVID pandemic. Two years back, FAST ’21 was held as a fully virtual conference for the first time. Last year, FAST ’22 was held as a hybrid (both in-person and online) conference. This year, we are happy to announce that the FAST ’23 conference is being held fully in person, and we are expecting participation from much of the broader storage community at the conference. However, the pandemic isn’t over yet, and some of the authors are unable to travel to the conference, so we are offering both in-person and video talks with online Q&A.

We have worked on a program with talks on a wide range of topics, including emerging and traditional storage technologies, cloud and remote storage, key-value stores, persistent memory systems, storage coding, learned storage systems, and, as always, new file system designs. The conference will also include posters and work-in-progress sessions.

FAST ’23 received 122 submissions from authors in academia, industry, government labs, and the open-source communities. Of these, we accepted 28 papers, for an acceptance rate of 23%. The Program Committee (PC) used a two-round online review process. In the first round, each paper was assigned three reviewers. This year, we adopted an early rejection notification for papers that did not advance to round two, allowing authors to receive and act upon feedback earlier. In the second round, 63 papers were assigned at least two more reviews, and these authors were invited to submit a response to the reviews before the PC meeting. This is the third year that FAST has included an author response period. After the author response period and online discussion, in which we pre-accepted 13 papers, the PC discussed 28 papers to select the final program. We held a two-day online PC meeting on December 5–6, 2022, with PC members joining virtually from global locations across 11 different time zones.

We used the HotCRP service to manage all the stages of the review process, from submission to author notification. 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.

We continued including a special category of deployed-systems papers, which address experience with the practical design, implementation, analysis, or deployment of large-scale, operational systems. We received nine deployed-systems submissions and accepted four such papers.

We wish to thank the many people who contributed to this conference. First and foremost, we are grateful to all the authors who submitted their work to FAST ’23. We would also like to thank the attendees of FAST ’23 and the future readers of these papers. Together with the authors, you form the FAST community and make storage research vibrant and exciting.

We extend our thanks to the entire USENIX staff, who have provided outstanding support throughout the planning and organizing of this conference with the highest degree of professionalism and friendliness. Most importantly, their behind-the-scenes work makes this conference happen.

We would like to thank the Work-in-Progress Session Chairs, Aishwarya Ganesan and Ram Alagappan. Our thanks 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 especially wish to acknowledge our Steering Committee Liaison, Keith Smith, for his guidance on tough issues and encouragement on various issues over the past year.

Finally, we wish to thank our Program Committee for their many hours of hard work reviewing, discussing, and shepherding the submissions. The reviewers’ evaluations, and 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 interesting and enjoyable conference!

Dalit Naor, *The Academic College of Tel Aviv–Yaffo*
Ashvin Goel, *University of Toronto*
FAST ’23 Program Co-Chairs

21st USENIX Conference on File and Storage Technologies (FAST '23)
February 21–23, 2023
Santa Clara, CA, USA

Tuesday, February 21

Coding and Cloud Storage

Key-Value Stores

AI and Storage

Wednesday, February 22

File Systems

Persistent Memory Systems

Remote Memory

Thursday, February 23

IO Stacks

SSDs and Smartphones

Practical Design Considerations for Wide Locally Recoverable Codes (LRCs)

Saurabh Kadekodi*, Shashwat Silas*, David Clausen, Arif Merchant
Google

Abstract

Most of the data in large-scale storage clusters is erasure coded. At exascale, optimizing erasure codes for low storage overhead, efficient reconstruction, and easy deployment is of critical importance. *Locally recoverable codes (LRCs)* have deservedly gained central importance in this field, because they can balance many of these requirements. In our work we study wide LRCs; LRCs with large number of blocks per stripe and low storage overhead. These codes are a natural next step for practitioners to unlock higher storage savings, but they come with their own challenges. Of particular interest is their *reliability*, since wider stripes are prone to more simultaneous failures.

We conduct a practically-minded analysis of several popular and novel LRCs. We find that wide LRC reliability is a subtle phenomenon that is sensitive to several design choices, some of which are overlooked by theoreticians, and others by practitioners. Based on these insights, we construct novel LRCs called *Uniform Cauchy LRCs*, which show excellent performance in simulations, and a 33% improvement in reliability on unavailability events observed by a wide LRC deployed in a Google storage cluster. We also show that these codes are easy to deploy in a manner that improves their robustness to common maintenance events. Along the way, we also give a remarkably simple and novel construction of distance optimal LRCs (other constructions are also known), which may be of interest to theory-minded readers.

1 Introduction

Large-scale storage clusters currently house exabytes of data, the bulk of which is encoded with erasure codes. With storage devices (hard-disk drives, or just disks) routinely becoming unavailable (due to maintenance or even failures), using erasure coding of some variety is essential to provide acceptable data durability. But this durability comes with the additional storage overhead incurred by erasure codes. At a time when data corpus size is growing exponentially [9, 43], reducing this storage overhead is essential. One way to accomplish this is to utilize wider stripes for encoding i.e. codes that have a higher ratio of data blocks to coded/redundancy blocks. Such codes are sometimes called ‘wide codes’ since they require the overall width of the stripe to be larger (width is also referred to as *blocklength* in the coding theory literature). To unlock large storage savings without compromising reliability, codes of larger widths like 20 [3] have been deployed, and in this work we present some data from a Google storage cluster using an erasure code of width ≈ 80 blocks. The drawback of most wide codes with low overhead is that they may require a large amount of IO to reconstruct any unavailable or lost data. This is why wide codes are usually designed as *locally recoverable codes (LRCs)* (Definition 5.3), which help mitigate this reconstruction cost in most cases. Wide LRCs can be utilized to balance the challenging storage needs of today (low storage overhead, competitive reliability, competitive average reconstruction cost), but unique challenges arise when designing *wide* LRCs for deployment in storage clusters.

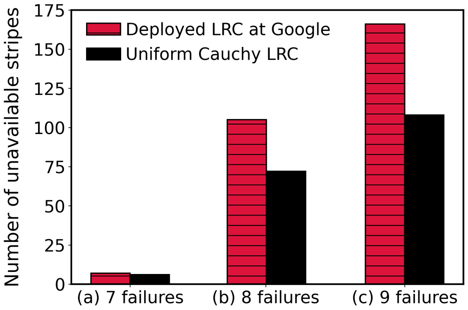


Figure 1: We captured a sample of 278 unavailable stripes captured from four Google storage clusters, along with information about the exact block failures in each sample. The deployed code had total width $n \approx 50$, and always succeeded in recovering data when there were ≤ 6 failures. We then test these failure scenarios with the Uniform Cauchy LRC of the same width and overhead. The deployed code could not recover any of the 278 stripes before restoration, whereas Uniform Cauchy LRC simulation was successful in recovering 92 stripes prior to restoration; a success ratio of 33%.

LRCs are optimized to shine in the case when there is a single erasure in a stripe, but much effort has gone into designing LRCs with various other desirable properties [17, 26, 27, 33, 47]. One obvious direction that has received much attention is to design *distance-optimal LRCs* [33, 49] – that is, LRCs with the best possible *distance*, given their width, dimension, and locality. A code with distance d guarantees recovery from any pattern of up to $d-1$ failures. Indeed, maximizing distance is especially important for wide codes, since they are more likely (simply due to their width) to encounter a larger number of failures simultaneously, as we show using data from Google storage clusters in Figure 2. But we find that even if we use distance-optimal LRCs, storage clusters using wide codes encounter a meaningful number of events where the number of failures is *larger* than the distance of the

[*] Equal contribution

distance-optimal LRC (see data from a Google storage cluster in Figure 1). So it becomes important to study LRCs that can successfully reconstruct even a significant fraction of erasure patterns *beyond the optimal distance*. The theory community has been tackling this problem by studying *maximally recoverable locally recoverable codes (MR-LRCs)*, which take the erasure correction capability of an LRC to the information theoretic limit implied by its design parameters [16, 18, 20]. However, to the best of our knowledge, current constructions of MR-LRCs do not yield codes fitting the limitations of current hardware. For example, wide MR-LRCs with storage overhead <20% require orders of magnitude larger field sizes than the computationally efficient field sizes of up to $\mathbb{F}_{256}$, which is explained in detail in Section 4.

Optimizing LRC reliability in practical parameter settings is a valuable and open problem, and we tackle aspects of it in this work. We highlight some of our main contributions.

Practical measurement of reliability. We study wide LRCs with a distinctively practical lens. One of our contributions is the curation of a set of *robust and practical* measures of LRC reliability. These include performance against random erasures, comparing the reliability of explicit LRC generator matrices (see Definition 5.1) against the information theoretic limit provided by MR-LRCs, and calculating *mean time to data loss (MTTDL)* using *observed* reliability metrics. As mentioned earlier, going wide creates new reliability challenges for LRCs, and these tools provide a clearer and more realistic set of tests to tackle the new challenges. Using these tools, we meaningfully compare popular deployed LRCs and showcase novel highly performant LRCs.

New distance-optimal LRC constructions. In Section 6, we provide a novel construction of distance-optimal LRCs (Definition 5.4), which we call *Optimal Cauchy LRCs*. This continues a long line of work [47, 49] for constructing distance-optimal LRCs, and while our construction is not the most general, it is remarkably simple and yields many codes (even wide LRCs) in practically useful parameter settings.

Insights into reliable wide LRC design. Our practical measures of reliability provide several insights. While it is true that higher distance gives guarantees about fixing up to a certain number of erasures, it is not enough by itself to guarantee strong empirical results. The first improvement that can be made in this direction is to find codes that are *approximately MR-LRCs* (since it is not yet possible to construct true MR-LRCs in our parameter regimes). Here we show good news: the coefficients used in our simple constructions get us over 99% of the reliability possible with MR-LRCs. But we find that just being (close to) maximally recoverable is not the end of the journey for reliability. Indeed, two codes that have the same width and same storage overhead, and are both MR-LRCs (true or approximate) can have significantly different resilience to *random patterns of erasures*. This is because erasure recovery is affected by how failures are distributed across the local repair groups (see Definition 5.5) of the LRC, so it is not enough to just optimize the coefficients of an LRC, but also the *size* of its local repair groups. To the best of our knowledge, this fact has not been considered in the literature, even though our experiments show that it can have a significant impact on reliability. In general, codes whose local groups are *evenly sized* have better performance (see discussion in Section 8). This has the additional perk of also lowering the degraded mode read cost and the reconstruction cost of the code.

Novel LRCs that excel in practice. Using some of these insights, we modify our construction of distance-optimal codes to create *Uniform Cauchy LRCs*. We find that these codes truly shine in most practically relevant reliability (and performance!) measures. Figure 1 shows data unavailability events from a deployed wide LRC of width ≈ 50 blocks along with their erasure patterns captured from four large storage clusters at Google with a total disk population of over 1.7 million disks, over a period of one year. For the same code width and storage overhead, our Uniform Cauchy LRC simulation recovered more than 33% of these stripes without the need for restoration. Indeed, further experiments confirm this observation by showing that Uniform Cauchy LRCs outperform many popular (and deployed) LRCs across our metrics. A comprehensive experimental evaluation of LRCs is provided in Section 8, along with the main observations.

Maintenance-robust deployment of wide LRCs. In Section 9 we highlight the importance of maintenance-robust deployment of wide LRCs. Even though a code may have many desirable properties, its exact layout in a cluster affects its robustness to common maintenance events such as kernel upgrades. It is desirable to construct codes that are easier to deploy in a maintenance-robust manner (not all codes are equal here), and we show that the design of Uniform Cauchy LRCs is ideal in this regard.

Our work shows that a myriad of design choices need to be considered in order to optimize wide LRCs. Indeed, accounting for these factors can lead to more reliable deployments of wide LRCs in practice.

2 Background

Large-scale storage clusters. Large-scale storage clusters typically constitute of public cloud offerings or high-performance computing systems. Hard-disk drives (HDDs) make up the primary storage tier in these clusters. It is common for the disk population in a large-scale cluster to be above 100K [2, 32], and some large ones are also reported to have close to 500K disks [30, 31]. Data being stored in large-scale storage clusters is increasing at an alarming rate [7, 9, 10, 41]. Data redundancy using erasure coding is the reliability mechanism of choice for bulk of the data.

Erasure coding. Erasure coding is a more space-efficient alternative to data replication. Usually described as an (n,k) code, an erasure coding *stripe* encodes k data blocks (typically one, or a few megabytes in size) along with $n-k$ 'parity' blocks (of the same size) to form an n block *stripe*. The storage overhead is calculated as $\frac{n}{k}$. Maximum Distance Separable (MDS) codes (like Reed-Solomon codes) are popular erasure codes used in practice because they provide the maximum erasure correction capability for any fixed value of n and k. MDS codes have the property that up to $n-k$ blocks missing from a stripe can be reconstructed using *any* k of the remaining blocks. As data has grown and space-efficiency has become more critical, *wider* MDS codes (i.e. codes with larger values of k) with less storage overhead have become more popular in practice. Indeed, even $(20,17)$ MDS codes have been deployed and studied [3], in place of the once ubiquitous schemes like RAID-6 (which is a $(6,4)$-Reed-Solomon code), (9, 6) [13] and (14, 10) [42]. But wider MDS codes have their own drawbacks because they require all the data from k blocks to reconstruct/repair even a single missing block, leading to a high *reconstruction cost*. A need to use wider encoding schemes with less overhead, combined with the very high reconstruction cost of MDS codes has motivated the study of Locally Recoverable Codes.

Data reconstruction process. Large-scale cluster storage systems have a background process that monitors the redundancy level of all stripes stored in the cluster. Whenever a disk becomes unavailable (either due to server unavailability, or disk failure), the background daemon flags the under-redundant stripe and starts a timeout of a few tens-of-minutes. When the timeout expires, the stripe is marked for reconstruction. Storage clusters often set a soft threshold on the bandwidth used for background activity such as reconstructions (except client-initiated degraded mode reads). In order to balance the reconstruction workload while maintaining highest data safety standards, the reconstructions are processed via a priority queue in which stripes that are more vulnerable are reconstructed before less vulnerable stripes.

Locally Recoverable Codes (LRCs). LRCs [17, 27, 33, 42, 47, 49] (also known as Local Reconstruction Codes) are erasure codes designed to mitigate the high reconstruction cost of MDS codes. An (n,k,ℓ) LRC code divides an n block stripe into local groups, each with at most $\ell < k$ blocks and a local parity[1]. In addition to the local parities, the stripe also has *global parities* which cover all k data blocks. One may think of a *typical* LRC in this way: the data blocks and the global parities together form an MDS code, and the local parities are added on top of this code, so as to mitigate the cost of reconstruction (reduce it from k to ℓ) in the case when there is *exactly* one failure in a local group. One may note that LRCs are not MDS codes since they cannot satisfy the Singleton bound unless $\ell = k$ (see Definition 5.6). If there is more than one failure in the same local group, the underlying MDS code can be used to reconstruct the data. Several different LRC constructions have been proposed over the years with different trade-offs [27, 33, 47, 49].

Distance of a code. Distance of a code (denoted by d) is the minimum number of failures/erasures that may render the stripe potentially non-recoverable (i.e. all patterns of $< d$ failures are always recoverable). For example, the distance of an (n,k) MDS code would be $n-k+1$ since an MDS code can recover from any $n-k$ failures. In fact, for an MDS code, any failure beyond $n-k$ failures is strictly non-recoverable. However, for a code with distance d which is *not* MDS (such as an LRC), some patterns of $\geq d$ failures could be recovered. Maximizing this capability to recover as many erasure patterns as possible *beyond the distance* leads us to the notion of *maximally recoverable codes*.

Maximally Recoverable LRCs (MR-LRC). For any code, simply specifying whether each entry in its generator matrix (see Definition 5.1) is zero or non-zero imposes an information theoretic limit on which patterns of erasures could be recoverable. The study of *maximally recoverable codes* is concerned with finding coefficients for the non-zero entries such that this limit is reached [16]. LRCs can also be optimized in this way: once we have specified which entries of the generator matrix are non-zero (this will fix various code parameters like n, k, number of local parities, number of global parities, and the size of the local groups), it is possible to find coefficients that maximize the number of recoverable erasure patterns (including potentially many patterns of $\geq d$ erasures). LRCs which are *maximially recoverable* are known as MR-LRCs. In terms of their reliability, MR-LRCs are the gold standard among LRCs.

3 Motivation for studying wide LRCs

Reducing storage overhead is critical. Storage overhead because of data redundancy is a major component of a cluster's storage cost. With 3-way replication, the overhead is 3×, which is prohibitive for large-scale clusters storing exabytes (EBs) of data on hundreds-of-thousands of disks. Even the popular MDS codes such as $(9,6)$ [13] or $(14,10)$ [42] which have an overhead of 1.5× and 1.4× respectively, are considered too expensive for exascale [9, 10, 29, 41]. Large-scale storage clusters are actively adopting *wide MDS codes* to minimize the storage overhead. Backblaze reported the use of a $(20,17)$ MDS code [3] which has a reasonably low overhead of 1.17× (a rate of $\frac{17}{20} = 0.85$, see Definition 5.1). With every percent reduction in storage overhead resulting in savings of millions in capital, operational and energy costs, lowering storage overhead continues to be a lucrative problem.

Wide MDS codes are costly. Using wide MDS codes has several challenges. The reconstruction IO cost of a wide (n,k) MDS code scales linearly with k. For example, while a $(9,6)$ MDS codes requires reading 6 data blocks for reconstructing a missing block, a wide MDS code such as $(20,17)$ requires reading 17 data blocks. Wide MDS codes also have a higher unavailability because they have a higher probability

[1] Theoretically each local group can have any number of parities, but in practice the most common configuration involves 1 parity per local group.

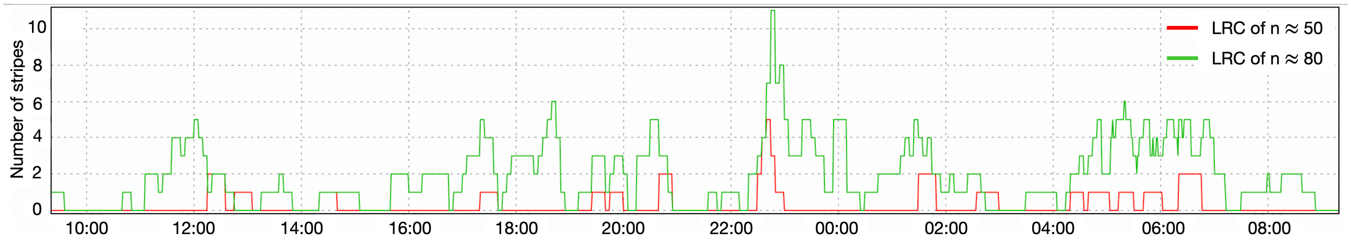


Figure 2: A 24 hour trace with the number of stripes with at least 4 failures in two deployed LRCs captured from a single storage cluster at Google. The wider LRC ($n \approx 80$) has many more stripes with at least 4 failures compared to the relatively narrower LRC ($n \approx 50$).

of having blocks stored on devices that are unreachable due to maintenance (since no two blocks of a stripe can be on the same disk, server, rack, etc.). Thus, not only does a wide MDS code have a higher degraded mode read cost (Definition 5.10), but the frequency of performing degraded-mode reads is also higher due to higher unavailability. Due to a larger number of disks, degraded reads and reconstructions in wide MDS codes are also more likely to suffer from high tail latency due to stragglers. So although wide MDS code are good for durability and reduced storage overhead, they are not good for reconstruction costs, availability, and degraded read performance, all of which are major performance concerns in large-scale storage clusters. Storage overhead minimization is critical, but cannot come at the expense of aforementioned problems. Indeed, this is the reason for the continued popularity of LRCs, which mitigate this drawback.

Real-world failure patterns favor LRCs. We empirically observe the failure patterns of stripes stored in three large-scale storage clusters totalling over 1.5 million disks for a period of 6 months. These clusters have multiple different erasure coding and replication schemes deployed simultaneously, and have over 1.2 trillion stripes. From among the stripes that have > 0 failures, we observe that $\approx 99.2\%$ of stripes have just a single failure. Similar data has been observed by others in [40], where they found that $\approx 98.08\%$ stripes had a single failure on a Facebook warehouse cluster. A single block failure in a stripe is a scenario where LRCs shine (in contrast to MDS codes). However, this comes at a cost of higher storage overhead (as we mentioned, typical LRCs are just MDS codes with the additional overhead of local parities, and they cannot lie on the Singleton bound).

Wide LRCs. In an ideal world, we would like to use *wide* LRCs which have significantly lower overhead than the LRCs showcased in [33, 42] (and many other works). Wide LRCs could reduce the overhead from the $30-60\%$ range (which is common in deployed LRCs and MDS codes) to the less than 20% range, while still maintaining many advantages over MDS codes. But wide LRCs create novel challenges. For example, wider stripes are much more likely to result in a larger percentage of stripes with more than a single failure. This makes it critical to study *robust and practical* measures of LRC reliability if we are to utilize wide codes in practice. Further, practical issues like deployment and ease-of-use also need to be addressed.

4 Practical challenges of wide LRCs

Many simultaneous failures are more common in wide LRCs. While most stripes (which have failures) have single-block failures, this does not mean that the tiny fraction of more than one block failures can be ignored. Moving to wider stripes increases the possibility that multiple blocks of a stripe need to be repaired at the same time, since there is a higher chance that devices or servers storing multiple blocks of the same stripe may be undergoing maintenance simultaneously. This is consistent with the observation that most data unavailability events in large-scale storage clusters are as a result of planned outages [13]. Another reason for increased number of block failures in a wide stripe is due to prioritization of reconstructions as explained in Section 2. Figure 2 shows the number of stripes that have at least 4 failures throughout a 24 hour period. There are two LRCs whose stripe failures are being compared, one with width approximately 50 blocks and the other with width approximately 80 blocks[2]. Note that this trace is collected from a single storage cluster, and so the failures seen in this trace are in response to the same machines failing. As is clearly visible, the wider LRC has a significantly higher number of stripes with at least 4 failures compared to the relatively narrower LRC.

Constructing MR-LRCs is hard. As mentioned above, a lot of research has looked at MR-LRCs in recent years [5, 12, 16, 18, 20]. However, even with recent advancements such as [12, 18], to the best of our knowledge it is not possible to construct MR-LRCs in the following regime: where the field size (search-space for coefficients to construct the LRC) is fixed to be 256, n is between 25-150, and the rate of the code (see Definition 5.1) is at least 0.85 (i.e. storage overhead is at most $1.17\times$ and $\ell < k$). This setting is particularly useful for applications, and is the natural next phase for practical LRCs which reduce overhead and maximize reliability.

Typically, maximally recoverable codes are easier to construct when the field is large, and much research is focused on finding explicit codes over small field sizes. In practical settings, most computations are done over individual bytes which restricts the field size of the LRCs we can use to 256 (the field $\mathbb{F}_{2^8}$). To the best of our knowledge, the current state of the art constructions are in [18], and they require that the

[2] We cannot disclose the exact configuration due to confidentiality.

field size q be at least $\sim (\ell+1)^r$, where $\ell+1$ is the size of a local repair group (Definition 5.5), and r is the number of global parity checks. For our purposes, we can think of $\ell+1 \sim 16$, and $r \geq 3$, thus $q = 4096$, making the required field size 16× greater than 256.

Measurements of reliability of wide LRCs are inadequate. While narrow LRCs have received a lot of attention in systems research, wide LRCs have not [25]. Existing systems research on LRCs optimizes metrics such as distance, degraded read cost and reconstruction cost [6,27,33,42,47,49]. We enhance this rich literature with a larger suite of *empirical* measures, which give more realistic comparisons in the practical realm, particularly in the case when there are $\geq d$ erasures in a code with distance d. Further, since explicit MR-LRCs are not available in our parameter regimes, it leaves open the question of evaluating whether we can construct codes which offer us *approximately* the same advantages as MR-LRCs could offer (the answer is yes!). Finally, some design choices such as creating evenly sized local groups have not been considered in the literature at all, which we show can have a large impact on performance against random patterns of erasures.

Deployment of wide erasure codes is non-trivial. Another practical hurdle that gets worse with code width is the deployment (placement of blocks) of an erasure coded stripe. Recall that no two blocks of an erasure coded stripe can reside on the same disk, rack, fault-domain, power-source, etc. As the code width increases, it becomes progressively harder to fulfill these placement constraints. The placement problem is further exacerbated because it is common for sets of servers/racks (which comprise a *maintenance zone*) to be down at the same time for maintenance events, potentially causing data unavailability. For example, if too many blocks of a stripe are in the same maintenance zone, then even a planned maintenance event could make the stripe unavailable. We comment on some ways that code design can make it easier to achieve maintenance robust layouts of data.

5 Definitions

We begin by defining important terms that are going to be useful when describing the wide LRC construction in Section 6.

Definition 5.1 (Linear error correcting code). *A linear error-correcting code is simply a subspace $\mathcal{C} \leq \mathbb{F}_q^n$ where q is a prime power and $n > 0$. It is customary to think of $\mathcal{C}$ as the image of an encoding map Enc: $\mathbb{F}_q^k \to \mathbb{F}_q^n$ for some $k \leq n$. This encoding may the expressed in matrix form as,*

$$Gx = y$$

where G is an $n \times k$ matrix called the generator matrix, x is the message and y is the codeword. The fraction $\frac{k}{n}$ is the rate of the code, k is the dimension, and n is the blocklength. The symbols $y_i \in \mathbb{F}_q$ of a codeword y are called codeword symbols.

Definition 5.2 (Distance of a code). *The minimum distance of a linear code (often just called the distance) is simply,*

$$\min_{x \in \mathcal{C} \setminus \{0\}} wt(x)$$

where $wt(x) = \sum_{i=1}^n 1_{x_i \neq 0}$. An error correcting code with distance d can always correct $d-1$ erasures.

Definition 5.3 ((n,k,ℓ)-Locally recoverable code (LRC)). *An (n,k,ℓ)-LRC is a linear error correcting code of dimension k and blocklength n. It has the additional property that any codeword symbol can be recovered from at most ℓ other codeword symbols. The parameter ℓ is called the locality paramter of the LRC. Note that $1 \leq \ell \leq k$.*

Definition 5.4 (Generalized Singleton bound and distance optimal LRCs). *Any locally recoverable code must satisfy the following bound [17],*

$$n \geq k + \left\lceil \frac{k}{\ell} \right\rceil + d - 2$$

where d is the distance of the code. Any LRC that meets this bound with equality is called a distance optimal LRC. Note that when $\ell = k$, this reduces to the well-known Singleton bound $n \geq k + d - 1$.

Definition 5.5 (Local repair group). *Given an (n,k,ℓ)-LRC, choose any codeword y. Due to the local recovery property, any codeword symbol y_i can be recovered using at most ℓ other codeword symbols $\{y_{r_1}, \ldots, y_{r_\ell}\}$ where $r_j \neq i$ for any j. These at most $\ell+1$ indices in $\{i, r_1, \ldots, r_\ell\}$ form a local repair group of the (n,k,ℓ)-LRC.*

Definition 5.6 (Maximum distance separable code). *Any linear error-correcting code which satisfies the Singleton bound with equality, i.e. satisfies $n = k + d - 1$, is known as a Maximum Distance Separable (MDS) code.*

Definition 5.7 (Cauchy matrix). *Let $\{x_1, \ldots, x_n\}$ and $\{y_1, \ldots, y_m\}$ be two disjoint sequences of distinct elements from $\mathbb{F}_q$. The $n \times m$ matrix defined as*

$$C_{ij} = \frac{1}{x_i + y_j}$$

is a Cauchy matrix. It is well-known that every square submatrix of a Cauchy matrix has non-zero determinant (and therefore, is invertible).

Fact 5.8 (Generator matrix of an MDS code). *Cauchy matrices are commonly used to design generator matrices for MDS codes. Indeed, an $n \times k$ matrix over $\mathbb{F}_q$, whose first k rows form I_k (the $k \times k$ identity matrix), and whose last $n-k$ form an $(n-k) \times k$ Cauchy matrix generates an MDS code of blocklength n and dimension k (see Thm 2.2 and 5.2 in [8]).*

We can visualize several of our definitions using the example shown in Figure 3.

$$G_C = \left(\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ c_{11}+c_{21} & c_{12}+c_{22} & c_{13}+c_{23} & c_{14}+c_{24} \end{array}\right) \begin{array}{l} d_1 \\ d_2 \\ d_3 \\ d_4 \\ c_1 \\ c_2 \\ l_1 \\ l_2 \\ l_g \end{array}$$

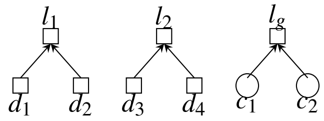


Figure 3: Example of an LRC with 4 data blocks (d_1,d_2,d_3,d_4), 2 global parities (c_1,c_2), and 3 local parities (l_1,l_2,l_g). This figure shows how each row of the generator matrix correponds to a 'block' in the picture representation of an LRC (the reader may note that the matrix representation is more informative as the specific choice of coefficients can affect reliability in practice). The first 4 rows of the matrix correspond to the data blocks which are encoded systematically; the next two rows correspond to the global parities (these two rows form a 2×4 Cauchy matrix in our implementation); the last 3 rows correspond to 3 local parities, all of which contain the XOR of the data they cover (as is evident by the coefficients of the matrix G_C). The last local parity, l_g, only covers the global parities. This design of LRC may be familiar to some readers as an Azure-LRC+1. The *local groups* of this code are (d_1,d_2,l_1), (d_3,d_4,l_2), and (c_1,c_2,l_g) and therefore $\ell = 2$. In this code, *ADRC* (see Def. 5.10) = ARC_1 (see Def. 5.11) = 2, since each data or parity block can be reconstructed by reading the two other blocks in its local repair group.

Definition 5.9 (Maximally recoverable locally recoverable code (MR-LRC))**.** *An (n,k,ℓ)-LRC is called a MR-LRC if it can recover from any pattern of $n-k$ erasures as long as there is at least one erasure in each local repair group of the code. To clarify, suppose the (n,k,ℓ)-LRC has local repair groups $L_1,\dots,L_p$ for some $p > 1$ where each $L_i \subseteq \{1,\dots,n\}$. Recall that $\cup_i L_i = \{1, \dots,n\}$ and that the L_i are not necessarily disjoint. Let $E = \{e_1,\dots,e_{n-k}\} \subseteq \{1,\dots,n\}$ be any pattern of $n-k$ erased symbols. Then as long as $L_i \cup E \neq \emptyset$ for any i, then the erasure pattern E can be recovered by the code.*

Definition 5.10 (Average degraded read cost (ADRC))**.** *The ADRC is the average of the cost of reconstructing any of the data blocks. We can measure $cost(b_i)$ as the number of blocks which need to be read in order to reconstruct block i.*

$$ADRC = \frac{\sum_{i=1}^{k} cost(b_i)}{k}$$

Definition 5.11 (Average repair (or reconstruction) cost (ARC))**.** *The average repair cost is defined identically to the ARDC, with the addition of the global and local parity blocks to the computation.*

$$ARC_1 = \frac{\sum_{i=1}^{n} cost(b_i)}{n}$$

For MDS codes, $\forall b_i, cost(b_i) = ARC = k$. For LRCs, $cost(b_i)$ may not be the same as ARC. The above ARC is defined for one block failing in a stripe. In our evaluation we also show the ARC of reconstructing two failed blocks defined as

$$ARC_2 = \frac{\sum_{i=1,j\neq i}^{n} cost(b_{i,j})}{\binom{n}{2}}$$

where $1 \leq i \leq n, 1 \leq j \leq n$ and $i \neq j$, and $cost(b_{i,j})$ = cost to reconstruct blocks i and j which can result in a combination of local and global reconstructions depending on i and j.

6 (n,k,r,p)-Optimal Cauchy LRCs

In this section we construct (n,k,r,p)-*Optimal Cauchy LRCs* and discuss their advantages. Here, n is the blocklength of the code, k is the dimension, r is the number of global parity checks, and p is the number of local parity checks. We make two design choices which are worth pointing out. First, the p local parities in our code are uniformly distributed across the k data, and they are all XOR-ed with the r global parity checks (this helps with proving the distance optimality of the code). Second, we restrict ourselves to the case where each of the data symbols of the code is covered by exactly *one* local parity. The second restriction is common in constructions proposed in the literature, and indeed, important for our goal of minimizing the storage overhead. For simplicity of exposition we assume that p is even (we will mention how this condition can be removed), and $p|k$ where we denote $\frac{k}{p} = t$. It will be clear that the *locality* parameter ℓ of the (n,k,r,p)-Optimal Cauchy LRCs we will construct is $\frac{k}{p} + r$.

In Section 6.2 will show that (n,k,r,p)-Optimal Cauchy LRCs have the best possible distance for LRCs with their dimension and locality (under some modest restrictions). In Appendix A.2 we show how some of these restrictions can be relaxed. We begin with explaining the construction of (n,k,r,p)-Optimal Cauchy LRCs.

6.1 Code construction

The generator matrix for an (n,k,r,p)-Optimal Cauchy LRC is derived naturally from the generator matrix for an $(k+r+1,k)$-MDS code which has dimension k and blocklength $n = k+r+1$. Specifically, we derive our code from the generator matrix of a $(k+r+1,k)$-cauchy MDS code.

The generator matrix $G_{(k+r+1),k}$ of a $(k+r+1,k)$-Cauchy MDS code is an $k+r+1 \times k$ matrix, which simply consists of a $k \times k$ identity matrix stacked on top of a $r+1 \times k$ Cauchy matrix. The resulting matrix is well-known to generate an MDS code with distance $r+2$ (See Thm 2.2 and 5.2 in [8]).

$$G_{(k+r+1),k} = \left[\begin{array}{ccc} 1 & & \\ & \ddots \quad 0 & \\ & \ddots & \\ 0 & & \ddots \\ & & 1 \\ \hline c_{11} & \cdots & c_{1k} \\ & \ddots & \\ c_{r1} & & c_{rk} \\ c_{(r+1)1} & & c_{(r+1)k} \end{array}\right]$$

When we do not need to refer to r and k are specifically, we refer to $G_{(k+r+1),k}$ as G, and the ith row of $G_{(k+r+1),k}$ as g_i.

To create the generator matrix for an (n,k,r,p)-optimal cauchy LRC, we first partition this last row of G, g_{k+r+1} into p rows $r_1,\ldots,r_p$ as follows,

$$\begin{aligned} r_1 &= (c_{(r+1)1}, c_{(r+1)2}, \ldots, c_{(r+1)t}, 0, \ldots, 0) \\ r_2 &= (0, \ldots, 0, c_{(r+1)(t+1)}, \ldots, c_{(r+1)2t}, 0, \ldots, 0) \\ &\vdots \\ r_p &= (0, \ldots, 0, c_{(r+1)(p(t-1))}, \ldots, c_{(r+1)pt}) \end{aligned}$$

It is clear that $r_1 + r_2 + \cdots + r_p = g_{k+r+1}$. We the compute the rows $\tilde{r}_i$ as follows,

$$\tilde{r}_i = r_i + g_{k+1} + g_{k+2} + \cdots + g_{k+r}$$

Note that $\tilde{r}_i$ are simply r_i with the addition of the first r 'cauchy rows' of G. Since p is even (and the underlying field has characteristic 2) we have,

$$\begin{aligned} \tilde{r}_1 + \tilde{r}_2 + \cdots + \tilde{r}_p &= p(g_{k+1} + g_{k+2} + \cdots + g_{k+r}) + g_{k+r+1} \\ &= g_{k+r+1} \end{aligned}$$

Finally, the generator matrix $O_{n,k,r,p}$ for the (n,k,r,p)-Optimal Cauchy LRC is given by,

$$O_{n,k,r,p} = \left[\begin{array}{ccc} 1 & & \\ & \ddots \quad 0 & \\ & \ddots & \\ 0 & & \ddots \\ & & 1 \\ \hline c_{11} & \cdots & c_{1k} \\ & \ddots & \\ c_{r1} & & c_{rk} \\ & \tilde{r}_1 & \\ & \vdots & \\ & \tilde{r}_p & \end{array}\right]$$

This matrix is simply the $k+r$ rows of $G_{k+r+1,k}$ stacked on top of the $\tilde{r}_1, \tilde{r}_2, \ldots, \tilde{r}_p$ vectors. We remark that $n = k+r+p$, and that $O_{n,k,r,p}$ generates an LRC with locality parameter $\ell = \frac{k}{p} + r$. This is clear since each row among $\tilde{r}_1, \tilde{r}_2, \ldots, \tilde{r}_p$ provides a local parity check on exactly ℓ other rows.

6.2 Distance

We note that the distance of an (n,k,r,p)-Optimal Cauchy LRC is exactly $r+2$. As long as $k,r,p > 0$, $p|k$, $rp^2 > k+rp$, and p is even, it is a distance optimal LRC (Definition 5.4).

Theorem 6.1. *For $r > 0$, $O_{n,k,r,p}$ generates an error correcting code with distance exactly $r+2$. Proof details are in Appendix A.1*

Lemma 6.2. *Given $k,r,p > 0$, $rp^2 > k+rp$, and p even, the locally recoverable code formed by $O_{n,k,r,p}$ is distance optimal. Proof details are in Appendix A.1*

We show how some of the conditions in Lemma 6.2 can be relaxed in Appendix A.2.

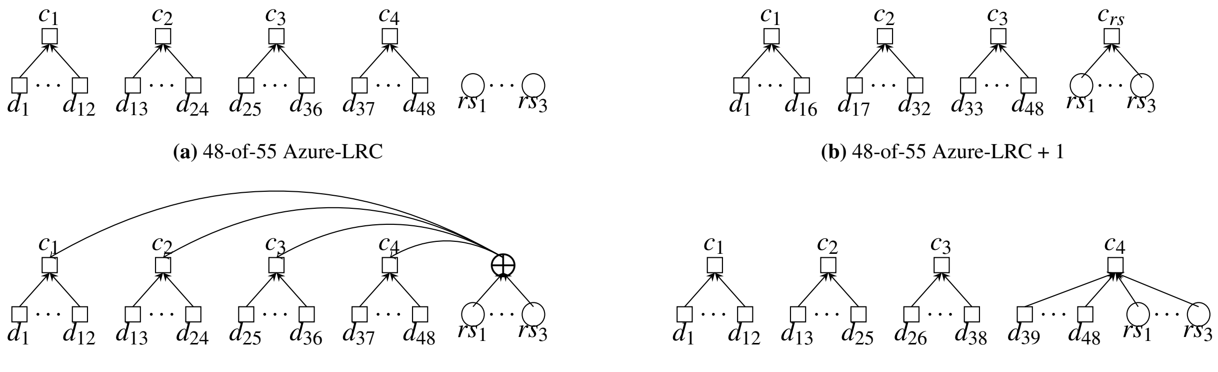
Remark 6.3. *Constructing distance optimal LRCs is a well-studied problem. At first, only constructions whose alphabet size was exponential in the blocklength were known [45, 49]. These do not yield codes which may be used in practical settings where the alphabet size q is fixed at 256. The most well-known construction of distance optimal LRCs may be found in [33, 47]. To get truly optimal codes the codes of [47] require that $\ell+1|n$, and none of the codes we use in our experiments meet this restriction. Distance optimal LRCs with very general parameters were finally shown in [33]. In our experimental analysis in the later sections, we use Optimal Cauchy LRCs (rather than other known constructions such as [33]) as examples of codes which lie on the generalized Singleton bound (essentially, distance optimal LRCs. See Definition 5.4). This is because Optimal Cauchy LRCs are easier to construct in our parameter regime.*

7 (n,k,r,p)-Uniform Cauchy LRCs

In the previous section we give a very simple construction of distance optimal LRCs which we will use in our experimental analysis as examples of distance optimal codes. In this section, we provide a simple heuristic modification of these codes which has several practical advantages (though they are not shown to be optimal with regards to distance).

The later experimental analysis of these codes highlights the point that distance optimal (i.e. on the generalized Singleton bound) does not mean most-durable or most cost-efficient from a practical perspective. For example, a code that has the same locality and distance can have different durability (as measured by mean-time-to-data-loss) and robustness against random patterns of erasures. Distance-optimality simply indicates the best distance for fixed values of n,k and ℓ.

These codes are constructed in much the same way as optimal cauchy LRCs, except that each local parity check covers $\frac{k+r}{p}$ of the data blocks and global parity blocks. Following the same notation as the previous section, the generator matrix for a uniform cauchy code has the form (assuming $p|k+r$ for simplicity),



(a) 48-of-55 Azure-LRC

(b) 48-of-55 Azure-LRC + 1

(c) 48-of-55 Optimal Cauchy LRC

(d) 48-of-55 Uniform Cauchy LRC

Figure 4: Different LRC constructions used in our evaluation.

$$U_{n,k,r,p} = \left[\begin{array}{ccc} 1 & & \\ & \ddots & 0 \\ & \ddots & \\ 0 & & \ddots \\ & & 1 \\ \hline c_{11} & \cdots & c_{1k} \\ & \ddots & \\ c_{r1} & & c_{rk} \\ & r_1 & \\ & \vdots & \\ & \tilde{r}_p & \end{array}\right]$$

Where (if we denote $\frac{k+r}{p} = t$),

$$r_1 = (c_{(r+1)1}, c_{(r+1)2}, \ldots, c_{(r+1)t}, 0, \ldots, 0)$$
$$r_2 = (0, \ldots, 0, c_{(r+1)(t+1)}, \ldots, c_{(r+1)2t}, 0, \ldots, 0)$$
$$\vdots$$
$$r_p = (0, \ldots, 0, c_{(r+1)(p(t-1))}, \ldots, c_k)$$

Finally, we modify the last row by adding the exclusive-or of the global parity checks.

$$\tilde{r}_p = r_p + g_{k+1} + \cdots + g_{k+r}$$

One may note that the locality parameter for this code $\ell = \frac{k+r}{p}$, which is lower than that for optimal cauchy LRCs. In the event that $p \nmid (k+r)$, we may simply divide the $k+r$ data and global parities as evenly as possible amongst the p local checks.

8 Experiments and analysis

We now use the following suite of practical measures of LRC quality to compare various codes:

1. Average degraded read cost (Definition 5.10).
2. Average repair costs (Definitions 5.11).
3. Reliability against random erasure patterns.
4. Comparison of reliability against the information theoretic limit (i.e. against MR-LRCs).
5. A practical computation of mean-time-to-data-loss (MTTDL).

We compare well-known LRC constructions including distance optimal LRCs (our own Optimal Cauchy LRCs) and novel codes like our Uniform Cauchy LRCs. We compare these codes for the following parameter settings, all of which have $< 20\%$ storage overhead.

LRCs used in experiments. The most popular deployed LRCs we came across are the Xorbas-LRC [42], Azure-LRC [27] and Azure-LRC+1 [33] constructions. Xorbas-LRC has the advantage of being distance optimal, but it is only shown to be so in very specific parameter settings, so we instead use Optimal Cauchy LRCs as examples of distance optimal codes in our evaluation. The Azure-LRC was one of the first LRCs proposed, and is reportedly used at Microsoft Azure [27]. We illustrate these codes for the 48-of-55 parameter setting in Figure 4.

Azure-LRC divides the data blocks into equally spaced local groups, with each local group having a local parity. Each local parity can be the XOR of the data blocks in that local group. The global parities are not protected by any local parities. Figure 4a gives a representation of the Azure-LRC construction. Note that Azure-LRC is not a *true* LRC since the global parities are not locally recoverable (i.e. $\ell = k$). In early works, locality for global parities was not considered a requirement for LRCs, but most recent works enforce it. How-

Scheme	n	k	r	p	Rate
24-of-28	28	24	2	2	$\frac{24}{28} = 0.857$
48-of-55	55	48	3	4	$\frac{48}{55} = 0.872$
72-of-80	80	72	4	4	$\frac{72}{80} = 0.9$
96-of-105	105	96	5	4	$\frac{96}{105} = 0.914$

Table 1: Wide LRC schemes used to compare different LRC constructions. Each scheme is chosen such that rate $>= 0.85$.

Locality	**Azure-LRC**	**Azure-LRC+1**	**Optimal Cauchy LRC**	**Uniform Cauchy LRC**
$n = 28$, $k = 24$, $r = 2$, $p = 2$	$\ell = 24$	$\ell = 24$	$\ell = 12 + 2 = 14$	$\ell =\mathbf{13}$
$n = 55$, $k = 48$, $r = 3$, $p = 4$	$\ell = 48$	$\ell = 16$	$\ell = 12 + 3 = 15$	$\ell =\mathbf{13}$
$n = 80$, $k = 72$, $r = 4$, $p = 4$	$\ell = 72$	$\ell = 24$	$\ell = 18 + 4 = 22$	$\ell =\mathbf{19}$
$n = 105$, $k = 96$, $r = 5$, $p = 4$	$\ell = 96$	$\ell = 32$	$\ell = 24 + 5 = 29$	$\ell =\mathbf{26}$
Avg. degraded read cost (*ADRC*)	**Azure-LRC**	**Azure-LRC+1**	**Optimal Cauchy LRC**	**Uniform Cauchy LRC**
$n = 28$, $k = 24$, $r = 2$, $p = 2$	$\frac{24\times12}{24} =\mathbf{12}$	$\frac{24\times24}{24} = 24$	$\frac{12\times14}{12} = 14$	$\frac{26\times13}{26} = 13$
$n = 55$, $k = 48$, $r = 3$, $p = 4$	$\frac{48\times12}{48} =\mathbf{12}$	$\frac{48\times16}{48} = 16$	$\frac{48\times15}{48} = 15$	$\frac{13\times12+35\times13}{48} = 12.72$
$n = 80$, $k = 72$, $r = 4$, $p = 4$	$\frac{72\times18}{72} =\mathbf{18}$	$\frac{72\times24}{72} = 24$	$\frac{72\times22}{72} = 22$	$\frac{72\times19}{72} = 19$
$n = 105$, $k = 96$, $r = 5$, $p = 4$	$\frac{96\times24}{96} =\mathbf{24}$	$\frac{96\times32}{96} = 32$	$\frac{96\times29}{96} = 29$	$\frac{78\times25+18\times26}{96} = 25.18$
Avg. repair cost 1 failure (*ARC*$_1$)	**Azure-LRC**	**Azure-LRC+1**	**Optimal Cauchy LRC**	**Uniform Cauchy LRC**
$n = 28$, $k = 24$, $r = 2$, $p = 2$	$\frac{26\times12+2\times24}{28} =\mathbf{12.85}$	$\frac{25\times24+3\times2}{28} = 21.64$	$\frac{28\times13}{28} = 13$	$\frac{28\times13}{28} = 13$
$n = 55$, $k = 48$, $r = 3$, $p = 4$	$\frac{52\times12+3\times48}{55} = 13.94$	$\frac{51\times16+4\times3}{56} = 15.05$	$\frac{55\times15}{55} = 15$	$\frac{13\times12+42\times13}{55} =\mathbf{12.76}$
$n = 80$, $k = 72$, $r = 4$, $p = 4$	$\frac{76\times18+4\times72}{80} = 20.7$	$\frac{75\times24+5\times4}{81} = 22.75$	$\frac{80\times22}{80} = 22$	$\frac{80\times19}{80} =\mathbf{19}$
$n = 105$, $k = 96$, $r = 5$, $p = 4$	$\frac{100\times24+5\times96}{80} = 27.42$	$\frac{99\times32+6\times5}{81} = 30.45$	$\frac{105\times29}{105} = 29$	$\frac{81\times25+24\times26}{105} =\mathbf{25.22}$
Avg. repair cost 2 failures (*ARC*$_2$)	**Azure-LRC**	**Azure-LRC+1**	**Optimal Cauchy LRC**	**Uniform Cauchy LRC**
$n = 28$, $k = 24$, $r = 2$, $p = 2$	30.66	43.46	32.12	**27.92**
$n = 55$, $k = 48$, $r = 3$, $p = 4$	35.49	39.22	36.93	**33.85**
$n = 80$, $k = 72$, $r = 4$, $p = 4$	52.80	59.38	54.82	**49.22**
$n = 105$, $k = 96$, $r = 5$, $p = 4$	70.68	79.73	74.50	**67.69**
Normalized MTTDL comparison	**Azure-LRC**	**Azure-LRC+1**	**Optimal Cauchy LRC**	**Uniform Cauchy LRC**
$n = 28$, $k = 24$, $r = 2$, $p = 2$	0.64×	0.14×	0.50×	**1.00**
$n = 55$, $k = 48$, $r = 3$, $p = 4$	0.99×	0.97×	**1.01**×	1.00
$n = 80$, $k = 72$, $r = 4$, $p = 4$	0.99×	0.97×	0.49×	**1.00**
$n = 105$, $k = 96$, $r = 5$, $p = 4$	0.99×	0.96×	0.96×	**1.00**

Table 2: This table captures all the analytical metrics used to compare the different LRC constructions across the different wide LRC parameters described in Table 1. Details of this comparison are described in Section 8. The takeaways are that for all metrics except average degraded mode read cost (in which it is $< 9\%$ worse than the best LRC), Uniform Cauchy LRCs outperform other LRCs (including the Optimal Cauchy LRC). Another surprising result was that Azure-LRC outperforms Azure-LRC+1 despite its global parities having no local parities protecting them.

ever, we include this code in our analysis due to its popularity and because it is actually used in practice (and our analysis shows that it gives great performance!).

Azure-LRC+1 is an optimization to the Azure-LRC proposed in [33]. Azure-LRC suffers from an expensive MDS-level reconstruction on the failure of any of the global parities. In order to prevent this, Azure-LRC+1 forms a local group of the global parities and protects them using a local parity. Figure 4b captures the Azure-LRC+1 construction. Note that the Azure-LRC+1 construction in [33] introduces an additional local parity to protect the global parities without removing any of the existing parity blocks. This construction changes the storage overhead of Azure-LRC+1 in comparison to other constructions. We deliberately choose to keep the numbers k-of-n the same across all schemes, otherwise we are comparing codes with different redundancy, which in our view is not a fair comparison. Thus, in our construction of Azure-LRC+1, we choose to remove one local parity protecting data blocks (as compared to removing a global parity since removing a local parity has a less adverse effect on the reliability), and add a local parity protecting global parity blocks.

Figure 4c and Figure 4d are representations of the Optimal Cauchy and Uniform Cauchy constructions which have been described comprehensively in previous sections.

Parameter regimes for comparison. We select four different widths representing wide LRCs, details of which are in Table 1. For an apples-to-apples comparison, we freeze the size of the data, k, the size of the code n, the number of global parities r, and the number of local parities, p. Since a $(20, 17)$ MDS code is reportedly in use [3], we explore schemes where $24 \leq n \leq 105$ (deliberately starting from n close to 20, and codes with rate strictly higher than 0.85). The entire set of results are presented in Table 2, but we highlight the main points below.

Uniform Cauchy LRC has the smallest locality. We first compare the locality of the different LRC constructions. Recall (Definition 5.3) that locality refers to the maximum number of blocks to be read for reconstruction of a single block. Since Azure-LRC requires reading all data blocks to reconstruct any failed global parity, its locality is the high-

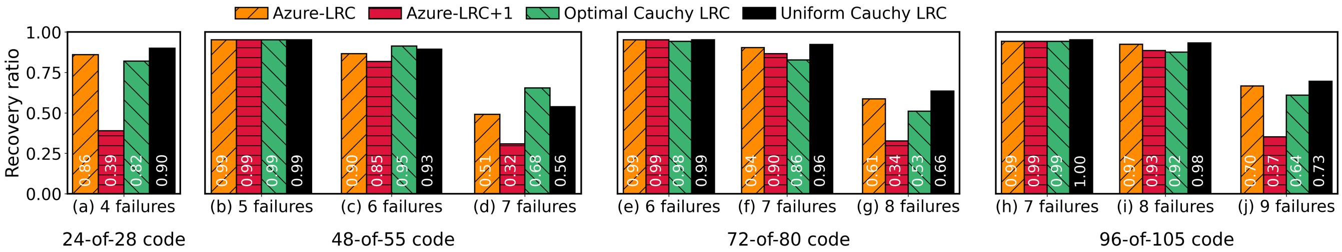


Figure 5: This plot compares the durability of the different LRC constructions (Azure-LRC, Azure-LRC+1, Optimal Cauchy and Uniform Cauchy) by measuring their ability to recover from random failures (only values where some recovery ratios were less than one are shown). We evaluate all wide LRCs discussed in evaluation ranging from 24-of-28 to 96-of-105. Except in 48-of-55, we see that Uniform Cauchy has the best durability. Optimal Cauchy has the best durability in 48-of-55. Surprisingly, Azure-LRC has better durability compared to Azure-LRC+1 even though global parities of Azure LRCs are not covered by a local parity.

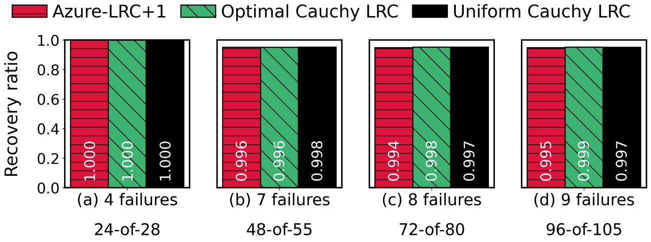


Figure 6: Comparing performance with MR-LRC. In this plot we show the results of a Monte-Carlo experiment where p failures were forced (one in each local group) and another $n-k-p$ failures were distributed randomly across remaining blocks. MR-LRCs can recover from all such failure patterns. Azure-LRC+1, Optimal Cauchy and Uniform Cauchy are all $> 99\%$ as durable as an MR-LRC in this scenario for our choice of coefficients. Azure-LRC is not shown because it does not have a local group covering its global parities, making it unsuitable for this comparison.

est. This is followed by Azure-LRC+1, whose local groups are larger than the other constructions (since we have constrained p). In fact for 24-of-28, Azure-LRC+1 has the same locality as Azure-LRC due to having only 1 local group that spans all data blocks. Optimal Cauchy LRC evenly divides the data blocks, but requires each local group to contain *all* the global parities (we need this to prove distance optimality as explained in Section 6.2), making its locality $\left\lceil \frac{k}{p} \right\rceil + g$. Uniform Cauchy LRC has the lowest locality $\left\lceil \frac{n}{p} \right\rceil$ since it uniformly divides n blocks into p groups.

Azure-LRC has the lowest average degraded read cost (*ADRC*), closely followed by Uniform Cauchy LRC. The *ADRC* is calculated only for the data blocks (Definition 5.10), and therefore is directly proportional to the size of local groups. Therefore, since Azure-LRC has the smallest local groups, it also has the smallest *ADRC*. Although, as the stripes become wider, the difference between the local group sizes of Azure-LRC and Uniform Cauchy LRC starts reducing. This reflects in the reduction of the ratio of the *ADRC* of Uniform Cauchy LRCs compared to Azure-LRCs. In particular, the *ADRC* of Uniform Cauchy LRC is about 8% more than Azure-LRC for 24-of-28, but is only about 5% more for 96-of-105. Azure-LRC+1 has the highest *ADRC* followed by Optimal Cauchy LRC, both owing to their larger local repair groups.

Average repair cost (ARC) is lowest for Uniform Cauchy LRC. Very similar to the *ADRC*, the ARC_1 and ARC_2 (Definition 5.11) are the degraded read cost for all n blocks of a stripe for 1 block failure and 2 block failures respectively. We choose to showcase both ARC_1 and ARC_2 because the wider the LRC, the higher the likelihood that more than one failure can occur in a stripe as explained in Section 3. Here, the disadvantage of Azure-LRC turns into the advantage for Uniform Cauchy LRC. Specifically, the degraded read cost for the global parities involves reading all data blocks in the case of Azure-LRC. For Uniform Cauchy LRC, the degraded read cost of global parities is the same as a data block, both of which are only slightly higher than the data block reconstruction cost of an Azure-LRC. The exception is 24-of-28 where Azure-LRC has a slightly lower ARC than Uniform Cauchy LRC, owing to these specific parameters, but this too becomes favorable for Uniform Cauchy LRCs in the case of ARC_2[3]. Across the schemes, as the number of data blocks increase, the ARC of Azure-LRC reduces, but at the same time, any additional global parity increases the *ARC* significantly. The difference between the Azure-LRC+1 and Optimal Cauchy LRC is lower than their difference in *ADRC*. This is due to Azure-LRC+1 having a very low-cost global parity reconstruction. Nevertheless, the Azure-LRC+1 and Optimal Cauchy LRCs have the highest, and second highest ARCs among the four constructions.

Uniform Cauchy LRCs have the best random failure tolerance. We compare the LRC constructions empirically by evaluating their durability against random failures (including when the number of failures are $\geq d$ when d is the distance of the code). For each code, we choose various values of i and conduct a Monte-Carlo experiment in which i blocks are removed uniformly at random (without replacement) from the n blocks of stripe. Then data recovery is attempted. The more times recovery can succeed, the more durable the code. For each random failures experiment at least 1 million recover-

[3] We refrain from detailing the calculation of ARC_2 because it significantly more complex than ARC_1.

ies from unique block failure combinations were attempted. The exception was 4 random failures for 24-of-28, where all combinations (20475) were exhaustively checked.

Figure 5 shows the results of the various failure scenarios on each of the four schemes across all four LRCs. Uniform Cauchy LRC outperforms all other LRC constructions in each scenario, for each scheme, except 48-of-55. In 48-of-55, the Optimal Cauchy LRC has the highest recoverability ratio. The intuition behind Uniform Cauchy LRC's superior performance is that when a high number of failures happen uniformly at random, it's more likely that at least one failure occurs in each local repair group. This results in all local parities contributing to the reconstruction process, leading to a higher success rate.

Simple choices of coefficients give *almost* MR-LRCs in our parameter regime. We construct all our generator matrices using Cauchy matrices. We then conduct an experiment to compare our codes to a *hypothetical MR-LRC* of the exact same design. We say 'hypothetical' because although there is proof that such an MR-LRC can exist [18, 22], currently there does not exist a deterministic way to construct such a code. Nevertheless, we can precisely characterize the erasure patterns that can be recovered by such MR-LRCs. Simply put, an MR-LRC with p local parities and r global parities can recover *any* pattern of $r+p$ failures, *as long as* there is at least one failure in each local repair group (so that each local parity may contribute towards the reconstruction). An LRC which has this property is an MR-LRC.

We conduct an experiment in which we *plant* one failure in each local group (i.e. p total planted failures), and then add another r failures at random. Then we attempt recovery. We find that at least with coefficients we chose (all derived from Cauchy matrices), *all* the codes we tested were very close to being MR-LRCs. So even if it is hard to construct MR-LRCs, in practical parameter settings, it is not hard to realize many of their benefits using common code constructions.

We do point out that even though all code constructions were close to being MR-LRCs, this does not mean that they were all equally durable against random erasures. This is because the shape of the code affects the probability that $r+p$ random failures will spread out so that each local repair group gets at least one failure. Indeed, this provides some intuition as to why Uniform Cauchy LRCs perform the best against random erasures. It is because the evenly sized local repair groups maximize the probability that each local repair group will see at least one failure (recall MR-LRC Definiton 5.9).

Mean-time-to-data-loss (MTTDL). MTTDL (for a stripe) has been a canonical metric for reliability in the coding community [14]. It is modeled using a continuous time Markov chain in which each state represents the number of erasures in a stripe. There is a final absorbing state, denoted *fail*, which represents a data loss/unavailability event. The MTTDL for a coding scheme is simply the mean time until a Markov chain starting at state zero reaches the absorbing state i.e. the expected time until a healthy stripe experiences a data unavailability event. Each state representing e failures, denoted s_e, may transition to the states s_{e-1}, s_{e+1} or *fail*. These transition probabilities are usually modelled using exponential distributions representing *repair time* (which captures the transition from s_e to s_{e-1}), *failure probability* (which captures the transition from s_e to s_{e+1}) and *complete failure probability* (which captures the transition from s_e to *fail*). These theoretical models are ubiquitous in modeling stripe reliability for MDS codes, but they have also been used to study the reliability of LRCs [27, 42, 44]. In our work, replace the *modelled* transition probabilities with *observed* ones, in order to get a better estimate of the MTTDL of our codes.

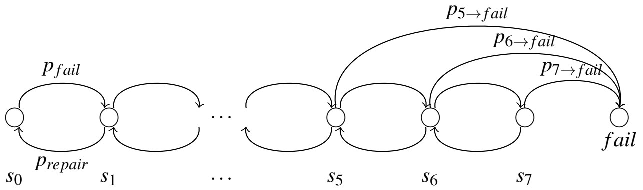


Figure 7: MTTDL Markov Chain for a 48-of-55 code. The probabilities $p_{5\to fail}$, $p_{6\to fail}$, and $p_{7\to fail}$ are added from Fig. 5 (b, c, d) where they are determined empirically. The probabilities $p_{i\to fail}$ for $i<5$ are not pictured because they are zero.

We accomplish this by supplementing our theoretical model with empirical data in the following way: for each state s_e, we add a transition to the irrecoverable data loss state with probability $p_{e\to fail}$, where $p_{e\to fail}$ is probability that a stripe with e randomly distributed failures cannot recover the data, as shown in Figure 7. To be clear, in order to model MTTDL this way, we need the explicit generator matrix of our code, and then to conduct the random failures experiment to generate data. While no model is perfect, we believe that this is a more realistic evaluation of MTTDL of an LRC, since we meaningfully account for the observed durability of the explicit code (i.e. accounting for coefficients).

Table 2 shows the MTTDL comparisons for different LRC constructions. Since we are interested in the relative comparison of MTTDLs, we show the ratio of each LRC construction with the Uniform Cauchy LRC MTTDL (where MTTDL is calculated as a function of p_{fail} and p_{repair} whose values can differ significantly based on specific cluster size, architecture, disk makes/models, etc.). Although p_{repair} can differ based on the architecture of the code, those differences are negligible in comparison to the wait-time for the detection of the missing blocks in a stripe as explained in Section 2. We find that Uniform Cauchy LRC achieves the highest MTTDL (up to a factor of 100× in our evaluation), while Azure-LRC+1 has the lowest MTTDL values. This is in line with our observations that Uniform Cauchy LRCs and Azure-LRC+1 have respectively the highest and lowest performance in reconstructing random erasures, and also the lowest and highest average locality. This is unsurprising since MTTDL of an LRC is inversely proportional to its average locality [44].

9 Maintenance-robust deployment

When deploying an erasure code in a large-scale storage cluster, there are several placement constraints that need to be met in order to ensure adequate data reliability. For example, usually no two blocks of the same stripe are put on the same disk, server, rack or at times even racks powered by the same power source to improve data reliability. Placement restrictions have been studied recently in [30], and in this work we comment that placement constraints can be considered during code design as well.

In a storage cluster, the smallest unit in which maintenance such as kernel/firmware/hardware upgrades can be performed is known as a *maintenance zone*. The design of maintenance zones affects reliability because all servers/devices in a single maintenance zone can be turned off simultaneously during a maintenance event. So we would like to have each stripe intersect a maintenance zone as little as possible, making it robust to the maintenance events in a real storage cluster. In an ideal world, we could make maintenance zones as small as possible (e.g. a single rack) to limit the impact of maintenance events on reliability. But this is not practically feasible, as it would increase the operational toil of maintenance tasks. One reason for this is: after every maintenance task (belonging to a single maintenance zone) cluster-wide (and therefore time-consuming and expensive) data reachability and reliability tests are usually conducted to ensure that the maintenance activity was completed without unforeseen events. If maintenance zones are too small (say an individual server), then a large-scale cluster of tens-of-thousands of servers would be in perennial testing activity. This severely limits the number of maintenance zones, and in practice, storage administrators have informed us that the total number of maintenance zones in a single cluster are typically restricted to below 20. This means that a maintenance event can affect a large number of stripes, and hence it is important to deploy stripes in a manner that is robust to maintenance events.

One desirable property of a code deployment might be that no maintenance event that affects a single zone should render any stripe unrecoverable. We name this *maintenance-robust deployment*. Since maintenance events are foreseeable, and usually performed one zone at a time, a maintenance-robust deployment would ensure high availability.

In the context of LRCs, we can further optimize deployment strategies to increase the likelihood of cost-efficient local repairs. For example, if we guarantee that every maintenance zone has at most one block from each local group, all degraded reads during a maintenance event would only require local repairs. We term such a deployment *maintenance-robust-efficient deployment*. Ideally we want all LRC deployments to be maintenance-robust-efficient deployments.

Consider the $(9,4,2)-$LRC from Figure 3. If the number of maintenance zones, say z is 3. We can assign each block of data from every local repair group to a different maintenance zone (i.e. a maintenance-robust-efficient deployment). Here, during *any* maintenance event, the degraded read cost for the data blocks stored in that maintenance zone is still $\ell = 2$. However, if $z = 2$, we cannot guarantee local repairs in all cases. As a best case, we could have $z_1 = d_1, d_3, c_1, l_g$ and put the rest of the data in z_2. Suppose z_2 undergoes maintenance, the data can be repaired by first repairing c_2 using c_1, l_g, and subsequently performing a global repair using d_1, d_3, c_1 and c_2 to reconstruct d_2 and d_4. Suppose z_1 undergoes maintenance, d_1 can be repaired using d_2, l_1, and d_3 can be repaired using d_4, l_2. Thus, with $z = 2$, a $(9,4,2)-$LRC can only be deployed in a maintenance-robust manner, where global repairs are required for reconstructing data. In general, when $z >= \ell + 1$ for an LRC, it can guarantee maintenance-robust-efficient deployment. The parameter $\ell + 1$ in Table 2 shows that minimum number of maintenance zones required for maintenance-robust-efficient deployment to be possible for the schemes we have evaluated in this work. Thus, with $z \approx 20$, all schemes except 96-of-105 can be deployed in a maintenance-robust-efficient manner for Uniform Cauchy LRCs.

10 Related Work

LRCs have widely studied and deployed in the last decade due to their practical improvements over MDS codes. While Microsoft Azure and Facebook first published the commercial use of LRCs [27, 42], much academic research has also been conducted to find LRC constructions with desirable properties [1, 6, 11, 17, 39, 39, 44, 46, 48]. One area of research has been explicit constructions of distance optimal codes (Definition 5.4) [6, 24, 28, 35, 37, 38, 47, 49, 51]. Tamo and Barg [47] were among the first to provide a general construction of distance-optimal LRCs (with small field sizes), but with some constraints on the allowable parameters. This construction was further generalized by Kolosov et al. [33] for a wide range of parameters. Recent years have seen a surge in research on MR-LRCs [4, 5, 12, 15, 16, 18–22, 36]. Recently, Gopi et al. [18] provided an explicit construction of MR-LRCs which is broad, but does not cover our parameter regime.

Wide codes are known to be deployed in two commercial settings, VAST [50] and Backblaze [3], and recent works have studied wide codes in academia. Haddock et al. analyze wide codes' performance using GPUs [23]. Li et al. show local erasures codes in the context of hard-disk drives [34]. More recently, Hu et al. [25] study the performance issues in repair, encoding, and update performance of wide codes. Our practically inspired work adds to this literature by studying the trade-offs of wide LRCs from construction to deployment.

11 Conclusion

In this work we show that many subtle factors can affect LRC reliability in real world scenarios: coefficients in the generator matrix, design of local repair groups, and maintenance-robust-deployment. We show the value of an experiment-driven understanding of reliability as it provides us novel insights into design choices which have a meaningful impact on reliability. Indeed, this culminates in our construction of Uniform Cauchy LRCs which outperform popular (and provably distance optimal) codes in practice.

12 Acknowledgements

We thank our shepherd Patrick P. C. Lee and the anonymous reviewers for their valuable feedback and suggestions. We extend special thanks to Larry Greenfield and numerous other researchers and engineers at Google.

A Optimal Cauchy LRCs

A.1 Distance optimality of Optimal Cauchy LRCs

Theorem A.1. *For $r > 0$, $O_{n,k,r,p}$ generates an error correcting code with distance exactly $r+2$.*

Proof. We will simply show that the distance of the code generated by $O_{n,k,r,p}$ is equal to the distance of the code generated by $G_{(k+r+1),k}$. Recall that $G_{(k+r+1),k}$ generates a code with distance exactly $r+2$, which is equivalent to saying that this code can correct *any* pattern of $r+1$ erasures, i.e. $G_{(k+r+1),k}$ has row rank at least k if any $r+1$ rows are deleted.

We will show that $O_{n,k,r,p}$ has row rank at least k whenever any $r+1$ rows are deleted. Note that whenever some $r+1$ rows are deleted from $O_{n,k,r,p}$, they may or may not contain any rows among the $\tilde{r}_1, \ldots, \tilde{r}_p$.

First we consider the case when at least one of $\tilde{r}_1, \ldots, \tilde{r}_p$ are deleted. In this case, we may consider *all* of them lost, and note that we have exactly the same rows remaining as if g_{r+k+1}, and at most r other rows were lost in $G_{(k+r+1),k}$. Since, $G_{(k+r+1),k}$ generates an MDS code with distance $r+2$, it is clear that the remainder of the rows will have rank k.

Now consider the case when the $r+1$ deleted rows do not contain any of the rows $\tilde{r}_1, \tilde{r}_2, \ldots, \tilde{r}_p$. In this case, we can compute $g_{k+r+1} = \tilde{r}_1 + \tilde{r}_2 + \cdots + \tilde{r}_p$, which reduces to the case when $r+1$ rows are deleted from $G_{(k+r+1),k}$. □

Lemma A.2. *Given $k, r, p > 0$, $rp^2 > k + rp$, and p even, the LRC formed by $O_{n,k,r,p}$ is distance optimal.*

Proof. Recall that for a LRC to be distance optimal we must have,

$$n = k + \left\lceil \frac{k}{\ell} \right\rceil + d - 2 \tag{1}$$

where ℓ is the locality parameter and d is the distance. We know that $\ell = \frac{k}{p} + r$ and $n = k + r + p$ for an (n, k, r, p)-optimal cauchy code. Substituting ℓ and $d = r+2$ (from Theorem A.1), equation 1 becomes,

$$n = k + \left\lceil \frac{k}{\frac{k}{p} + r} \right\rceil + r = k + \left\lceil \frac{pk}{k + pr} \right\rceil + r$$

So an (n, k, r, p)-optimal cauchy code will be distance optimal whenever $\left\lceil \frac{pk}{k+pr} \right\rceil = p$, i.e. $p - 1 < \frac{pk}{k+pr} \leq p$. Both inequalities hold as long as $r, p, k > 0$ and $rp^2 < k + rp$. □

A.2 Relaxing constraints

We have shown how to construct distance optimal LRCs with blocklength $n = k + r + p$, where k is the dimension of the code, r is the number of global parity checks, and p is the number of local parity checks as long as p is even, $r, p, k > 0$ and $rp^2 < k + rp$. We will briefly discuss some extensions of these parameter regimes in this section.

Claim A.3. *We may modify our construction of distance optimal codes to work with p odd.*

If $p = 1$, we may modify the construction so that $r_1 = r_p = \tilde{r}_p = g_{k+r+1}$, and so $G_{(k+r+1),k} = O_{n,k,r,p}$. This just reduces to the code defined by $G_{(k+r+1),k}$.

If $p \geq 3$, we may modify the construction so that $\tilde{r}_i = r_i$ for $1 \leq i \leq p-2$, $\tilde{r}_{p-1} = r_{p-1} + g_{k+1} + \cdots + g_{k+r}$, and $\tilde{r}_p = r_p + g_{k+1} + \cdots + g_{k+r}$. In this case,

$$\begin{aligned} \tilde{r}_1 + \tilde{r}_2 + \cdots + \tilde{r}_p &= 2(g_{k+1} + g_{k+2} + \cdots + g_{k+r}) + g_{k+r+1} \\ &= g_{k+r+1} \end{aligned}$$

From here, the same reasoning as the previous sections implies that these codes are distance optimal. Notice that when $p \geq 3$, these modifications do not change ℓ because $\tilde{r}_p$ is a local parity check on $\frac{k}{p} + r$ data blocks.

Claim A.4. *If $\frac{rp^2}{2} < k + rp < rp^2$, our construction gives codes whose blocklength is at most one greater than a distance optimal code.*

This is easily observed by following the proof of Lemma 6.2 and considering the cases where $p - 2 < \frac{pk}{k+pr} \leq p - 1$. Previous works have also found such 'off by one' LRCs in broader parameter regimes than truly optimal LRCs [47]. This allows us more freedom to explicitly construct almost distance optimal and wide LRCs.

Claim A.5. *We may construct codes with alphabet size $q \geq k + r + 1$.*

Note that we can construct our LRCs starting from the generator matrix of any MDS code in place of $G_{k+r+1,k}$. In particular, taking $G_{k+r+1,k}$ to be the generator matrix of any Reed-Solomon code, we only need $q \geq k + r + 1$ (since Reed-Solomon codes need $q \geq n$). For our settings q is fixed to be 256, so we have the flexibility to construct many wide explicit codes.

References

[1] Abhishek Agarwal, Alexander Barg, Sihuang Hu, Arya Mazumdar, and Itzhak Tamo. Combinatorial alphabet-dependent bounds for locally recoverable codes. *IEEE Transactions on Information Theory*, 2018.

[2] Backblaze. Disk Reliability Dataset. https://www.backblaze.com/b2/hard-drive-test-data.html, 2013-2018.

[3] Backblaze. Erasure coding used by Backblaze. https://www.backblaze.com/blog/reed-solomon/, 2013-2018.

[4] SB Balaji and P Vijay Kumar. On partial maximally-recoverable and maximally-recoverable codes. In *IEEE International Symposium on Information Theory (ISIT)*, 2015.

[5] Alexander Barg, Zitan Chen, and Itzhak Tamo. A construction of maximally recoverable codes. *Designs, Codes and Cryptography*, 2022.

[6] Alexander Barg, Itzhak Tamo, and Serge Vlăduţ. Locally recoverable codes on algebraic curves. *IEEE Transactions on Information Theory*, 2017.

[7] Shimrit Ben-Yair. Updating Google Photos' storage policy to build for the future. https://blog.google/products/photos/storage-changes/, 2020.

[8] Johannes Bloemer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. An xor-based erasure-resilient coding scheme. 1995.

[9] Eric Brewer. Spinning Disks and Their Cloudy Future. https://www.usenix.org/node/194391, 2018.

[10] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for data centers. Technical report, Google, 2016.

[11] Viveck Cadambe and Arya Mazumdar. An upper bound on the size of locally recoverable codes. In *2013 International Symposium on Network Coding (NetCod)*. IEEE, 2013.

[12] Han Cai, Ying Miao, Moshe Schwartz, and Xiaohu Tang. A construction of maximally recoverable codes with order-optimal field size. *IEEE Transactions on Information Theory*, 2021.

[13] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[14] Garth Alan Gibson. *Redundant disk arrays: Reliable, parallel secondary storage*. PhD thesis, University of California, Berkeley, 1991.

[15] Parikshit Gopalan, Guangda Hu, Swastik Kopparty, Shubhangi Saraf, Carol Wang, and Sergey Yekhanin. Maximally recoverable codes for grid-like topologies. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2017.

[16] Parikshit Gopalan, Cheng Huang, Bob Jenkins, and Sergey Yekhanin. Explicit maximally recoverable codes with locality. *IEEE Transactions on Information Theory*, 2014.

[17] Parikshit Gopalan, Cheng Huang, Huseyin Simitci, and Sergey Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information Theory*, 2012.

[18] Sivakanth Gopi and Venkatesan Guruswami. Improved maximally recoverable lrcs using skew polynomials. *IEEE Transactions on Information Theory*, 2022.

[19] Sivakanth Gopi, Venkatesan Guruswami, and Sergey Yekhanin. On maximally recoverable local reconstruction codes. In *Electron. Colloquium Comput. Complex.*, 2017.

[20] Sivakanth Gopi, Venkatesan Guruswami, and Sergey Yekhanin. Maximally recoverable lrcs: A field size lower bound and constructions for few heavy parities. *IEEE Transactions on Information Theory*, 2020.

[21] Matthias Grezet, Thomas Westerbäck, Ragnar Freij-Hollanti, and Camilla Hollanti. Uniform minors in maximally recoverable codes. *IEEE Communications Letters*, 2019.

[22] Venkatesan Guruswami, Satyanarayana V Lokam, and Sai Vikneshwar Mani Jayaraman. -msr codes: Contacting fewer code blocks for exact repair. *IEEE Transactions on Information Theory*, 2020.

[23] Walker Haddock, Purushotham V Bangalore, Matthew L Curry, and Anthony Skjellum. High performance erasure coding for very large stripe sizes. In *2019 Spring Simulation Conference (SpringSim)*.

[24] Kathryn Haymaker, Beth Malmskog, and Gretchen Matthews. Locally recoverable codes with availability $t <= 2$ from fiber products of curves. *arXiv preprint arXiv:1612.03841*, 2016.

[25] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick PC Lee, Weichun Wang, and Wei Chen. Exploiting combined locality for {Wide-Stripe} erasure coding in distributed storage. In *USENIX File and Storage Technologies (FAST)*, 2021.

[26] Cheng Huang, Minghua Chen, and Jin Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *ACM Transactions on Storage (TOS)*, 2013.

[27] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure Coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)*, 2012.

[28] Lingfei Jin. Explicit construction of optimal locally recoverable codes of distance 5 and 6 via binary constant weight codes. *IEEE Transactions on Information Theory*, 2019.

[29] Saurabh Kadekodi. *DISK-ADAPTIVE REDUNDANCY: tailoring data redundancy to disk-reliability heterogeneity in cluster storage systems*. PhD thesis, Carnegie Mellon University, 2020.

[30] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, KV Rashmi, and Gregory R Ganger. Tiger:{Disk-Adaptive} redundancy without placement restrictions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[31] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, KV Rashmi, and Gregory R Ganger. PACEMAKER: Avoiding heart attacks in storage clusters with disk-adaptive redundancy. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[32] Saurabh Kadekodi, K V Rashmi, and Gregory R Ganger. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. In *USENIX File and Storage Technologies (FAST)*, 2019.

[33] Oleg Kolosov, Gala Yadgar, Matan Liram, Itzhak Tamo, and Alexander Barg. On fault tolerance, locality, and optimality in locally repairable codes. *ACM Transactions on Storage (TOS)*, 2020.

[34] Yin Li, Hao Wang, Xuebin Zhang, Ning Zheng, Shafa Dahandeh, and Tong Zhang. Facilitating magnetic recording technology scaling for data center hard disk drives through {Filesystem-Level} transparent local erasure coding. In *USENIX File and Storage Technologies (FAST)*.

[35] Jian Liu, Sihem Mesnager, and Lusheng Chen. New constructions of optimal locally recoverable codes via good polynomials. *IEEE Transactions on Information Theory*, 2017.

[36] Shu Liu and Chaoping Xing. Maximally recoverable local reconstruction codes from subspace direct sum systems. *arXiv preprint arXiv:2111.03244*, 2021.

[37] Gaojun Luo and Xiwang Cao. Constructions of optimal binary locally recoverable codes via a general construction of linear codes. *IEEE Transactions on Communications*, 2021.

[38] Giacomo Micheli. Constructions of locally recoverable codes which are optimal. *IEEE Transactions on Information Theory*, 2019.

[39] D.S. Papailiopoulos and A.G. Dimakis. Locally repairable codes. In *IEEE International Symposium on Information Theory (ISIT)*, 2012.

[40] K V Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[41] David Reinsel-John Gantz-John Rydning, J Reinsel, and J Gantz. The digitization of the world from edge to core. *Framingham: International Data Corporation*, 16, 2018.

[42] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. In *International Conference on Very Large Data Bases (VLDB)*, 2013.

[43] Seagate. The Digitization of the World From Edge to Core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2018.

[44] Mostafa Shahabinejad. Locally repairable linear block codes for distributed storage systems. 2018.

[45] Natalia Silberstein, Ankit Singh Rawat, O Ozan Koyluoglu, and Sriram Vishwanath. Optimal locally repairable codes via rank-metric codes. In *IEEE International Symposium on Information Theory (ISIT)*, 2013.

[46] Itzhak Tamo and Alexander Barg. Bounds on locally recoverable codes with multiple recovering sets. In *IEEE Transactions on Information Theory*, 2014.

[47] Itzhak Tamo and Alexander Barg. A family of optimal locally recoverable codes. *IEEE Transactions on Information Theory*, 2014.

[48] Itzhak Tamo, Alexander Barg, and Alexey Frolov. Bounds on the parameters of locally recoverable codes. *IEEE Transactions on Information Theory*, 2016.

[49] Itzhak Tamo, Dimitris S Papailiopoulos, and Alexandros G Dimakis. Optimal locally repairable codes and connections to matroid theory. *IEEE Transactions on Information Theory*, 2016.

[50] VAST. Providing Resilience, Efficiently. https://www.usenix.org/node/194391, 2019.

[51] Guanghui Zhang. A new construction of optimal (r, δ) locally recoverable codes. *IEEE Communications Letters*, 2020.

ParaRC: Embracing Sub-Packetization for Repair Parallelization in MSR-Coded Storage

Xiaolu Li[†], Keyun Cheng[‡], Kaicheng Tang[‡], Patrick P. C. Lee[‡],
Yuchong Hu[†], Dan Feng[†], Jie Li[*], and Ting-Yi Wu[*]

[†]*Huazhong University of Science and Technology* [‡]*The Chinese University of Hong Kong*
[*]*Huawei Technologies Co., Ltd., Hong Kong*

Abstract

Minimum-storage regenerating (MSR) codes are provably optimal erasure codes that minimize the repair bandwidth (i.e., the amount of traffic being transferred during a repair operation), with the minimum storage redundancy, in distributed storage systems. However, the practical repair performance of MSR codes still has significant room to improve, as the mathematical structure of MSR codes makes their repair operations difficult to parallelize. We present ParaRC, a parallel repair framework for MSR codes. ParaRC exploits the sub-packetization nature of MSR codes to parallelize the repair of sub-blocks and balance the repair load (i.e., the amount of traffic sent or received by a node) across the available nodes. We show that there exists a trade-off between the repair bandwidth and the maximum repair load, and further propose a fast heuristic that approximately minimizes the maximum repair load with limited search time for large coding parameters. We prototype our heuristic in ParaRC and show that ParaRC reduces the degraded read and full-node recovery times over the conventional centralized repair approach in MSR codes by up to 59.3% and 39.2%, respectively.

1 Introduction

Erasure coding has been widely deployed in practical distributed storage systems for providing fault tolerance against the lost data in failed storage nodes, while incurring significantly lower redundancy overhead than traditional replication [37]. Among many erasure codes, Reed-Solomon (RS) codes are the most popular and reportedly deployed in production, such as Google [11], Facebook [23], Backblaze [9], and CERN [25]. However, RS codes are known to incur high *repair bandwidth* (i.e., the amount of traffic being transferred during a repair operation) when repairing a failed node, as the repair of any lost block needs to retrieve multiple coded blocks from other available nodes for decoding, thereby leading to bandwidth amplification.

Many repair-friendly erasure codes have been proposed in the literature to reduce the repair bandwidth of RS codes. Examples include regenerating codes [10, 24, 27, 33, 36], locally repairable codes [15, 17, 32], and piggybacking codes [29, 30]. In particular, minimum-storage regenerating (MSR) codes [10] are theoretically proven to be repair-optimal, such that they minimize the repair bandwidth for repairing a single node failure, while preserving the minimum storage redundancy as in RS codes (i.e., the redundancy is minimum among any erasure code that tolerates the same number of node failures). For example, compared with the (14,10) RS code adopted by Facebook [23, 29] (i.e., 10 original uncoded blocks are encoded into 14 RS-coded blocks), MSR codes with the same coding parameters can reduce the repair bandwidth by 67.5%. Given the theoretical guarantees of MSR codes, many follow-up efforts have proposed practical constructions for MSR codes and evaluated their performance in real-world distributed storage systems (e.g., [13, 24, 27, 36]); for example, Clay codes [36] are shown to minimize both repair bandwidth and I/Os (i.e., the amount of disk I/Os to local storage during a repair operation is the same as the minimum repair bandwidth), support general coding parameters, and be deployed and integrated in Ceph [3].

While MSR codes provably minimize the repair bandwidth, we argue that their practical repair performance remains bottlenecked by the node where the lost block is decoded, as the node needs to retrieve an amount of data from other available nodes more than the amount of lost data; in other words, bandwidth amplification still exists, albeit less severe than RS codes. To mitigate the repair bottleneck issue, recent studies [20, 22] have shown how to parallelize and load-balance the repair for RS codes across multiple available nodes, by decomposing the repair operation into partial repair sub-operations that are executed in different nodes in parallel and combining the partially repaired blocks into the final decoded block. Thus, it is natural to ask whether we can also decompose and parallelize the repair for MSR codes like RS codes. Unfortunately, the answer is negative: the repair for RS codes satisfies the additive associativity of linear combinations and the repair operation can be decomposed; in contrast, MSR codes have a different mathematical structure from RS codes, such that the repair of MSR codes needs to solve a system of linear combinations and cannot be directly decomposed (see §2 for details).

This motivates an alternative to parallelize the repair of MSR codes. Our insight is that MSR codes build on *sub-packetization*, in which a block is partitioned into sub-blocks and the repair of a lost block in MSR codes is to retrieve a subset of sub-blocks from other available nodes for decoding. The sub-blocks of a lost block can be represented as different linear combinations, and are finally decoded by solving the system of linear combinations. Based on sub-packetization, our idea is to distribute the repair of sub-blocks across different available nodes and later combine the repaired sub-blocks to reconstruct the lost block. An open question is how to distribute the repair of sub-blocks to balance the *repair load* (i.e., the amount of traffic sent or received by a node) across the available nodes.

We present ParaRC, a parallel repair framework for MSR codes that aims to balance the repair load across the available nodes and hence accelerate the repair operation. We make the following contributions:

- We observe that there exists a trade-off between the repair bandwidth and the maximum repair load. To formally analyze the trade-off, we model the repair operation of MSR codes as a directed acyclic graph (DAG) [19] and solve the repair parallelization problem as a DAG coloring problem. We identify an extreme point, the *min-max repair load (MLP) point*, which minimizes the maximum repair load with the smallest possible repair bandwidth.
- We show that finding the MLP is computationally expensive in general, and hence propose a fast heuristic that quickly identifies the approximate MLP point even for large coding parameters.
- We prototype ParaRC atop Hadoop 3.3.4 HDFS [4] and evaluate our prototype on Alibaba Cloud [1]. We show that ParaRC reduces the degraded read and full-node recovery times by up to 59.3% and 39.2%, respectively, compared with the centralized repair for Clay codes. We also show that ParaRC reduces the full-node recovery time of the default repair method in Hadoop-3.3.4 HDFS by 71.4%.

We release the source code of our ParaRC prototype at: **http://adslab.cse.cuhk.edu.hk/software/pararc**.

2 Background and Motivation

2.1 Basics of Erasure Coding

We review the basics of erasure coding. We consider a large-scale distributed storage system that organizes data and performs reads/writes in fixed-size *blocks*, such that the block size is large enough (e.g., 128 MiB in Hadoop 3.3.4 HDFS [4] and 256 MiB in Facebook [28]) to mitigate I/O overhead. In this work, we target the distributed storage environments where the network bandwidth and disk I/Os are the bottlenecks, as opposed to the computational overhead for encoding and decoding operations in erasure coding.

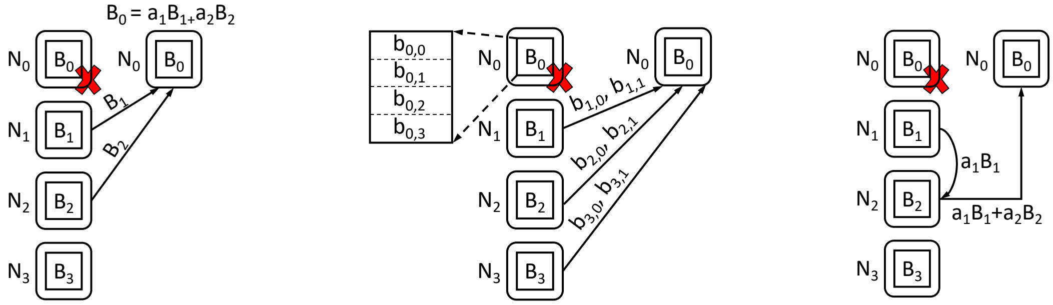
There are many approaches to construct erasure codes, among which Reed-Solomon (RS) codes [31] are the most widely deployed (e.g., [9, 11, 23, 25]). Specifically, an (n,k) RS code, configured by two parameters k and n (where $n > k$), encodes every set of k original uncoded blocks into n coded blocks, such that any k out of n coded blocks suffice to decode the k original uncoded blocks. Each set of n coded blocks is called a *stripe*. In this work, we focus on a single stripe, while multiple stripes are independently and identically encoded. Each stripe is stored in n distinct nodes, so as to tolerate any $n-k$ node (or block) failures. RS codes satisfy three practical properties: (i) *generality*, where n and k can be general parameters (provided that $n > k$), (ii) *maximum distance separable (MDS)*, where the redundancy overhead n/k is the minimum for tolerating any $n-k$ node failures, and (iii) *systematic*, where the k uncoded blocks are kept in the n coded blocks (i.e., the uncoded blocks remain directly accessible after encoding).

We elaborate on the mathematical properties of RS codes to help motivate our work. In this paper, we treat the uncoded and coded blocks equivalently in a systematic stripe and simply refer to them as "blocks" in our discussion. Let $B_0, B_1, \cdots, B_{n-1}$ be the n blocks of a stripe in an (n,k) RS code that are respectively stored in n nodes, denoted by $N_0, N_1, \cdots, N_{n-1}$. Each block can be expressed as a linear combination of k blocks of the same stripe under Galois Field arithmetic. For example, we have $B_0 = \sum_{i=1}^{k} a_i B_i$ for some coding coefficients a_i's ($1 \le i \le k$).

Despite the popularity, RS codes are known to incur high repair penalty, since repairing a single lost block in RS codes needs to transfer multiple blocks of the same stripe from other available nodes. The repair penalty manifests in two aspects. First, the repair incurs high *repair bandwidth*, defined as the amount of traffic transferred over the network during a repair operation. In general, an (n,k) RS code incurs a repair bandwidth of k times the block size when repairing a lost block. Figure 1(a) shows an example of the conventional centralized repair for the $(4,2)$ RS code. To repair a lost block (say B_0), the new node (say N_0) downloads *any* $k = 2$ blocks (say B_1 and B_2 from N_1 and N_2, respectively), so as to repair B_0 via the linear combination of the downloaded blocks. The repair bandwidth is 512 MiB for a block size of 256 MiB.

Second, the conventional centralized repair also incurs high *maximum repair load*, where the repair load of a node is defined as the amount of traffic that the node sends or receives, whichever is larger, during a repair operation, and the maximum repair load is the largest repair load among all nodes. In the centralized repair, the new node has the most traffic among all nodes, as it receives an amount of traffic that is k times the block size, while each other available node sends one block only. Thus, the performance of the centralized repair is bottlenecked by the new node. For example, from Figure 1(a), the new node N_0 has the most received traffic, and the maximum repair load is also 512 MiB for a block size of 256 MiB.

Thus, the repair performance in RS codes is dominated by both the repair bandwidth and the maximum repair load. We argue that while many studies focus on reducing the repair bandwidth (§2.2) or reducing the maximum repair load (§2.3), there exists a trade-off in minimizing both of the performance metrics (§2.4).



(a) Centralized repair for RS codes (b) Centralized repair for Clay codes (c) Repair pipelining for RS codes

Figure 1: Repair examples: (a) conventional repair for the (4,2) RS code; (b) centralized repair for the (4,2) Clay code (which minimizes the repair bandwidth); and (c) repair pipelining for the (4,2) RS code (which minimizes the maximum repair load).

2.2 Reducing Repair Bandwidth

Existing studies on erasure coding reduce the repair bandwidth by proposing new erasure code constructions. Examples include regenerating codes [10, 13, 24, 27, 33, 35, 36], locally repairable codes [15, 32], and piggybacking codes [29, 30]. In this paper, we focus on *minimum-storage regenerating (MSR)* codes (first proposed in [10]), which theoretically minimize the repair bandwidth for repairing a single lost block, with the minimum redundancy (i.e., MDS property) as RS codes.

MSR codes differ from RS codes by performing *subpacketization*, which divides a block into multiple sub-blocks and performs encoding and repair at the sub-block granularity. Specifically, an (n,k) MSR code partitions each block B_i $(0 \leq i \leq n-1)$ into w sub-blocks $(w > 1)$, denoted by $b_{i,0}$, $b_{i,1}, \cdots, b_{i,w-1}$, such that each sub-block is encoded through a linear combination of $k \times w$ sub-blocks from k blocks (under Galois Field arithmetic). To repair any lost block (or w sub-blocks therein), MSR codes only transfer sub-blocks from the other nodes, such that the total amount of traffic of the transferred sub-blocks is minimized.

Classical MSR codes [10] require that the available nodes read all their locally stored sub-blocks, encode them, and transfer the encoded sub-blocks to the new node (with the minimum repair bandwidth) to repair the lost block. In this work, we consider two state-of-the-art MSR codes, namely Clay codes [36] and Butterfly codes [24], both of which have been implemented and empirically evaluated. Our goal is to show the applicability of our work to different MSR codes, using Clay codes and Butterfly codes as two representatives. In particular, Clay codes minimize both repair bandwidth and I/Os (a.k.a. repair-by-transfer [33])) for general coding parameters n and k, while Butterfly codes minimize both repair bandwidth and I/Os for the k uncoded blocks in a systematic stripe and support $n-k=2$ only. Thus, we use Clay codes as our major baseline throughout the paper.

We first provide an overview for Clay codes. At a high level, Clay codes repair a lost block in three steps: (i) *pairwise reverse transformation (PRT)*, which couples sub-blocks in pairs and generates intermediate sub-blocks; (ii) *MDS decoding*, which performs linear combinations on k sub-blocks to decode intermediate sub-blocks and a subset of repaired sub-blocks; and (iii) *pairwise forward transformation (PFT)*, which again couples sub-blocks in pairs to generate the remaining repaired sub-blocks, such that the lost block is completely repaired. In Clay codes, the number of sub-blocks w is given by $w = (n-k)^{\lceil n/(n-k) \rceil}$.

Let us take the (4,2) Clay code (where $w = 4$) as an example, as shown in Figure 1(b). Let c_i be the i^{th} intermediate sub-block generated in the repair. Also, let $\langle \ldots \rangle_i$ denote some linear combination of sub-blocks within the brackets, where the subscript i differentiates the linear combinations with different coding coefficients. To repair a lost block, say B_0, the new node N_0 downloads two sub-blocks $b_{i,0}$ and $b_{i,1}$ from each N_i, where $1 \leq i \leq 3$. N_0 repairs the four sub-blocks of B_0 as follows. First, in the PRT step, N_0 generates two intermediate sub-blocks c_0 and c_1 by coupling $b_{2,1}$ and $b_{3,0}$:

$$c_0 = \langle b_{2,1}, b_{3,0} \rangle_0, \quad c_1 = \langle b_{2,1}, b_{3,0} \rangle_1. \tag{1}$$

Second, in the MDS decoding step, N_0 performs linear combinations on $b_{2,0}$ and c_0, and on $b_{3,1}$ and c_1. It repairs $b_{0,0}$ and $b_{0,1}$, and generates two intermediate sub-blocks c_2 and c_3:

$$\begin{aligned} b_{0,0} &= \langle b_{2,0}, c_0 \rangle_2, \quad c_2 = \langle b_{2,0}, c_0 \rangle_3, \\ b_{0,1} &= \langle b_{3,1}, c_1 \rangle_4, \quad c_3 = \langle b_{3,1}, c_1 \rangle_5. \end{aligned} \tag{2}$$

Finally, in the PFT step, N_0 repairs $b_{0,2}$ by coupling $b_{1,0}$ and c_2, and repairs $b_{0,3}$ by coupling $b_{1,1}$ and c_3:

$$b_{0,2} = \langle b_{1,0}, c_2 \rangle_6, \quad b_{0,3} = \langle b_{1,1}, c_3 \rangle_7. \tag{3}$$

The (4,2) Clay code minimizes the repair bandwidth to 384 MiB for a block size of 256 MiB (it downloads six sub-blocks of size 64 MiB each). Compared with the (4,2) RS code, the (4,2) Clay code reduces the repair bandwidth by 25%. Note that the maximum repair load of the Clay code is also 384 MiB (same as the repair bandwidth), which is the amount of traffic downloaded in the new node.

We also consider Butterfly codes [24] in this paper. For an (n,k) Butterfly code $(n-k=2)$, we focus on the repair of the first k original uncoded blocks in a systematic stripe (whose repair bandwidth and I/Os are both minimized). An (n,k)

Butterfly code divides each block into $w = 2^{k-1}$ sub-blocks. When repairing a lost block, a new node first downloads half of the sub-blocks from each available node. It then selects different subsets of sub-blocks among all the received sub-blocks and performs XOR operations to repair the w sub-blocks of the lost block. For example, to repair a lost block of size 256 MiB for the $(4,2)$ Butterfly code, the new node downloads 128 MiB of sub-blocks from each of the three available nodes, such that the repair bandwidth and the maximum repair load are both 384 MiB.

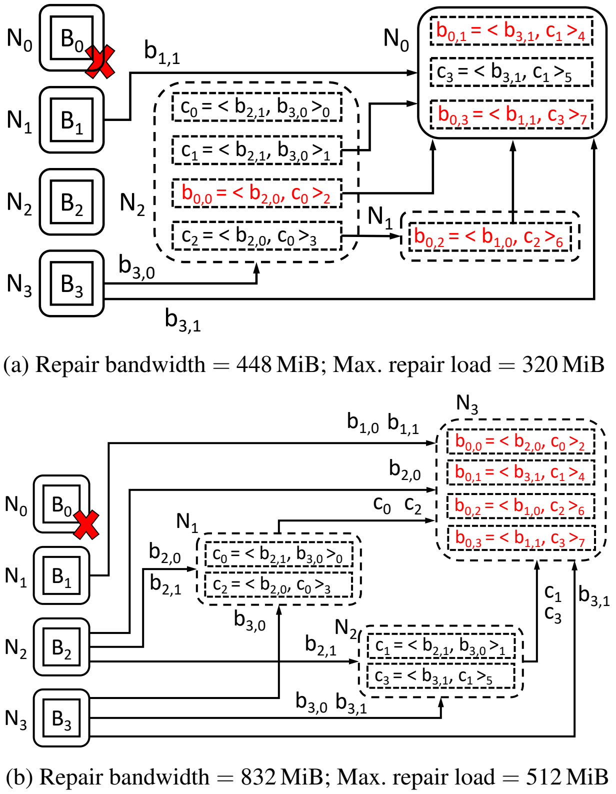
2.3 Reducing Maximum Repair Load

Some studies reduce the maximum repair load by decomposing and parallelizing a repair operation across the available nodes [20, 22]. In this work, we focus on *repair pipelining* [20], which reduces the time of repairing a lost block to almost the same as the time of directly reading a block.

Repair pipelining is mainly designed for RS codes [31]. It divides a single-block repair operation into multiple sub-block repair operations and evenly distributes sub-block repair operations across all nodes. For example, suppose that we use repair pipelining to repair a lost block B_0 for an (n,k) RS code. It first divides each block B_i $(0 \le i \le n-1)$ into multiple sub-blocks, denoted by $b_{i,0}, b_{i,1}, \cdots$. Recall that each block can be expressed as a linear combination of k blocks (§2.1), say $B_0 = \sum_{i=1}^{k} a_i B_i$ for some coding coefficients a_i's. Repair pipelining makes two observations. First, each sub-block in B_0 is also a linear combination of the k sub-blocks at the same block offset with the same coding coefficients, i.e., $b_{0,j} = \sum_{i=1}^{k} a_i b_{i,j}$, for the j-th sub-block. Second, the linear combination is addition associative, meaning that $b_{0,j}$ can be computed from the linear combinations of partial terms.

To repair B_0, repair pipelining works as follows. First, N_1 starts the repair of $b_{0,0}$ by sending $a_1 b_{1,0}$ from its local storage to N_2. Second, N_2 combines the received $a_1 b_{1,0}$ with $a_2 b_{2,0}$ from its local storage to form $a_1 b_{1,0} + a_2 b_{2,0}$. Third, N_2 sends $a_1 b_{1,0} + a_2 b_{2,0}$ to N_3; meanwhile, N_1 can start the repair of $b_{0,1}$ by sending $a_1 b_{1,1}$ from its local storage to N_2 without interfering with N_2's transmission. Finally, the last available node N_k reconstructs $b_{0,j}$ for each j-th sub-block and sends $b_{0,j}$ to N_0.

Repair pipelining reduces the maximum repair load to the same as the block size. For example, Figure 1(c) shows an example of repair pipelining for the $(4,2)$ RS code. The maximum repair load is 256 MiB for a block size of 256 MiB since each of the k available nodes sends or receives one block of data; it is even less than that in Clay codes (Figure 1(b)). Note that the repair bandwidth remains 512 MiB, the same as in the conventional repair for RS codes (Figure 1(a)), since k available nodes transfer k blocks of data in total.



(a) Repair bandwidth = 448 MiB; Max. repair load = 320 MiB

(b) Repair bandwidth = 832 MiB; Max. repair load = 512 MiB

Figure 2: Examples of the parallel repair for the $(4,2)$ Clay code. The example in figure (a), with more careful repair scheduling, has both less repair bandwidth and less maximum repair load than the example in figure (b).

2.4 Motivation and Challenges

From §2.3, a natural question to ask is whether we can apply repair pipelining to MSR codes (§2.2) to reduce the maximum repair load. Unfortunately, the answer is negative, mainly because the repair of sub-blocks is not based on the addition associativity as in RS codes; instead, it is done by solving a system of linear combinations (e.g., see Equations (1)-(3) in §2.2 for Clay codes). Thus, we cannot pipeline the repair of individual sub-blocks of MSR codes as in RS codes.

Nevertheless, the sub-packetization nature of MSR codes offers an opportunity for parallelizing a repair operation to reduce the maximum repair load. First, the repair of a sub-block in MSR codes only requires a subset of available sub-blocks; for example, in the (4,2) Clay code, each sub-block is a linear combination of two currently stored or intermediate sub-blocks. Thus, we can distribute the repair operations of sub-blocks across different nodes for load balancing. Second, in erasure coding implementation, each block is further divided into smaller-sized units (called *packets*), so that the repair of a block can be parallelized at the packet level (see §6 for implementation details).

Figure 2(a) shows a parallel repair example for the (4,2) Clay code. First, in the PRT step, N_2 generates c_0 and c_1 from $b_{3,0}$ (retrieved from N_3) and $b_{2,1}$ (locally stored in N_2). Second, in the MDS decoding step, N_2 decodes c_2 and $b_{0,0}$ from $b_{2,0}$ (locally stored in N_2) and c_0 (generated in the PRT step), while N_0 generates c_3 and $b_{0,1}$ from c_1 (retrieved from

	Repair bandwidth (MiB)	Maximum repair load (MiB)
RS; centralized	512 (highest)	512 (highest)
Clay; centralized	384 (lowest)	384 (high)
RS; parallel	512 (highest)	256 (lowest)
Clay; parallel	448 (medium)	320 (medium)

Table 1: Summary of the four repair methods for $(n,k) = (4,2)$.

N_2) and $b_{3,1}$ (retrieved from N_3). Finally, in the PFT step, N_1 repairs $b_{0,2}$ from $b_{1,0}$ (locally stored in N_1) and c_2 (retrieved from N_2), while N_0 repairs $b_{0,3}$ from $b_{1,1}$ (retrieved from N_1) and c_3 (generated in the MDS decoding step). Also, N_0 retrieves the repaired $b_{0,0}$ and $b_{0,2}$ from N_2 and N_1, respectively. In this example, the repair operation can be parallelized in two aspects: (i) the repair of $b_{0,1}$ and $b_{0,3}$ in N_0, as well as the repair of $b_{0,2}$ in N_1, can be performed in parallel; and (ii) the sub-block repair operations in N_0, N_1, and N_2 can be parallelized at the packet level. Thus, the maximum repair load is 320 MiB (i.e., the five sub-blocks $b_{0,0}$, $b_{0,2}$, $b_{1,1}$, $b_{3,1}$, and c_1 retrieved by N_0) for a block size of 256 MiB.

Such a parallel repair approach may amplify the repair bandwidth, as some sub-blocks are reused more than once by different nodes. For example, the sub-blocks $b_{2,1}$ and $b_{3,0}$ are used to compute c_1, c_2, and $b_{0,0}$. Each of the three sub-blocks will be transmitted over the network. Thus, instead of transmitting each of the sub-blocks $b_{2,1}$ and $b_{3,0}$ only once as in the centralized repair (Figure 1(b)), the parallel repair now includes the sub-blocks $b_{2,1}$ and $b_{3,0}$ in three transmissions. The repair bandwidth increases from the minimum point of 384 MiB to 448 MiB.

How to carefully schedule the parallel repair of different sub-blocks is a critical issue. Figure 2(b) shows another example of the parallel repair of the $(4,2)$ Clay code, where the repair is less efficiently scheduled. In this example, the sub-blocks $b_{2,0}$, $b_{2,1}$, and $b_{3,0}$ are all transmitted twice. Thus, the repair bandwidth is 832 MiB, while the maximum repair load is 512 MiB.

In summary, the parallel repair of MSR codes can be scheduled to balance the trade-off between the repair bandwidth and the maximum repair load, as shown in Table 1 for the $(4,2)$ Clay code. Our goal in this paper is to design a parallel repair solution that can effectively balance the trade-off for general coding parameters of MSR codes.

3 Model and Analysis

Before we design the parallel repair solution for MSR codes, we first formulate a generic repair model that characterizes the trade-offs between the repair bandwidth and the maximum repair load for different repair solutions, either centralized (e.g., Figures 1(a) and 1(b)) or parallel (e.g., Figures 1(c) and 2). In this section, we design our repair model (§3.1) and evaluate the repair bandwidth and the maximum repair load of a repair solution (§3.2). Finally, we analyze the trade-off between the repair bandwidth and the maximum repair load for different repair solutions on RS and MSR codes (§3.3).

3.1 Characterizing Repair Solutions

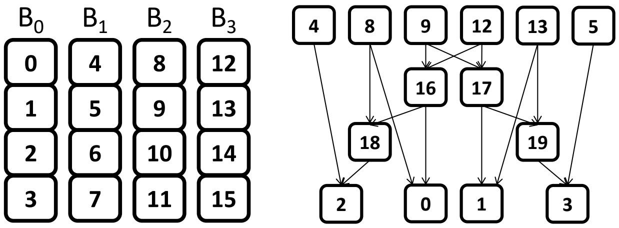
Design requirements. We first identify three design requirements for our repair model to characterize repair solutions based on our example in Figure 2:

- R1: It can describe the linear combination relationships of sub-blocks (e.g., $b_{0,0}$ is the linear combination of $b_{2,0}$, $b_{2,1}$, and $b_{3,0}$).
- R2: It can describe which node is scheduled to execute a repair operation for each sub-block and how the repair operation is executed (e.g., N_2 downloads $b_{3,0}$ from N_3 and generates $b_{0,0}$ with its locally stored $b_{2,0}$ and $b_{2,1}$).
- R3: It can describe how the repaired sub-blocks are collected (e.g., $b_{0,0}$, $b_{0,1}$, $b_{0,2}$, and $b_{0,3}$ can be repaired in different nodes, but are finally collected by N_0 for reconstructing block B_0).

Our repair model builds on the ECDAG abstraction [19], which characterizes and schedules erasure coding operations in distributed storage systems. Note that an ECDAG can model the linear combination relationships of sub-blocks (i.e., R1 addressed), but cannot directly schedule the repair operations for different sub-blocks in different nodes (i.e., R2 and R3 not addressed). In the following, we first introduce the ECDAG abstraction, and then explain how it can be extended to address all our requirements.

Basics of an ECDAG. We provide an overview of an ECDAG. An ECDAG $G = (V,E)$ is a directed acyclic graph (DAG) that describes an erasure coding operation (including the repair of a block), where V is the set of vertices and E is the set of edges. A vertex $v_\ell \in V$ (where $\ell \geq 0$) refers to either a sub-block that is stored in a node (i.e., $\ell = i \times w + j$ for $b_{i,j}$, where $i, j \geq 0$) or an intermediate sub-block that is generated on-the-fly but will not be finally stored (i.e., $\ell \geq n \times w$). With a slight abuse of notation, we refer to a sub-block with its vertex v_ℓ, where ℓ is the index. An edge $e(\ell_1, \ell_2) \in E$ means that the sub-block v_{ℓ_1} is an input to the linear combination for computing the sub-block v_{ℓ_2}. Note that the repair workflows vary across blocks, so the repair of each block will lead to a different ECDAG instance.

We use Clay codes [36] as an example to show how an ECDAG describes its repair workflow. Figure 3(a) shows the block layout of the $(4,2)$ Clay code (where $w = 4$) in an ECDAG, and Figure 3(b) shows the repair flow for block B_0, which we introduce in §2.2. First, in the PRT step, we couple sub-blocks v_9 ($b_{2,1}$) and v_{12} ($b_{3,0}$) as a pair and perform linear combinations to generate two intermediate sub-blocks v_{16} (c_0) and v_{17} (c_1). Second, in the MDS decoding step, we decode sub-blocks v_0 ($b_{0,0}$) and v_{18} (c_2) from sub-blocks v_8 ($b_{2,0}$) and v_{16} (c_0), and we decode sub-blocks v_1 ($b_{0,1}$) and v_{19} (c_3) from sub-blocks v_{13} ($b_{3,1}$) and v_{17} (c_1). Note that the sub-blocks v_0 ($b_{0,0}$) and v_1 ($b_{0,1}$) of B_0 are repaired. Finally, in the PFT



(a) Layout (b) ECDAG

Figure 3: An ECDAG example of repairing B_0 using the $(4,2)$ Clay code $(w=4)$.

step, we couple sub-blocks v_4 ($b_{1,0}$) and v_{18} (c_2) to repair sub-block v_2 ($b_{0,2}$), and also couple sub-blocks v_5 ($b_{1,1}$) and v_{19} (c_3) to repair sub-block v_3 ($b_{0,3}$). B_0 is now fully repaired.

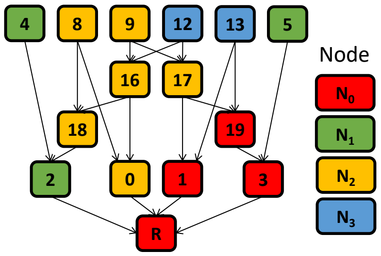
pECDAG. We extend the ECDAG abstraction into the pECDAG abstraction to support the scheduling of parallel sub-block repair operations, so that we can model the trade-off between the repair bandwidth and the maximum repair load. Specifically, a pECDAG makes two extensions over an ECDAG. First, it associates each vertex with a *color* that corresponds to a node, such that the node is responsible for generating or storing all sub-blocks associated with the same-colored vertices (i.e., R2 addressed). Second, it connects all repaired sub-blocks, which may reside in different nodes to a vertex R, which represents a data collector (i.e., R3 addressed). Figure 4(a) shows the pECDAG for the parallel repair in Figure 2. To help our discussion, we refer to the topmost vertices (e.g., v_4, v_5, v_8, v_9, v_{12}, and v_{13}) that correspond to the sub-blocks retrieved from the other available nodes as the *leaf vertices*, and the vertex R that corresponds to the data collector as the *root vertex*. Note that the colors of both the leaf vertices and root vertex are fixed, as they depend on where the retrieved sub-blocks and repaired block reside, respectively.

For example, from Figure 4(a), N_2 (i.e., yellow-colored) computes the sub-block v_{17} (c_1) in Figure 2 and sends it to N_0 (i.e., red-colored), which repairs the sub-blocks v_1 ($b_{0,1}$) and v_3 (i.e., $b_{0,3}$). Also, N_2 computes the sub-blocks v_0 ($b_{0,0}$) and v_{18} (c_2). It sends v_{18} to N_1 (i.e., green-colored), which repairs the sub-block v_2 ($b_{0,2}$). Finally, N_0 collects all the repaired sub-blocks for the reconstruction of B_0.

3.2 Evaluating Repair Solutions

Given (n,k,w) and the block to repair, there are different ways to color the vertices of a pECDAG, so there are multiple possible pECDAG instances. We associate each pECDAG instance with a *traffic table*, so as to efficiently quantify the repair bandwidth and the maximum repair load of the corresponding repair solution.

Definition of a traffic table. A traffic table maintains the amount of data that each node sends or receives when repairing a block. For each node in the system, the traffic table records the number of incoming sub-blocks received by the node and the number of outgoing sub-blocks sent by the node. The repair bandwidth is the total number of incoming sub-blocks (or equivalently, the total number of outgoing sub-blocks) of all nodes, while the maximum repair load is the largest number of incoming or outgoing sub-blocks of a node across all nodes. For example, Figure 4(b) shows the traffic table for the parallel repair solution shown in Figure 2, in which the repair bandwidth is 7 sub-blocks and the maximum repair load is 5 sub-blocks.



(a) pECDAG

	In	Out
N_0	5	0
N_1	1	2
N_2	1	3
N_3	0	2

(b) Load table

Figure 4: A pECDAG example of $(4,2)$ Clay code with $w=4$ to repair B_0.

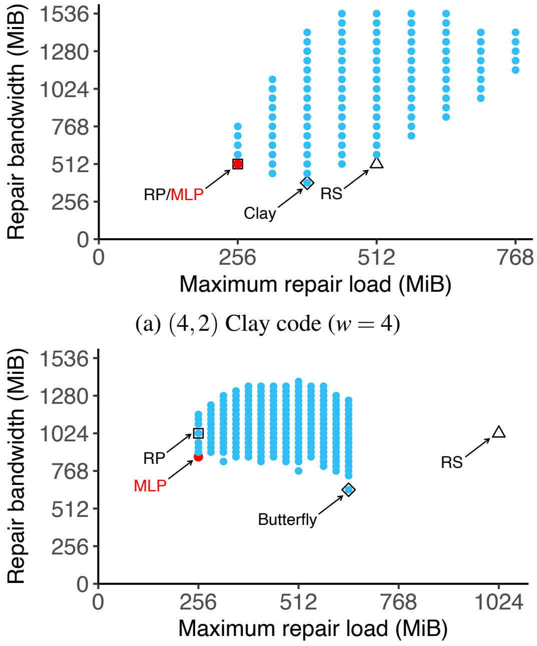
Construction of a traffic table. We show how we generate the traffic table for a given pECDAG instance. We initialize a traffic table T with two arrays $T.In$ and $T.Out$, which record the numbers of incoming and outgoing sub-blocks for each node, respectively. For each vertex v_i, we traverse each edge $e(v_i, v_j)$. Let N' and N'' be two nodes with respect to the colors of v_i and v_j, respectively. If v_i and v_j have different colors, we increment $T.Out[N']$ and $T.In[N'']$ by one; however, if there exist two edges, say $e(v_i, v_j)$ and $e(v_i, v_h)$, such that v_j and v_h have the same color that is different from v_i's color, we only increment T once for the corresponding pairs of nodes. The rationale is that the sub-block v_i only needs to be transmitted once to calculate the sub-blocks v_j and v_h.

For example, in Figure 4(a), both v_{18} and v_0 have the same color as v_8, we do not need to update the traffic table. For v_{17}, as v_1 has a different color, we count $e(v_{17}, v_1)$ as a transmission and increment the traffic table. As v_{19} and v_1 have the same color, we do not need to increment the traffic table for $e(v_{17}, v_{19})$.

3.3 Trade-off Analysis

Based on a pECDAG and its traffic table, we study the trade-off between the repair bandwidth and the maximum repair load. Our idea is to enumerate all possible color combinations of a pECDAG and find the corresponding traffic table for each color combination. Note that the colors of the leaf vertices and the root vertex are fixed (§3.1). Thus, for a pECDAG, we only need to enumerate the color combinations for the intermediate sub-blocks and repaired sub-blocks. Currently, we assume that the repair operation of a stripe is scheduled among the nodes (i.e., n nodes for an (n,k) code) that store the blocks of the stripe, so as to limit the interference across different stripes.

We consider the repair scenarios of two MSR codes: the



(a) $(4,2)$ Clay code ($w = 4$)

(b) $(6,4)$ Butterfly code ($w = 8$)

Figure 5: Trade-off analysis between the repair bandwidth and the maximum repair load.

$(4,2)$ Clay code and the $(6,4)$ Butterfly code. We consider the repair of block B_0 (i.e., the first block of a stripe) and construct a pECDAG for each of them. We apply a brute-force search to enumerate all color combinations; for each color combination, we generate the traffic table and obtain the corresponding repair bandwidth and maximum repair load. We show the spectrum of repair bandwidth and maximum repair load for different color combinations under the $(4,2)$ Clay code (Figure 5(a)) and the $(6,4)$ Clay code (Figure 5(b)). In the figures, we highlight the points corresponding to the centralized repair for RS codes (RS), repair pipelining for RS codes (RP), and the centralized repair for Clay codes (Clay) or Butterfly codes (Butterfly) for comparisons.

We find that different color combinations for an MSR code have different trade-offs between the repair bandwidth and the maximum repair load. We define a *min-max repair load point (MLP)*, which minimizes the maximum repair load, and whose repair bandwidth is minimized given this optimal maximum repair load. Note that the MLP does not guarantee the absolute minimum value of repair bandwidth. For example, for the $(4,2)$ Clay code, the MLP happens to be overlapped with the point of RP; for the $(6,4)$ Butterfly code, the MLP reduces the repair bandwidth by 15.6% compared with that of RP, while it achieves the same maximum repair load as RP.

This observation indicates that the parallel repair of an MSR code may further improve the repair performance of a distributed storage system if we can find the MLP. However, it is non-trivial to find the MLP in general. While the brute-force approach can always find the MLP, it also has high complexity. For a pECDAG of an (n,k) MSR code with w sub-blocks in a block, the lower bound of the number of vertices being colored is w (i.e., when there is no intermediate sub-block, we only need to color the w repaired sub-blocks). In this case, the lower bound of the total number of color combinations is n^w. For Clay codes, the lower bound is $n^{(n-k)^{\lceil n/(n-k)\rceil}}$, while for Butterfly codes, the lower bound is $n^{2^{k-1}}$. For example, for the (14,10) Clay code, the number of color combinations is no less than 14^{256}, while for the (12,10) Butterfly code, the number of combinations is no less than 12^{512}, which are not solvable in polynomial time. Thus, for reasonably large (n,k), it is important to reduce the size of the search space, and hence the running time.

4 Heuristic

As the brute-force approach in general is time-consuming to find the MLP, we propose a heuristic to find an approximate point that is close to the MLP. Our goal is to find a parallel repair solution represented in a pECDAG that keeps both the repair bandwidth and the maximum repair load as low as possible.

Design idea. The high-level idea is to search all the color combinations for a pECDAG, while pruning some branches based on the heuristic to reduce the search space. Intuitively, we can view our heuristic as searching for the solution based on Pareto optimality, such that it searches for the MLP on the Pareto frontier and prunes the dominated solutions that have both larger repair bandwidth and larger maximum repair load than a candidate solution.

We first introduce the key definitions. If two pECDAGs, say X and Y, have the same DAG structure except in the color of a single vertex that refers to an intermediate sub-block or a repaired sub-block, we call X and Y the *neighbors*. We perform the search on a pECDAG by examining all the neighbors of the pECDAG. If we have examined all the neighbors of a pECDAG, we say that the pECDAG is *searched*; otherwise, the pECDAG is *un-searched*. Our heuristic is composed of the following three steps.

Step 1: Initialization. We define an *un-searched pool*, which is used to keep the pECDAGs that will be searched, as well as a *candidate pool*, which is used to record the candidate pECDAG solutions to be returned. At the beginning, we generate a random pECDAG, in which the color of a vertex that refers to an intermediate sub-block or a repaired sub-block is randomly selected from a set of candidate colors that represent the nodes storing the available blocks and the node storing the repaired block. We add the random pECDAG to the un-searched pool and the candidate pool for initialization.

Step 2: Searching. Each time we retrieve a pECDAG from the un-searched pool. We enumerate all the neighbors of this pECDAG by changing the color of only one vertex (which refers to an intermediate sub-block or a repaired sub-block). If there are α such vertices and β candidate colors, a pECDAG has $\alpha \times (\beta - 1)$ neighbors (note that for each vertex, there are $\beta - 1$ different candidate colors to which we can change). After we examine all the neighbors of the pECDAG (i.e., the pECDAG is searched), we remove the pECDAG from the un-searched pool.

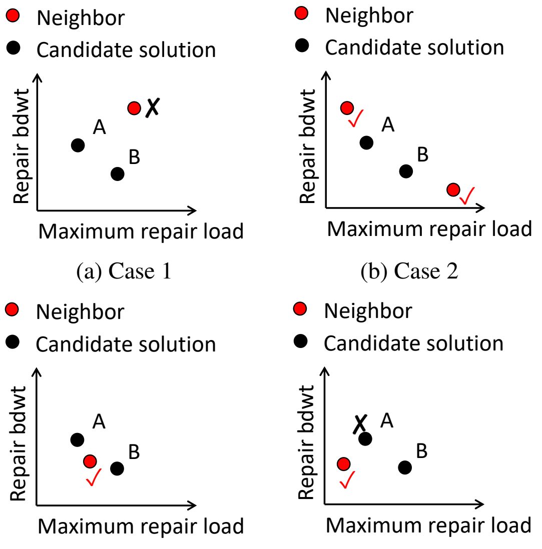
Step 3: Pruning. After Step 2, we generate $\alpha \times (\beta - 1)$ new neighbors of a pECDAG. However, not all of them are suitable for future search. Here, we consider different cases of how we compare a neighbor pECDAG that we generate in Step 2 with the pECDAGs in the candidate pool to decide whether the neighbor pECDAG is suitable for future search. Suppose that there are two pECDAGs in the candidate pool (say, A and B), and Figure 6 shows the four cases when we compare a neighbor pECDAG with the solutions in the candidate pool:

- Case 1 (Figure 6(a)): This is an example of a generated neighbor pECDAG that is not suitable for future search. If there exists a pECDAG in the candidate pool that provides less maximum repair load and less repair bandwidth than the neighbor that we generate in Step 2, it means that we already have an existing solution that outperforms the neighbor. Thus, we discard the neighbor. The remaining three cases show the examples of when we can add a neighbor pECDAG to the candidate pool.
- Case 2 (Figure 6(b)): If the neighbor has the least maximum repair load or the least repair bandwidth compared with all the pECDAGs in the candidate pool, we can add the neighbor to the candidate pool.
- Case 3 (Figure 6(c)): If the neighbor lies between two solutions in the candidate pool, we can add the neighbor to the candidate pool.
- Case 4 (Figure 6(d)): If we find that the repair bandwidth and the maximum repair load of the neighbor are both less than those of an existing pECDAG in the candidate pool, we can add the neighbor to the candidate pool and also remove the existing one from the candidate pool.

For the neighbors that have been added to the candidate pool, we also add them to the un-searched pool for our future search.

Note that Step 2 and Step 3 are performed iteratively until the un-searched pool is empty. Then, we report the pECDAG that has the least maximum repair load as an approximate MLP from the candidate pool.

Discussion. As our heuristic in general only provides a local optimal solution, we can repeat the process to find an approximate MLP starting from Step 1 by multiple runs, such that we can choose the one with the least maximum repair load from all the runs. We show in §7.2 how the heuristic performs in finding the approximate MLP.



(a) Case 1

(b) Case 2

(c) Case 3

(d) Case 4

Figure 6: Compare a neighbor generated in Step 2 with solutions in the candidate pool.

5 Design of Repair Operations

Repair operations in a distributed storage system will be triggered in two scenarios, namely *degraded reads*, where a client reads an unavailable block, and *full-node recovery*, where the distributed storage system repairs the lost blocks of a failed node in a new node. In this section, we design the two repair operations based on our heuristic in §4.

Degrade reads. A client issues a degraded read operation when it requests an unavailable block, in which it needs to repair the requested block through the available blocks of the same stripe stored in other nodes. We generate an approximate MLP from our heuristic. We then associate the approximate MLP with a pECDAG, which describes the repair workflow with the sub-blocks (including the available sub-blocks, repaired sub-blocks, and intermediate sub-blocks) and the nodes (i.e., the nodes that store the available blocks and the node associated with client). Note that a pECDAG varies for different unavailable blocks of a stripe. Also, as the blocks of different stripes are distributed across different sets of nodes in a distributed storage system, a pECDAG varies across different stripes and needs to be generated for each requested unavailable block of a stripe.

Full-node recovery. In a full-node recovery operation, a new node is added to the system, and we repair all the lost blocks of a failed node and store the repaired blocks in the new node. We run our heuristic to generate an approximate MLP for each lost block to be repaired and associate the approximate MLP with a pECDAG. For each block, we associate the colors in the corresponding pECDAG with both the nodes that store the available blocks in the stripe and the new node that is added for full-node recovery.

6 ParaRC

We propose a parallel repair middleware, ParaRC, to balance the repair bandwidth and the maximum repair load for MSR codes. We first introduce the architecture of ParaRC in §6.1. We then elaborate on the implementation details in §6.2.

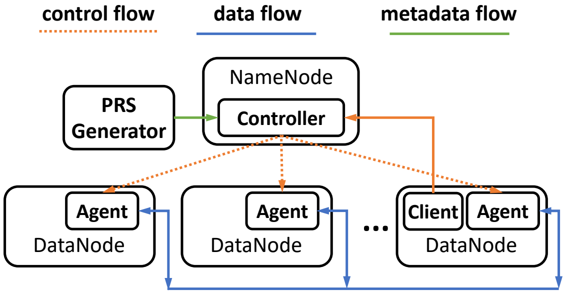


Figure 7: System architecture.

6.1 Architecture

We have built ParaRC as a repair middleware based on OpenEC [19], an erasure coding framework that supports the deployment of custom erasure coding solutions in existing distributed storage systems. ParaRC leverages OpenEC to deploy the parallel repair of MSR codes on Hadoop HDFS [4]. HDFS stores data in fixed-size blocks. It comprises a *NameNode* and multiple *DataNodes*: the NameNode manages the storage of all DataNodes and maintains the metadata of all stored blocks, while the DataNodes provide the storage for the blocks. ParaRC performs encoding across HDFS blocks: for an (n,k) code, it encodes every k uncoded HDFS blocks (i.e., data blocks) into $n-k$ coded HDFS blocks (i.e., parity blocks) to form a stripe, and stores the stripe in n DataNodes.

Figure 7 shows the architecture of ParaRC when it is integrated with HDFS. ParaRC includes a parallel repair solution generator, called the *PRS generator*. It also deploys a *controller* that runs within the NameNode, and multiple *agents*, each of which runs within a DataNode. We also deploy a *client* that is co-located with an agent in a DataNode to issue repair requests to ParaRC (note that the client can also run in a standalone machine outside of the DataNodes). We now elaborate on each component in detail.

PRS generator. The PRS generator pre-computes the parallel repair solution for each single-block repair scenario *offline* and stores the results before the system starts [16]; this offline approach is suitable since the number of repair scenarios is limited for moderate ranges of (n,k) that are commonly used in practice [26]. The PRS generator runs the heuristic in §4 for an MSR code to generate a parallel repair solution that operates at an approximate MLP. It constructs a pECDAG based on the parallel repair solution. It stores the solution in the controller, which coordinates the actual repair operation.

Controller. The controller coordinates the parallel repair operation for the lost blocks that are encoded with MSR codes. Upon receiving a repair request for a block, the controller first reads the metadata of the block from HDFS to determine the location of other blocks in the same stripe. Then, the controller constructs a pECDAG to repair the block with the parallel repair solution returned from the PRS generator. Finally, it translates the pECDAG into a set of *basic tasks* defined in OpenEC [19], including (i) reading sub-blocks from disk, (ii) fetching sub-blocks from other nodes, (iii) computing intermediate sub-blocks and repaired sub-blocks, and (iv) persisting the repaired sub-blocks as the final repaired block. Then, the controller sends the basic tasks to the agents to perform sub-block repair operations to repair a lost block.

Agent. Each agent performs the basic tasks assigned by the controller. For a reading task, an agent directly reads the sub-blocks of a block stored in the local file system. For a fetching task, an agent downloads the sub-blocks from another agent. For a computing task, an agent generates the intermediate sub-blocks or repaired sub-blocks. For a persisting task, an agent stores the repaired sub-blocks as the final repaired block.

Client. A client sends repair requests to ParaRC. It can send a degraded read request or a full-node recovery request to ParaRC (§5). For a degraded read request, ParaRC coordinates the parallel repair for an unavailable block requested by the client; for a full-node recovery request, ParaRC repairs all lost blocks of a failed DataNode in parallel in a new DataNode.

6.2 Implementation

We implement ParaRC in C++ with around 9.4 K LoC and integrate ParaRC with Hadoop-3.3.4 HDFS [4] (HDFS-3 for short). ParaRC uses Redis [7] for internal communications among the controller, agents, and clients. It uses Intel's Intelligent Storage Acceleration Library (ISA-L) [6] to perform encoding and decoding operations for erasure codes. It supports both the centralized repair and parallel repair for MSR codes. In the following, we elaborate on the deployment details of ParaRC and how ParaRC is integrated with HDFS-3.

Deployment. To generate basic tasks for parallel repair, we need to carefully co-locate sub-block repair operations to avoid redundant data transmissions. For example, when we deploy the pECDAG in Figure 4(a), we need to co-locate the repair of sub-blocks v_{18} and v_0, to make sure that the sub-blocks v_8 and v_{16} are only downloaded once in N_2 in the sub-block repair operation. To achieve this goal, we first divide vertices into groups based on topological sorting, in which we can co-locate the sub-block repair operations for the vertices of the same color in the same group.

For example, the vertices in Figure 4(a) can be divided into the following five groups according to topological sorting: (i) v_4, v_5, v_8, v_9, v_{12}, and v_{13}; (ii) v_{16} and v_{17}; (iii) v_0, v_1, v_{18}, and v_{19}; (iv) v_2 and v_3; and (v) R. In group (ii), as v_{16} and v_{17} have the same color, we can co-locate the two sub-block repair operations, such that N_2 can only download sub-block v_{12} from N_3 only once to compute the two sub-blocks. Similarly, we can co-locate the sub-block repair operations specified by v_0 and v_{18} in N_2, and the sub-block repair operations specified by v_1 and v_{19} in N_0.

HDFS-3 integration. To improve parallelism, in ParaRC, the encoding of a stripe of blocks is divided into the encoding of multiple small sub-stripes, where the data unit in each node within a sub-stripe is called a *packet*. In MSR codes, each packet contains w sub-packets. Each sub-stripe encodes $k \times w$ sub-packets into $n \times w$ MSR-coded sub-packets, where the size of a sub-packet is as small as 64 KiB. Thus, we implement sub-packetization across sub-packets instead of sub-blocks as in OpenEC [19], so that ParaRC can encode different sub-stripes in parallel to fully utilize the system resources.

Note that HDFS-3 does not directly support MSR codes, so we rely on ParaRC to generate MSR-coded blocks and store them in HDFS-3. To enable the parallel repair for MSR codes in HDFS-3, we run the ParaRC controller with the NameNode and run each ParaRC agent with a DataNode. The controller maintains a *stripe store* for MSR-coded stripes, which records the metadata of each stripe, including the blocks of the same stripe, and the location of each block in the same stripe. We store the metadata of HDFS-3 blocks in the stripe store of ParaRC, such that when repairing a block, the controller can retrieve metadata from the stripe store.

Support for RS codes. ParaRC also supports RS codes. It now implements both the conventional centralized repair approach and the parallel repair approach based on repair pipelining [20]. In repair pipelining, we divide a packet into sub-packets and pipeline the repair of different sub-packets across a repair path (i.e., each sub-packet is viewed as a slice in repair pipelining [20]). The corresponding parallel repair solutions are stored in the PRS generator. Note that RS codes have no sub-packetization and a sub-stripe encodes k packets into n RS-coded packets.

7 Evaluation

We conduct experiments for ParaRC on Alibaba Cloud [1]. We aim to answer the following questions:

- What is the performance of our heuristic in §4 in finding the approximate MLP? (§7.2)
- How does the performance of ParaRC vary across different system configurations? (§7.3 and §7.4)
- What is the performance overhead of ParaRC to HDFS-3 and how is the repair performance improved by the parallel repair from ParaRC? (§7.5)

7.1 Setup

Testbed. We provision 23 memory-optimized instances on Alibaba Cloud [1] for ParaRC, which includes the PRS generator, the controller, 20 agents, and a node that serves as the client for degraded reads or the new node for full-node recovery. The PRS generator runs on an `ecs.r7.2xlarge` instance with 8 vCPUs and 64 GiB RAM, while other components are deployed in `ecs.r7.xlarge` instances with 4 vCPUs and 16 GiB RAM. Each instance is also equipped with a 40 GiB enhanced SSD with performance level PL0 [2] and is installed with Ubuntu 18.04. All instances are connected via a 10 Gbps network.

Default settings. We configure the default block size as 256 MiB and the default sub-packet size as 64 KiB; for example, the packet size for the $(14,10)$ Clay code is 256×64 KiB $= 16$ MiB, so a stripe can be divided into 16 sub-stripes. In our evaluation, we compare four repair approaches: (i) the centralized repair for RS codes (RS); (ii) repair pipelining for RS codes (RP); (iii) the centralized repair for Clay/Butterfly codes (Clay/Butterfly); and (iv) the parallel repair for Clay/Butterfly codes (ParaRC). If we consider an (n,k) Clay/Butterfly code, we also use the same (n,k) for RS and RP.

For degraded reads, we evaluate the average degraded read time for the first k uncoded blocks. For full-node recovery, we measure the total time of repairing 20 lost blocks of a failed storage node from 20 stripes (whose available blocks are randomly distributed across the non-failed storage nodes). We plot the average results over 5 runs, including the error bars showing the maximum and minimum of the 5 runs.

7.2 Finding the Approximate MLP

E1: Performance of finding the approximate MLP. We evaluate our heuristic in §4 in finding the approximate MLP. We focus on repairing B_0 for Clay codes [36] and Butterfly codes [24]. We evaluate the algorithm running times of our heuristic and the brute-force approach. We also compare the maximum repair load and repair bandwidth of the approximate MLP returned by our heuristic with those of RP and the centralized repair for Clay/Butterfly codes.

We first compare the running time of our heuristic with that of the brute-force approach. We only consider the $(4,2)$ Clay code ($w = 4$) and the $(6,4)$ Butterfly code ($w = 8$), as the brute-force approach for large (n,k) cannot be solved within reasonable time. As shown in Table 2, for the $(4,2)$ Clay code, the heuristic reduces the running time from 264.1 s to 1.8 s, while for the $(6,4)$ Butterfly code, the heuristic reduces the running time from 34.2 s to 0.3 s. We also examine the number of pECDAGs being examined by the heuristic. For example, for the $(14,10)$ Clay code, the heuristic examines about 14 million pECDAGs only; the number is much less than the lower bound of the number of pECDAGs (i.e., 14^{256}) that need to be examined by the brute-force approach (§3.3). Thus, the heuristic significantly reduces the search space.

We note that the heuristic can find the solution whose maximum repair load has the same size as the block size, but sometimes cannot. For example, the solution for the $(6,4)$ Butterfly code ($w = 8$) achieves the maximum repair load of 256 MiB (which is also the minimum), while the maximum repair load of the solution for the $(4,2)$ Clay code ($w = 4$) is larger than the block size 256 MiB. The reason is that the heuristic may return a local optimal solution.

We then compare the maximum repair load and repair bandwidth of different repair approaches. The maximum repair load of our heuristic is significantly less than that of the centralized repair for Clay/Butterfly codes. For example, for the $(14,10)$ Clay code ($w = 256$) the maximum repair load of the MLP decreases to 271 MiB, which is 67.4% less than that of the centralized repair for Clay codes (i.e., 832 MiB). We also

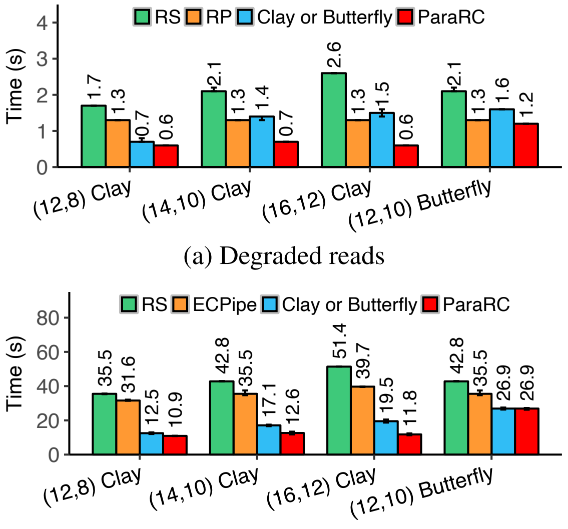
(n,k,w)	**RP**	**Clay**	**Approximate MLP**	**Heuristic**	**Brute-force**
(4,2,4)	(256,512)	(384,384)	(320,448)	1.8 s	264.1 s
(12,8,64)	(256,2048)	(704,704)	(264,1224)	425.9 s	-
(14,10,256)	(256,2560)	(832,832)	(271,1609)	57.2 h	-
(16,12,256)	(256,3072)	(960,960)	(281,1774)	61.9 h	-

(a) Clay codes

(n,k,w)	**RP**	**Butterfly**	**Approximate MLP**	**Heuristic**	**Brute-force**
(6,4,8)	(256,1024)	(640,640)	(256,896)	0.3 s	34.2 s
(12,10,512)	(256,2560)	(1408,1408)	(297,2216)	31.64 h	-

(b) Butterfly codes

Table 2: E1: Performance of finding the approximate MLP to repair B_0 for Clay codes and Butterfly codes. We show the (maximum repair load, repair bandwidth) for each repair approach. We also show the running time for the heuristic. For the brute-force approach, we only show the running time for the $(4,2,4)$ Clay code and $(6,4,8)$ Butterfly code, as the other configurations cannot be completed within reasonable time.



(a) Degraded reads

(b) Full-node recovery

Figure 8: E2: Varying MSR codes.

observe that when the maximum repair load decreases, the repair bandwidth of our heuristic is higher than that of the centralized repair, while it is still much less than the repair bandwidth of RP (by 37.1%). We also observe similar trends in Butterfly codes.

7.3 ParaRC Performance

We evaluate the performance of ParaRC in degraded reads and full-node recovery under different settings.

E2: Varying MSR codes. We evaluate the performance of ParaRC for different MSR code configurations, including the $(12,8)$ Clay code, the $(14,10)$ Clay code, the $(16,12)$ Clay code, and the $(12,10)$ Butterfly code. Figure 8 shows the evaluation results.

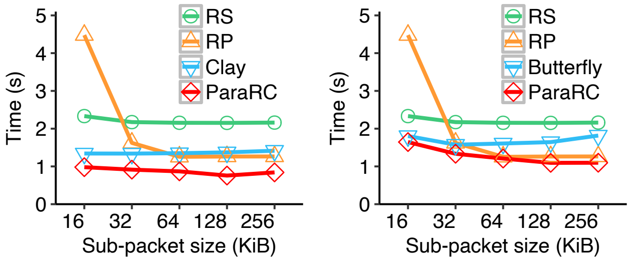
We first analyze the performance of the degraded reads, as shown in Figure 8(a). Overall, ParaRC has the smallest degraded read time compared with other baseline repair approaches. For example, for the $(16,12)$ Clay code, ParaRC reduces the degraded read time by 76.4%, 51.9%, and 59.3%, compared with RS, RP, and Clay, respectively. Although RP minimizes the maximum repair load, its degraded read time is not necessarily minimized, as RP still has high repair bandwidth and needs to read the whole block from each available node in degraded reads. For the $(12,10)$ Butterfly code, ParaRC reduces the degraded read time by 43.8%, 3.7%, and 24.8% compared with RS, RP, and Butterfly, respectively.

We next analyze the performance of full-node recovery, as shown in Figure 8(b). Like degraded reads, ParaRC also has the smallest full-node recovery time compared with other baseline repair approaches. For example, for the $(16,12)$ Clay code, ParaRC reduces the full-node recovery time by 76.9%, 70.2%, and 39.2% compared with RS, RP, and Clay, respectively. For the $(12,10)$ Butterfly code, ParaRC reduces the full-node recovery time by 37.2% and 24.2% compared with RS and RP, respectively. We observe that the network bandwidth usages of the centralized repair for the $(12,8)$ Clay code, the $(14,10)$ Clay code, the $(16,12)$ Clay code, and the $(12,10)$ Butterfly code are 1,126 MiB/s, 973 MiB/s, 984 MiB/s, and 1,046 MiB/s, respectively, implying that it is bottlenecked by the high maximum repair load at the new node where the lost blocks are reconstructed. As ParaRC reduces the maximum repair load, we observe that it significantly improves the repair performance for Clay codes. However, we also observe that the performance improvement of ParaRC is marginal for Butterfly codes. The reason is that while the maximum repair load reduces for Butterfly codes, the repair bandwidth also increases (i.e., from 1,408 MiB to 2,216 MiB), thereby limiting the performance improvements.

7.4 Micro-benchmarks

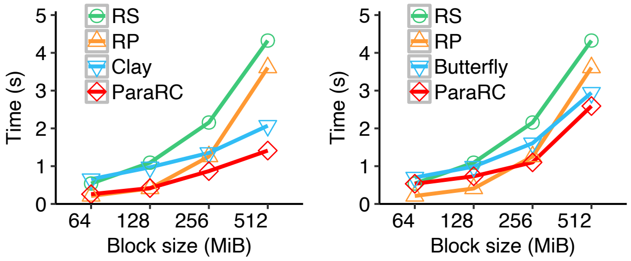
We study how the performance of ParaRC varies for different sub-packet sizes and block sizes. We focus on the $(14,10)$ Clay code and the $(12,10)$ Butterfly code.

E3: Varying sub-packet size for degraded reads. We evaluate ParaRC under different sub-packet sizes. We vary the sub-packet size from 16 KiB to 256 KiB, and fix the block size at 256 MiB (note that the packet size is the sub-packet size multiplied by w, where w depends on the erasure code).



(a) (14,10) Clay code (b) (12,10) Butterfly code

Figure 9: E3: Varying sub-packet size for degraded reads.



(a) (14,10) Clay code (b) (12,10) Butterfly code

Figure 10: E4: Varying block size for degraded reads.

Figure 9 shows the results. For the (14,10) Clay code, ParaRC has the smallest degraded read time compared with other repair approaches for all the sub-packet sizes being considered. For example, when the sub-packet size is 128 KiB, ParaRC reduces the degraded read time by 64.8%, 40.0%, and 44.8% compared with RS, RP, and Clay, respectively. For each repair approach, we see a performance drop when the sub-packet size decreases to 16 KiB due to the overhead of processing a large number of sub-packets, and such a performance drop is especially significant for RP. For example, when we decrease the sub-packet size from 64 KiB to 16 KiB, the degraded read time of ParaRC increases by 12.8%. However, ParaRC still outperforms other baseline repair approaches.

For the Butterfly code, we also observe similar results. For all sub-packet sizes, ParaRC has the smallest degraded read time. For example, when the sub-packet size is 128 KiB, ParaRC reduces the degraded read time by 49.2%, 13.3%, and 33.4% compared with RS, RP, and Butterfly, respectively. When the sub-packet size decreases from 64 KiB to 16 KiB, the degraded read time increases by 26.5%.

E4: Varying block size. We evaluate ParaRC under different block sizes. We vary the block size from 64 MiB to 512 MiB, and fix the sub-packet size at 64 KiB. The block size determines the number of sub-stripes; for example, for the default block size of 256 MiB, the number of sub-stripes for the (14,10) Clay code is 16 (§7.1).

Figure 10 shows the results. When the block size is large, ParaRC outperforms other repair approaches. For example, when the block size is 512 MiB, ParaRC for the (14,10) Clay code reduces the degraded read time by 67.3%, 60.9%, and 31.8% compared with RS, RP, and Clay, respectively, while ParaRC for the (12,10) Butterfly code reduces the degraded read time by 40.3%, 28.4%, and 12.3% compared with RS, RP, and Butterfly, respectively.

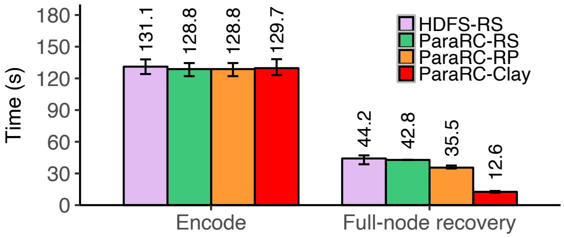


Figure 11: E5: HDFS-3 integration.

When the block size is small, RP outperforms ParaRC. For example, when the block size is 64 MiB, ParaRC for the (14,10) Clay code has 22.1% higher degraded read time than RP. The reason is that ParaRC fails to benefit from the parallel repair for small block sizes due to high sub-packetization. For example, when the block size is 64 MiB, a stripe is only divided into 4 sub-stripes that are repaired in parallel for the (14,10) Clay code, so the degree of parallelism is low. In contrast, RP can pipeline the repair of 1,024 sub-stripes in parallel (§6.2) and outperform ParaRC for small block sizes.

7.5 Performance in HDFS-3

E5: HDFS-3 integration. We evaluate the integration of ParaRC into HDFS-3. Recall that we have shown the benefits of ParaRC over other repair approaches (E2-E4). In this experiment, we only focus on the performance overhead and performance gain of ParaRC in HDFS-3 deployment. We focus on the (14,10) Clay code with the default block size of 256 MiB.

Currently, HDFS-3 does not support Clay codes in its codebase, so we mainly compare ParaRC with the RS code implementation in HDFS-3. We focus on evaluating the overhead of encoding data by Clay codes in ParaRC and the full-node recovery performance gain of ParaRC. We omit the results for degraded reads to a lost block. The reason is that in HDFS-3, a degraded read is triggered when reading a file, where all blocks of the original file are returned to the client anyway. In this case, the centralized repair of the lost block is sufficient.

We evaluate the performance of encoding 20 stripes and repairing 20 lost blocks of a failed node in full-node recovery (the full-node recovery procedure is described in §7.1). We consider four approaches: (i) encoding by the default RS codes and performing the default recovery approach in HDFS (denoted by HDFS-RS); (ii) encoding by RS codes and performing the centralized repair for RS codes in ParaRC (denoted by ParaRC-RS); (iii) encoding by RS codes and performing repair pipelining [20] in ParaRC (denoted by ParaRC-RP); and (iv) encoding by Clay codes and performing the parallel repair in ParaRC (denoted by ParaRC-Clay).

Figure 11 shows the results. For encoding, we observe that the encoding of Clay codes in ParaRC and that of the encoding of RS codes in HDFS-3 have similar overhead. For example, HDFS-RS takes 131.1 s, while ParaRC-Clay takes 129.7 s for encoding 20 stripes. For full-node recovery, ParaRC-Clay reduces the full-node recovery time by 71.4% compared with HDFS-RS; note that the total repair bandwidth for the full-node recovery of 20 lost blocks in ParaRC-Clay is 16.25 GiB, while that in HDFS-RS is 50 GiB (where $(n,k) = (14,10)$).

8 Related Work

RS codes [31] are popularly deployed in distributed storage systems [3, 5, 9, 11, 23, 25], but incur high repair bandwidth (§2.1). Thus, research efforts are made to improve the repair performance of RS codes. One direction is to design fast repair algorithms over RS codes, while another direction is to design regenerating codes to minimize the repair bandwidth.

Repair algorithms for RS codes. PPR [22] divides the repair of a block into partial operations and parallelizes them for improved repair performance. Repair pipelining [18, 20] divides the repair of a block into the repair of small slices, organizes the available nodes that participate in the repair operation into a repair path, and pipelines the repair of slices along the repair path to reduce the degraded read time to be almost the same as the time of reading a block. PPT [8], SMFRepair [39], and PivotRepair [38] propose different parallel repair strategies for RS codes in heterogeneous bandwidth environments. However, the above repair algorithms do not reduce the repair bandwidth of RS codes. Our work focuses on designing parallel repair algorithms for regenerating codes, which have much lower repair bandwidth than RS codes.

Regenerating codes. Regenerating codes [10] are a family of erasure codes that minimize the repair bandwidth. Minimum-storage regenerating (MSR) codes not only minimize the repair bandwidth, but also achieve the MDS property. Many research studies propose new designs of MSR codes, including F-MSR codes [13], PM-RBT codes [27], Butterfly codes [24], and Clay codes [36]. Such MSR codes operate in different parameter regimes, such as $n-k=2$ for F-MSR codes [13] and Butterfly codes [24], and $n \geq 2k-1$ for PM-RBT codes [27]. In particular, Clay codes [36] are state-of-the-art MSR codes that support general parameters of n and k and are proven to minimize both repair bandwidth and I/Os (§1). Geometric partitioning [34] builds on Clay codes and divides an object into variable-sized blocks to trade between the performance of degraded reads and full-node recovery. However, the repair strategy for existing MSR codes is still based on the centralized repair approach, in which a node downloads the required data from all available nodes to repair a failed block. Even though the repair bandwidth is still the minimum, the maximum repair load is high. ParaRC addresses this trade-off by proposing a parallel repair strategy for MSR codes.

DAG-based erasure coding. OpenEC [19] proposes an ECDAG abstraction to model and configure erasure coding operations as a directed acyclic graph (DAG) without modifying the I/O workflows of the underlying distributed storage system. RepairBoost [21] proposes a DAG abstraction to load-balance the full-node recovery workflow. Our work extends ECDAG [19] to support the parallel repair for MSR codes.

9 Conclusions and Future Work

We present ParaRC, a parallel repair framework that improves the repair performance of MSR-coded distributed storage systems by exploiting the sub-packetization nature of MSR codes. We show that there is a trade-off between the repair bandwidth and the maximum repair load. ParaRC builds on a fast heuristic that aims to minimize the maximum repair load, while maintaining the low repair bandwidth. We implement ParaRC as a middleware that runs atop HDFS. Our evaluation results on Alibaba Cloud demonstrate that ParaRC improves the repair performance of degraded reads and full-node recovery compared with the conventional centralized repair approach for Clay codes and Butterfly codes as well as the repair pipelining approach for RS codes.

We discuss the limitations of our work and pose the following open issues for future work.

- Currently, we only empirically show the performance gain of ParaRC, but the theoretical analysis of the ParaRC design remains unexplored. Some open theoretical issues include: (i) the formulation of a multi-objective optimization problem that minimizes both the repair bandwidth and the maximum repair load; (ii) the difference between the returned solution of the heuristic in §4 and the MLP; (iii) the convergence of the heuristic in §4 to the MLP; and (iv) faster and more efficient heuristics.
- ParaRC now focuses on the repair parallelism within a single stripe (i.e., intra-stripe parallelism). One optimization is to extend ParaRC with the repair parallelism across multiple stripes (i.e., inter-stripe parallelism) for further performance gains, particularly in declustered settings where the stripes span across a large number of nodes [12]. Also, the full-node recovery of ParaRC currently stores all reconstructed blocks in a new node that replaces the failed node. We can extend it to distribute the reconstructed blocks across different nodes to avoid bottlenecking the new node.
- ParaRC is currently designed for large blocks and the moderate parameters n and k. In future work, we consider small blocks and wide stripes [14] (wide stripes encode data with large parameters n and k for ultra-low redundancy).

Acknowledgments. We thank our shepherd, Gala Yadgar, and the anonymous reviewers for their comments. This work was supported in part by the National Key Research and Development Program of China for Young Scholars (No. 2021YFB0301400), Key Laboratory of Information Storage System Ministry of Education of China, the National Natural Science Foundation of China (No. 61821003 and No. 61832007), and Research Grants Council of HKSAR (AoE/P-404/18). The corresponding author is Yuchong Hu.

References

[1] Alibaba Cloud. https://www.alibabacloud.com/product/ecs-pricing-list/en.

[2] Alibaba Cloud - ESSDs. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/essds.

[3] Ceph - Erasure code. http://docs.ceph.com/docs/master/rados/operations/erasure-code/.

[4] Hadoop 3.3.4. https://hadoop.apache.org/docs/r3.3.4/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[5] HDFS erasure coding. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[6] Intel's intelligent storage acceleration library (ISA-L). https://www.intel.com/content/www/us/en/developer/tools/isa-l/overview.html.

[7] redis.io. https://redis.io/.

[8] Yunren Bai, Zihan Xu, Haixia Wang, and Dongsheng Wang. Fast recovery techniques for erasure-coded clusters in non-uniform traffic network. In *Proc. of ICPP*, 2019.

[9] Brian Beach. Backblaze Vaults: Zettabyte-scale cloud storage architecture. https://www.backblaze.com/blog/vault-cloud-storage-architecture/, 2019.

[10] 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.

[11] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.

[12] Mark Holland, Garth A. Gibson, and Daniel P. Siewiorek. Architectures and algorithms for on-line failure recovery in redundant disk arrays. *Distributed Parallel Databases*, 2(3):295–335, 1994.

[13] Yuchong Hu, Henry C. H. Chen, Patrick P. C. Lee, and Yang Tang. NCCloud: Applying network coding for the storage repair in a cloud-of-clouds. In *Proc. of USENIX FAST*, 2012.

[14] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. Exploiting combined locality for wide-stripe erasure coding in distributed storage. In *Proc. of USENIX FAST*, 2021.

[15] 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.

[16] Osama Khan, Randal C Burns, James S. Plank, William Pierce, and Cheng Huang. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In *Proc. of USENIX FAST*, 2012.

[17] Oleg Kolosov, Gala Yadgar, Matan Liram, Itzhak Tamo, and Alexander Barg. On fault tolerance, locality, and optimality in locally repairable codes. In *Proc. of USENIX ATC*, 2018.

[18] Runhui Li, Xiaolu Li, Patrick P. C. Lee, and Qun Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, 2017.

[19] Xiaolu Li, Runhui Li, Patrick P. C. Lee, and Yuchong Hu. OpenEC: Toward unified and configurable erasure coding management in distributed storage systems. In *Proc. of USENIX FAST*, 2019.

[20] Xiaolu Li, Zuoru Yang, Jinhong Li, Runhui Li, Patrick P. C. Lee, Qun Huang, and Yuchong Hu. Repair pipelining for erasure-coded storage: Algorithms and evaluation. *ACM Trans. on Storage*, 17(2):13:1–13:29, 2021.

[21] Shiyao Lin, Guowen Gong, Zhirong Shen, Patrick P. C. Lee, and Jiwu Shu. Boosting full-node repair in erasure-coded storage. In *Proc. of USENIX ATC*, 2021.

[22] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage. In *Proc. of ACM EuroSys*, 2016.

[23] 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.

[24] Lluis Pamies-Juarez, Filip Blagojevic, Robert Mateescu, Cyril Guyot, Eyal En Gad, and Zvonimir Bandic. Opening the chrysalis: On the real repair performance of msr codes. In *Proc. of USENIX FAST*, 2016.

[25] Andreas-Joachim Peters, Michal Kamil Simon, and Elvin Alin Sindrilaru. Erasure coding for production in the EOS open storage system. In *Proc. of CHEP*, 2019.

[26] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *Proc. of USENIX FAST*, 2009.

[27] K. V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *Proc. of USENIX FAST*, 2015.

[28] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proc. of USENIX HotStorage*, 2013.

[29] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proc. of ACM SIGCOMM*, 2014.

[30] K. V. Rashmi, Nihar B. Shah, and Kannan Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. *IEEE Trans. on Information Theory*, 63(9):5802–5820, 2017.

[31] 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.

[32] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing elephants: Novel erasure codes for big data. *Proc. of the VLDB Endowment*, 6(5):325–336, 2013.

[33] Nihar B. Shah, K. V. Rashmi, P. Vijay Kumar, and Kannan Ramchandran. Interference alignment in regenerating codes for distributed storage: Necessity and code constructions. *IEEE Trans. on Information Theory*, 58(4):2134–2158, 2012.

[34] Yingdi Shan, Kang Chen, Tuoyu Gong, Lidong Zhou, Tai Zhou, and Yongwei Wu. Geometric partitioning: Explore the boundary of optimal erasure code repair. In *Proc. of ACM SOSP*, 2021.

[35] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Trans. on Information Theory*, 2012.

[36] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, Syed Hussain, and Siddhartha Nandi. Clay codes: Moulding MDS codes to yield an MSR code. In *Proc. of USENIX FAST*, 2018.

[37] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, 2002.

[38] Qiaori Yao, Yuchong Hu, Xinyuan Tu, Patrick P. C. Lee Lee, Dan Feng, Xia Zhu, Xiaoyang Zhang, Zhen Yao, and Wenjia Wei. PivotRepair: Fast pipelined repair for erasure-coded hot storage. In *Proc. of ICDCS*, 2022.

[39] Hai Zhou, Dan Feng, and Yuchong Hu. Multi-level forwarding and scheduling repair technique in heterogeneous network for erasure-coded clusters. In *Proc. of ICPP*, 2021.

InftyDedup: Scalable and Cost-Effective Cloud Tiering with Deduplication

Iwona Kotlarska
9LivesData, LLC

Andrzej Jackowski
9LivesData, LLC

Krzysztof Lichota
9LivesData, LLC

Michal Welnicki
9LivesData, LLC

Cezary Dubnicki
9LivesData, LLC

Konrad Iwanicki
University of Warsaw

Abstract

Cloud tiering is the process of moving selected data from on-premise storage to the cloud, which has recently become important for backup solutions. As subsequent backups usually contain repeating data, deduplication in cloud tiering can significantly reduce cloud storage utilization, and hence costs.

In this paper, we introduce InftyDedup, a novel system for cloud tiering with deduplication. Unlike existing solutions, it maximizes scalability by utilizing cloud services not only for storage but also for computation. Following a distributed batch approach with dynamically assigned cloud computation resources, InftyDedup can deduplicate multi-petabyte backups from multiple sources at costs on the order of a couple of dollars. Moreover, by selecting between hot and cold cloud storage based on the characteristics of each data chunk, our solution further reduces the overall costs by up to 26%–44%. InftyDedup is implemented in a state-of-the-art commercial backup system and evaluated in the cloud of a hyperscaler.

1 Introduction

Managing the surging volumes of data that require protection or long-term retention increasingly necessitates novel backup strategies [18]. A popular approach is employing cloud-based solutions. For instance, according to Veeam, the number of organizations adopting cloud-powered data protection is expected to rise from 60% in 2020 to 79% in 2024 [66]. Similarly, in a survey by ESG, 72% of the participants confirmed using *tiering* techniques to move colder data (e.g., older backups and archives) from on-premise storage to the cloud [22].

In this context, data *deduplication* can become effective. Since consecutive backups often contain repeating data [47, 50], this technique reduces storage utilization tens of times [70]. As a result, deduplication is a core feature of several storage systems for on-premise backup applications [33, 57, 84]. In this light, for backup use cases, it is sensible to consider *cloud tiering with deduplication*, that is, moving data from a local tier (e.g., on-premise backup appliances) to a cloud tier (e.g., a cloud object store), so that ultimately the data kept in the cloud tier are deduplicated.

However, implementing cloud tiering with deduplication poses two major problems. First, state-of-the-art cloud storage systems provided by hyperscalers (e.g., Amazon, Google, and Microsoft) do not offer deduplication as a core functionality for their clients. Consequently, deduplication algorithms tailored for cloud tiering have to be developed. In the process, the extra tier should be treated not only as a challenge but also a potential opportunity for exploring novel deduplication paradigms dedicated for the cloud. Second, there is a large variety of available cloud storage service types, notably differing in pricing models. Initially, a lower storage cost implied a longer retrieval time (e.g., AWS Glacier) but nowadays, systems like AWS Glacier Instant Retrieval [77] offer the same performance as other cloud storage services. The trade-off is that with a decreased per-byte monthly storage fee, the costs of data retrieval and the minimal data storage period are increased. Therefore, algorithms have to be devised to decide what type of service to use for which data, specifically considering the peculiarities due to deduplication.

As we discuss in more detail further in the paper, despite some research progress, these two problems are largely open. In short, regarding the first problem, although a few backup applications [54, 59] and backend appliances [34, 68] with deduplication offer mechanisms for cloud tiering, they heavily rely on and are implemented mainly in the local tier. In effect, deduplication between different local tier systems is not supported for data stored in the cloud. Moreover, the entire process is fundamentally limited by the resources of the local tier. In other words, despite the possibilities offered by the hyperscalers, the actual scalability of the cloud tier in such solutions is severely limited, proportionally to what is offered by the local tier. When it comes to the second problem, although the diversity of the service models offered by the hyperscalers can also be exploited in some solutions [52], this has to be configured manually or, at best, through policies depending on the ages of data collections. However, deduplication typically entails chunking data collections into smaller pieces that may be referenced multiple times, thereby possibly having different access patterns. This calls for finer-grained and more automated approaches to storage type selection.

In this paper, we address both these problems, introducing solutions for scalable and cost-effective cloud tiering with deduplication. Accordingly, our contribution is twofold.

First, we present InftyDedup, a novel system for cloud tiering with deduplication. Like the existing tiering-to-cloud backup solutions, InftyDedup moves selected data from a local-tier system to the cloud, based on customer-specific backup policies. However, its operation aims to maximize scalability by exploiting cloud services — not only for storage but also for computation. Therefore, rather than relying on deduplication methods of on-premise solutions, InftyDedup deduplicates data using the cloud infrastructure. This is done periodically in batches before actually transferring data to the cloud, which, among others, enables the dynamic allocation of cloud resources. Other functionalities, such as garbage collection of deleted data, are supported in the same way. We integrate InftyDedup with HydraStor [33], a commercial backup system with deduplication, and evaluate its performance in AWS, demonstrating that multiple petabytes can be deduplicated for a couple of dollars. Being highly independent of the local tier, InftyDedup overcomes the limitations of similar state-of-the-art technologies and offers unprecedented scalability. To the best of our knowledge, this is the first application of such solutions to backup systems.

The second contribution is an algorithm for decreasing the financial cost of storing deduplicated data in the cloud tier. It extends InftyDedup by allowing it to move deduplicated data chunks between cloud services dedicated to hot and cold storage. Whereas existing solutions do not address the problem at all or enable some optimizations at the level of data collections (e.g., backups or files), the fact that chunks are deduplicated between backups/files makes them a better unit for optimizations. In InftyDedup, the chunks are moved based on their metadata, notably deduplication reference counts and terse information provided by system administrators on their data collections. Our empirical evaluation of the algorithm shows that mixing storage types can reduce the total financial cost of cloud tiering with deduplication by up to 26–44%.

The rest of the paper is organized as follows. Section 2 gives the background. Section 3 describes the overall architecture and specific algorithms comprising InftyDedup. Section 4 discusses the algorithm for exploiting cold cloud storage for cost minimization. Section 5 presents the experimental results. Section 6 surveys related work. Finally, Section 7 concludes.

2 Background

This section reviews the characteristics of deduplication storage, backups, and cloud services, which are essential to InftyDedup architecture.

2.1 Deduplication Storage

Deduplication is a data reduction technique that avoids writing the same data twice. For data with many duplicates, deduplication reduces the storage capacity requirements of the system [70], increases throughput, and decreases network traffic [3]. Typically, deduplication is implemented in the following steps [79]. Firstly, the data stream is chunked into small immutable blocks of size from 2 KB to 128 KB [71]. Secondly, each block receives a fingerprint, for instance, by computing the SHA-256 hash of the block's data. Finally, the fingerprint is compared with other fingerprints in the system, and if the fingerprint is unique, the block's data is written.

The deduplicated blocks are typically organized in a directed acyclic graph. Each file has its *root block* corresponding to a vertex that references other blocks. The blocks with actual data are leaves of the DAG and keep no references. Blocks with data corresponding to a particular file form a subgraph reachable from the root block representing that file. Therefore, the movement of a deduplicated file to a different tier is effectively the movement of a subset of leaves that are reachable from the root block of the file.

A block can be removed after it is migrated to another system. However, reclaiming storage capacity in the presence of deduplication is nontrivial, as the system must ensure there are no other references to the removed block. Therefore, complex garbage-collecting algorithms that can process blocks' metadata for hours are implemented [36, 69].

The most natural use case for deduplication is backup storage, as most data do not change in consecutive backups. In our research, we leverage the characteristics and lifecycle of backups to decrease the total storage cost.

2.2 Lifecycle of Backups

Typically, backups are created and managed based on assigned retention policies [61]. From the perspective of our research, there are two essential constraints regarding the timing and life cycle of protected data.

On the one hand, the data should be up-to-date and available quickly in case of a disaster. For instance, Zerto reports [82] that their customers achieve Recovery Point Objectives of seconds and Recovery Time Objectives of minutes. To achieve such ambitious objectives, recent data is kept as closely as possible to the infrastructure being recovered.

On the other hand, older versions of backups need to be stored for weeks, months, or even years [65]. As the objective points for older data differ, backups are often moved to cheaper storage after a specific time [55, 83]. Cloud is often chosen to keep the older backups for many reasons, including storing data in a different physical location. The pricing model of cloud storage is also appealing, but as described in the next section, many factors influence the total costs.

2.3 Cloud Storage

The market of cloud storage is mostly shared between three hyperscalers (Amazon Web Services, Microsoft Azure, and Google Cloud) [74], so in our considerations, we assume services offered by the three as a market standard.[1] The portfolio of hyperscalers comprises numerous storage and computing products: from databases, queues, and distributed filesystems to simple storage primitives, such as objects or blocks. Our goal is to minimize the storage cost of backups, so our research focuses on the most affordable products. The lowest price per stored gigabyte is offered by cold archival object stores, which are orders of magnitude cheaper than block devices, as shown in Tab. 1. However, many factors determine the total cost, including fees per request or IO, charges for removing data before meeting the minimal storage duration, and data transfer costs. Accessing data in some types of the coldest storage takes additional time (e.g., 12 hours), but every hyperscaler offers cold storage with instant access [23, 28, 77].

	Amazon Web Services	Microsoft Azure	Google Cloud
Block Storage [$/GB]	0.08	0.15	0.04
Object Storage [$/GB]	0.021	0.0166	0.02
Archival Object Storage [$/GB]	0.004	0.01	0.004
Coldest Archival Object Storage [$/GB]	0.00099	0.00099	0.0012

Table 1: Sample[2]monthly costs of storing blocks and objects in public clouds [5, 7, 26, 27, 49].

Uploading data to the cloud is usually free, whereas the cost of downloading data once a month can outweigh the cost of monthly data storage. In either case, network throughput to the cloud is a major concern. Hyperscalers offer connecting data centers to the cloud directly (e.g., with 100 GbE) [9, 15], but the availability of such networks is limited to specific regions. Alternatively, physical devices can be used for the movement of data [11], but it is rather for niche applications. Therefore, moving terabytes to the cloud can take up days.

2.4 Cloud Computing

The product portfolio of cloud computing services is also versatile. There are virtual machines (e.g., AWS EC2), containers (e.g., AWS ECS), and other services, such as event-driven function execution (e.g., AWS Lambda). Some products are prepared for specific use cases, including machine learning [29] and databases [4, 25].

The pricing model of computation services is typically based on the cost of the lower-level resources. For instance, ECS allows running containers on EC2 instances, so the cost of container execution depends on the amount and size of virtual machines which host the containers [6]. This billing model enables using numerous nodes (e.g., hundreds of servers) for short periods at a very low cost.

What is important for cost reduction, hyperscalers offer so-called *spot instances*, that are virtual machines with a discounted price of up to 90%. Spot instances can be interrupted by a cloud provider at any moment, but the computations interrupted within the first hour are free [13]. The exact price of a spot instance depends on multiple factors (e.g., the momentary demand), but historical data shows that achieving both a very low risk of termination and a significant cost reduction is possible [35]. Virtual machines (including spot instances) can have their local storage (e.g., SSD drives), which is cheaper than network-attached drives but has limited durability as the data are lost if the machine is destroyed or fails.

To minimize the costs of computations, we considered these cloud attributes in InftyDedup architecture, which we describe in the next section.

3 InftyDedup Architecture

InftyDedup moves selected data from local tier systems (i.e., on-premise backup appliances implemented as described in Section 2.1) to the cloud tier. The local tier is expected to have its own deduplication and to be hardware-failure resistant (e.g., by implementing erasure codes or RAID), as it persistently stores local data (e.g., data not selected for tiering). As shown in Fig. 1, the cloud tier stores deduplicated data with necessary persistent metadata, and occasionally executes highly optimized *batch algorithms*.

Before we describe the details of the structures and algorithms, we discuss our study of cloud characteristics (Section 3.1) and the assumptions we made based on them (Section 3.2). After that, we describe the structure of in-cloud data and metadata (Section 3.3), the model of communication between tiers (Section 3.4), and algorithms of deduplication (Section 3.5), garbage collection (Section 3.6), and file restore (Section 3.7).

3.1 Cloud Cost Considerations

We studied the pricing of public clouds to design InftyDedup in line with the current trends. First, we chose product types common for all vendors and compared the pricing models and capabilities of each product with other products of the same vendor. We did not compare pricing between vendors, as our goal was to design cost-efficient architecture for any regular cloud, not choosing a particular vendor.

[1] However, there are numerous innovative services offered by other providers. For instance, the latest trend to decentralize the cloud [62, 63] can help to implement InftyDedup efficiently.

[2] The price of storage products depends on many factors, including region. Each cloud provides many products (e.g., each provider offers more than one cold object store). The prices between providers cannot be compared directly because the products differ. However, there are several categories of cloud storage products similar to the order of magnitude of the price. The table contains list prices as of 2023-01-01.

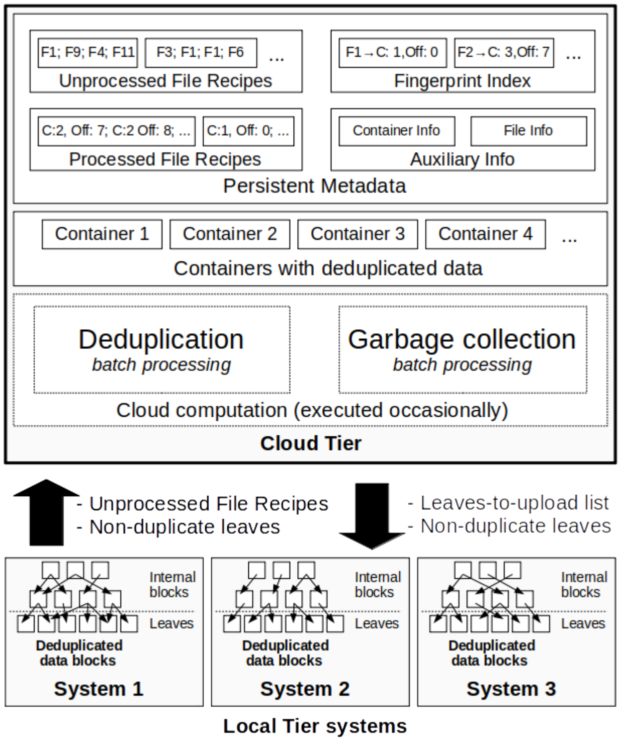


Figure 1: InftyDedup architecture.

Keeping 1 PB of non-deduplicated data in standard cloud object storage costs between $\$16,600$ and $\$21,000$ per month, and between $\$4,000$ and $\$10,000$ for archival object storage with instant access. Therefore, the overall cost of storing data with deduplication, including additional storage for deduplication metadata and costs of computations, must be lower than that to bring any financial benefit.

Assuming a deduplication block size of 8 KB, 10:1 deduplication, and 20 bytes per fingerprint, 1 PB of data requires 262 GB of fingerprints. If new backups of similar size are written each week, over 496 billion fingerprint existence queries to the cloud are needed each month.

Modern architectures of inline deduplication often keep the fingerprint index (or its parts) on SSDs [3,30,48]. Considering a naive approach in which each deduplication query requires a read IO from an SSD drive, at least 190k IOPS are required to perform the necessary queries each month. To estimate the cost, let us consider AWS as an example. The monthly cost of EBS gp3 block storage which provides such an amount of IOPS is $\$978$, and EC2 instances (m5.large) capable of utilizing the IOPS cost $\$3827$. With a total cost of nearly $\$5000$ monthly for just handling deduplication queries, there is still room for a cost benefit from deduplication (depending on the deduplication ratio). Still, the price is significant compared to the cost of storage without deduplication.

These calculations led us to our conclusion that, despite the fact that SSDs provide a high number of random-read IOPS, relying on a random-read-intensive fingerprint index *is not negligibly cheap* in the cloud environment. Although there are techniques that reduce the number of read IOs for traditional sequential workloads [84], their efficiency is decreased for modern non-sequential workloads, which need to be handled in addition to classic sequential workloads, as explained by Y.Allu et al. [2]. Similarly, the efficiency of methods that rely on data locality (like SISL [84]) decreases when data is highly fragmented.[3] Finally, these methods are often not prepared to update block information during deduplication, which is a necessary part of our algorithms for cold storage.

On the other hand, transferring data within the cloud is free of charge, and even the cheapest instance can transfer hundreds of gigabytes per hour [14]. Having the possibility of dynamically scaling resources between zero and hundreds of servers, processing the fingerprint index sequentially with a batch job can be more cost-effective than keeping the fingerprint index online 24/7 or relying on short-lived lambdas [10]. This is particularly true considering up to 10 times cheaper computation using the aforementioned spot instances. This key observation was used when designing the InftyDedup architecture based on assumptions explained in the next section.

3.2 Assumptions and Design Decisions

Our principal assumption is that *our cloud tiering deduplication must be processed outside the local tier* to prevent resource restrictions and enable functionalities like deduplication between many local tier systems. Therefore, all metadata required for deduplication must be stored and processed outside the local tier.

As network throughput between the tiers is limited, *data movement between the tiers should be minimal*. Therefore, only non-duplicate data must be uploaded to the cloud tier. When restoring data, it must be possible to download only data absent from the local tier. However, for efficient disaster recovery, *quick and granular backup restores must be possible*, even when the local tier is unavailable.

The next central assumption is that *batch processing is preferred over streaming processing*. Therefore, the algorithms are executed occasionally (e.g., once a day or week for deduplication and even less frequently for garbage collection). There are multiple reasons for that. Firstly, as our cost analysis of public clouds shows, being prepared for data deduplication 24/7 is not negligibly cheap. Secondly, as explained in Section 2.2, backups are typically moved to the cloud after a specified period, so batch processing can be done without disrupting the data lifecycle. Finally, tiering to cloud with deduplication requires steps that take a significant amount of time: uploading data to the cloud, and running garbage collection in the local tier to reclaim data there. All in all, per-

[3] Fragmentation also concerns restore throughput [40,44]. However, in the case of cloud storage, the read performance scales, and even with random 8 KB reads, the egress traffic cost is equal to the per-request fee.

forming a costly deduplication query with each write brings few benefits in practice, and we decided to use the cheaper option of infrequent batch processing.

Garbage collection in the cloud tier must be cost-aware to ensure that data removal costs are not higher than keeping data for a longer period. Similarly, storing frequently accessed data in cold cloud storage actually increases the costs, so the deduplication and garbage collection algorithms must be extendable with intelligent storage type selection.

Finally, our solution is meant to be suitable for a variety of cloud platforms and providers. Although in our description and evaluation we focus on the most popular hyperscalers, our architecture can be easily adapted to others. In particular, private clouds ensure privacy and compliance, so we verified our solution in our private cloud environment.

3.3 Data and Metadata in Cloud

Based on the assumptions, we designed persistent structures of InftyDedup to be kept in cloud object storage as follows.

The largest structure contains blocks with deduplicated data grouped into *containers*. Selecting the size of containers depends on the cloud pricing, as writing and reading larger containers requires fewer requests but can increase rewriting costs when reclaiming space after garbage collection.

The largest metadata structure contains *file recipes*, which are effectively a list of per-block metadata as they appear in each file. If one block exists in a file multiple times, its metadata also occurs multiple times in its file recipe. There are two types of file recipes. Firstly, there are *unprocessed file recipes* (UFR in short), which are provided by the local tier. UFRs contain the fingerprint of each block, as the local tier does not know the block's cloud location. Later, during deduplication processing, each entry of UFR receives a cloud address of the block it references, so the file recipes are converted to *processed file recipes* (PFR in short). PFRs can be a simple list of cloud addresses or have a tree structure to enable the deduplication of PFR's parts. In the latter case, fingerprints of PFR chunks are added to the fingerprint index, which is described shortly.

The second largest metadata structure is Fingerprint Index (FingIdx in short) which contains a mapping from the deduplication fingerprint of each block to the block's cloud location. FingIdx is expected to be smaller than PFRs, as it contains only one entry per unique fingerprint. FingIdx is bucketed [72] rather than sorted, meaning the fingerprints are divided into thousands of buckets based on a hash function. Such data representation enables optimization of distributed FingIdx processing, as each bucket is small enough to fit into server memory.

There are also a few orders of magnitude smaller structures that keep information per file or container. The metadata structures are compressed to reduce space and network usage.

3.4 Communication between Tiers

The data exchange between the tiers is bidirectional but kept to a minimum, as the network connection between the tiers can easily become a bottleneck. Two types of information are sent from the local tier to the cloud. For each file selected for cloud tiering, the local tier system generates a UFR (a list of fingerprints of all blocks in the file). The UFR is later used as an input to batch deduplication, which generates in return a *blocks-to-upload list* that is, in fact, a list of containers. Each container comprises unique blocks that still need to be uploaded to the cloud tier. Based on the list, the local tier uploads the blocks to the cloud. During a file restore operation, blocks can be later downloaded from the cloud tier.

Therefore, the cloud tier has minimal requirements on the interface of the local tier. It is sufficient that the local tier can generate a UFR and later upload blocks based on the list of fingerprints. The local tier can be composed of multiple systems if each system uses consistent chunking and fingerprinting.

3.5 Batch Deduplication

Batch deduplication (BatchDedup in short) is our distributed method of block deduplication in the cloud. It is expected to be run periodically, in harmony with the schedule of backups and garbage collection in local tier systems. Each execution of BatchDedup is a distributed, fault-tolerant computation that ultimately changes persistent structures kept in the cloud object storage. The computations are divided into steps, and each of the steps comprises smaller jobs that are parallelized and repeated in the event of failure. In our implementation, we used YARN [76] to schedule jobs, and HDFS [64] for reliable storage of temporary data, so the jobs can be run on spot instances as proposed in the AWS guide [12]. The state of computation is maintained by the YARN master node, which can be hosted on a non-spot instance to increase reliability, but even if the entire computation fails, the valid version of metadata always remains in the cloud object storage.

In short, BatchDedup takes UFRs as input, specifies new containers with blocks to be uploaded, waits until the local tier uploads the blocks, and updates persistent metadata. The UFRs are expected to be uploaded to the cloud before BatchDedup is started (partially uploaded UFRs do not take part in the process). The steps are as follows:

Step #1: UFR processing selects blocks that need to be uploaded to the cloud by comparing fingerprints from both UFRs and FingIdx. FingIdx and UFRs are bucketed based on fingerprints, and the buckets are distributed across multiple servers. After that, the fingerprints are compared in batches that are small enough to fit in memory.

Step #2: Container generation splits blocks selected in *Step #1* into containers to generate descriptions for the local tier. Each server processes a subset of blocks, and the blocks are distributed based on their original file (so blocks from the

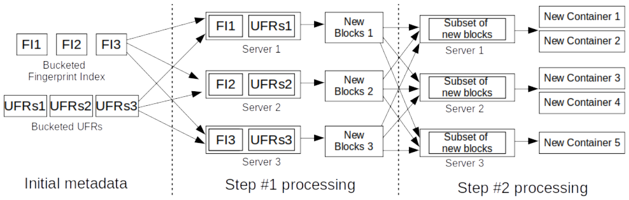


Figure 2: The first two steps of BatchDedup processed in a distributed manner.

same file can be placed in the same container). The blocks are sorted by the order (offsets) in their original files,[4] as preserving the original order makes the latter step of uploading the container easier, and reduces the number of requests for garbage collection and data restores for non-fragmented data.

Step #3: PFR update is conducted after the first two steps, when the block location (its container and offset) is finally known for both new and old blocks. Based on that information, each newly written file receives its PFR.

Step #4: Blocks upload is initiated by the local tier systems. The local tier systems first download the descriptions of new containers (i.e., which blocks should be uploaded to which container). After that, each of the local tier systems uploads the actual data. When the uploads are successfully completed, the in-cloud metadata structures are updated to mark the new files as ready in the cloud.

The first two steps of BatchDedup are depicted in Fig. 2. Similar techniques are used to perform the remaining steps of BatchDedup and garbage collection at scale.

BatchDedup processes FingIdx and all recently uploaded UFRs but does not touch any previously generated PFRs. As UFRs likely contain duplicates, in practice, the size of UFRs is expected to be at least comparable to the size of the whole FingIdx, and with such assumption, processing FingIdx does not dominate the asymptotic cost. Overall, the process is expected to take time: BatchDedup is executed periodically, the computations in Steps #1-#3 take from minutes to hours, and the block upload in Step #4 can even take days, depending on the data volume and network bandwidth. As Step #4 is inevitable in any cloud-tiering solution, the cloud tier alone is not suitable for providing very short RPO. However, as backups are moved to the cloud typically after a specific time, the steps can be scheduled in periods that will not violate the timing constraints of the backup policy.

3.6 Batch Garbage Collection

Batch garbage collection (BatchGC in short) identifies blocks no longer referenced by any PFR and reclaims free space in the containers. BatchGC is expected to be executed periodically but less frequently than BatchDedup. Both algorithms modify the same metadata structures, so they cannot be executed simultaneously. However, file restores are possible at any moment.

PFRs keep the addresses of containers, so rewriting a container requires modifications of PFRs. The cost of processing PFRs is discouraging, as PFRs can be many times larger than FingIdx. However, garbage collection is done only occasionally, so even if it is a few times more expensive than BatchDedup, the overall cost of InftyDedup is not affected that much. Therefore, our primary goal is ensuring scalability, which enables meeting the time constraints of other garbage collection algorithms for deduplication storage [31, 69].

BatchGC comprises the following steps:

Step #1: File removal processes non-removed PFRs to find blocks that are still referenced by at least one file.

Step #2: Container verification checks how many blocks in each container are live. Based on one of the strategies (which we introduce shortly), a set of containers that will be removed or rewritten is selected.

Step #3: Metadata are updated based on the results of *Step #2*. More specifically, new metadata for modified containers are calculated. Some blocks may receive a new address, so new versions of FingIdx and PFRs are also needed.

Step #4: Containers are rewritten to actually reduce space usage. When all newly generated containers are written, the metadata computed in *Step #3* take effect, and old containers are deleted.

Immediate removal of unreferenced data is not always optimal, as rewriting a container in the cloud has a significant cost. Therefore, we investigated three strategies to decide whether a container should be rewritten:

GC-Strategy #1: Reclaim only empty containers. In most cloud services sending a request to remove an entire container is free, so the strategy brings cost reduction (as less capacity needs to be stored) with no additional cost. However, the strategy does not remove containers in which only a fraction of data has been deleted.

GC-Strategy #2: Reclaim containers if the rewrite pays for itself after T days. To determine whether rewriting a container will bring a cost-benefit, the following ratio can be calculated for each container:

$$x = \frac{COST_{rewrite}}{T_{days} * CAPACITY_{to\_be\_reclaimed} * COST_{byte\_per\_day}} \quad (1)$$

Only if $x < 1.0$, rewriting a container is cheaper than storing deleted data from the container for T_{days}. However, picking the proper value of T_{days} is nontrivial. For instance, if T_{days} is the time left until the next BatchGC, the containers are rewritten only if it brings financial benefit before the next chance to remove any data. In many cases, such T_{days} value is too small and will prevent rewriting a container, although rewriting the container would bring a financial benefit in the long run. On the other hand, a large T_{days} value implies frequent rewriting, which can lead to exceeding *Strategy #1* costs.

[4] A block is expected to exist in multiple files or to be repeated within one file. In such a case, only the first appearance is stored in a container.

GC-Strategy #3: Reclaim containers based on file expiration dates. *GC-Strategy #2* can be improved if files contain information about their expiration date (denoted as EXP_{time}). Such information can be provided by the local tier systems in UFRs, if the EXP_{time} results from the backup configuration. Therefore, for each container, T_{days} can be calculated as the maximal EXP_{time} of its blocks (aligned up to the BatchGC schedule). EXP_{time} is expected to increase in time,[5] as new files with later EXP_{time} are stored. However, even with rising EXP_{time}, the cost never exceeds *GC-Strategy #1*, as a non-empty container is rewritten only when it is beneficial.

3.7 File Restore

The cloud metadata format supports straightforward file restores. Each file has its own object, with the key based on the local tier system identifier and file path. Therefore, object storage interface features such as ACLs and per-prefix listings can be used for convenient file management. Based on the PFR, which stores the container address and data offset, the file can be read without accessing the local tier systems. As PFRs are updated during BatchGC, the movement of data between containers during GC does not spoil the reads.

However, egress traffic is a major cost, so restores can be additionally integrated with the local tier for cost reduction. For blocks available locally, the download from the cloud can be omitted. Blocks absent locally can be optionally stored in the local tier system after downloading, as some workloads require reading data again in the near future (e.g., restoring multiple similar VMs). Implementing such local-tier assisted reads requires storing fingerprints in PFRs, which increases the metadata size, but the fingerprints can be easily added and removed from PFRs on-demand in batch algorithms.

4 Cold Storage Utilization

To reduce the cost of storing data in the cloud, InftyDedup can be extended with an algorithm that selects whether a block should be stored in hot or cold cloud storage. We aimed to use cold storage services offering different pricing models than other cloud storage products but comparable durability and latency [28, 77] (otherwise, the movement of data to cold storage negatively affects the recovery time).[6] Therefore, we focused on colder storage which offers a reduced price of storing data but increases the price of restores, and demands a minimal storage period (e.g., 90 days). To utilize the storage effectively, we rely on two additional pieces of information provided with each file (in UFRs):

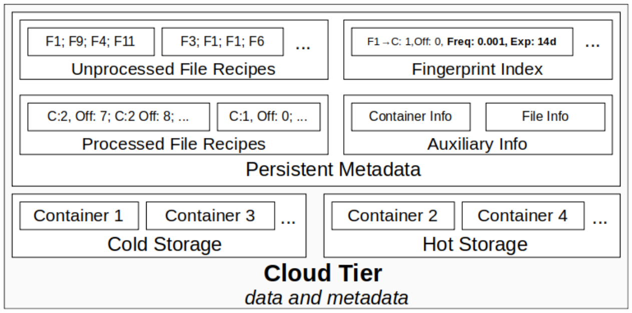


Figure 3: The architecture of data and metadata with two types of data storage (hot and cold). Fingerprint Index is extended.

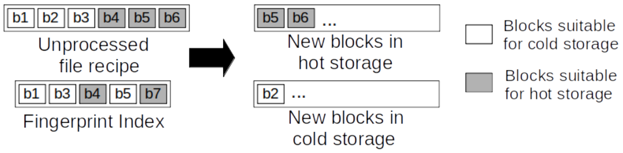

b1 b2 b3 b4 b5 b6
Unprocessed file recipe
b1 b3 b4 b5 b7
Fingerprint Index
b5 b6 ...
New blocks in hot storage
b2 ...
New blocks in cold storage
Blocks suitable for cold storage
Blocks suitable for hot storage

Figure 4: Writing blocks to more than one storage type. Block *b5* is written to hotter storage, although it is already available in colder storage if it brings a cost benefit (e.g., due to expected frequent restores of *b5*).

1. Current expiration date, as in *GC-Strategy #3*.

2. Rough, expected frequency of file restore.

As explained earlier, the expiration time is typically known. The restore frequency is unknown in advance, but assessing the read frequency of a file is common practice for data kept in the cloud. For instance, Amazon explicitly recommends different storage classes for data accessed "once per quarter" and "1-2 times per year" [8]. In the specific case of backups, assessing restore frequency should be possible, as a study of numerous backup jobs [16] suggests that backup domains fall into three categories: those with very frequent restores, sporadic restores, and virtually no restores. Moreover, particular backup policies influence the restore frequency [58], and an upper bound on the restores can be calculated based on restore SLAs. Finally, modern backup software already implements tools that allow viewing historical data on the restore frequency of selected resources [67].

The persistent data and metadata structures are organized as shown in Fig. 3. The process of container writing during BatchDedup and BatchGC is extended to store each block in an appropriate cloud storage type, as shown in Fig. 4. Each block is stored in a storage type for which the following formula has lower value:

$$t = COST_{insert} + (COST_{B/day} + COST_{restore} * FREQ_{restore}) * EXP_{time} \qquad (2)$$

In the formula, $COST_{insert}$ depends on cloud pricing, as well as the sizes of the block and its container, as the amortized cost of data insertion is included. $COST_{B/day}$ describes the storage cost of the block. $COST_{restore}$ depends on the data

[5] Theoretically, EXP_{time} can decrease if someone deletes a file before the expiration date. We find such a case rather marginal. In particular, enabling WORM protection [53] prevents such removals.

[6] Our algorithms can also work with the coldest storage services, which lengthens the retrieval process. However, in such case additional information are needed to specify the allowed retrieval time of each file.

locality, as many blocks can be read with one request, so the upper bound for the $COST_{restore}$ can be calculated as *one request per block* or assessed with a heuristic. $FREQ_{restore}$ and EXP_{time} are inherited from files referencing block, and stored with each block in FingIdx.

However, further adjustments to $FREQ_{restore}$ and EXP_{time} are required. That is because the first decision about storage type must be taken when the block is stored for the first time and block's $FREQ_{restore}$ and EXP_{time} are understated, as more references will come in the future. For instance, a block can be initially stored in cold storage but later it receives more references (and its $FREQ_{restore}$ increases). Vice versa, data with a short EXP_{time} can be kept in hot storage, although a reference with a more distant EXP_{time} will come soon.

Therefore, both $FREQ_{restore}$ and EXP_{time} should be heuristically modified. A heuristic that worked very well in our experiments relies on block reference counts. First, we select a number R of expected references for each block (e.g., a hard-coded value 5 or a value calculated from the system state). Then, we modify $FREQ_{restore}$ and EXP_{time} for blocks that have not reached the expected number based on the formula (e.g., we multiply it by $R - r$, where r is the actual number of references). In the end, $FREQ_{restore}$ and EXP_{time} for newly written blocks are more similar to their future values.

In justified cases, a block can be stored in multiple storage types (e.g., when a block stored in cold storage receives a reference with high $FREQ_{restore}$), but BatchGC will eventually remove the unnecessary copies. Similarly, BatchGC can move a block from one type of storage to another (e.g., when a reference with high restore frequency has been deleted). Generally, during BatchGC, a formula for calculating whether a container should be rewritten considers the potential cost reduction caused by a change of the storage type. A decision on whether rewriting a particular container is profitable must be made for the whole container because rewriting the container also introduces costs. Nevertheless, blocks from one container can be moved to containers in various storage types (Fig. 5).

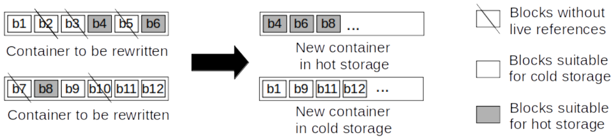

b1
b2
b3
b4
b5
b6
Container to be rewritten
b7
b8
b9
b10
b11
b12
Container to be rewritten
b4
b6
b8
...
New container in hot storage
b1
b9
b11
b12
...
New container in cold storage
Blocks without live references
Blocks suitable for cold storage
Blocks suitable for hot storage

Figure 5: Rewriting containers to multiple types of storage.

5 Evaluation

We present our experimental evaluation of InftyDedup in two parts. Firstly, we evaluate the performance and cost of our implementation executed in a public cloud. Secondly, we evaluate our strategies for garbage collection and storage type selection under various workloads.

5.1 Performance Evaluation

To evaluate the performance, we implemented InftyDedup using Apache Hive [24], which we selected as a possible approach to provide portability between different public and private clouds. We present results of the evaluation of our batch algorithms, as uploading containers and restoring data are straightforward object storage operations in which the bottleneck is expected mostly on the network to the cloud (even a naive implementation can saturate 1 GbE network with uploads and restores using a single core).

Our batch algorithms differ greatly from the state-of-the-art tiering to cloud with deduplication techniques, therefore a fair comparison with existing solutions was virtually impossible. Instead, we present the results using publicly available hardware. The evaluation was conducted in AWS using m5d.xlarge instances with 4x vCPU, 16 GiB of RAM, and 1x 150 GB NVMe (which costs less than network-attached EBS). We aimed to use the smallest possible instances (to maximize the horizontal scaling), but in our workloads, the technological stack of Apache Hive was inefficient in utilizing the limited memory of the smallest instances.

The presented experiments used synthetic data with the following characteristics. Each file contained approximately 51 GB (as backup files typically have tens of gigabytes or more [78]) chunked into blocks of approximately 64 KB (the target block size of the deduplication system for which we prepared InftyDedup). The contents of the files are described in each experiment. We present results with synthetically generated data, as our algorithms mostly distribute the data (e.g., based on fingerprints) and later sort the data in small portions, so the exact characteristic of the data (e.g., the initial order of blocks) does not affect the performance much.

5.1.1 Batch Deduplication Processing

We evaluated BatchDedup in configurations varying in size. Each experiment comprised two steps. In the first (initial) step, a large number of files without duplicates is processed to resemble a situation in which new backups are uploaded to the cloud. In the second (incremental) step, a dataset 3x smaller than the initial backup is uploaded (as typically incremental backups are smaller than their corresponding full backups [16]), where 90% of the blocks are duplicates (which matches the expected average daily deduplication ratio [16]). The smallest configuration (8 instances) uploads 3072 files in the first step and 1024 in the second step. In larger configurations, the amount of data to be processed is scaled linearly with the system size. Therefore, the smallest experiment processed metadata of 208 TB data and the largest one of 1.66 PB.

In all configurations, the first step takes between 1h53m and 2h10m (Fig. 6), and the second step takes up to 30m. Overall, the performance scales close to linearly. We analyzed the resource utilization, and the main bottleneck is the CPU, as most of the time its usage is above 95%. The network and

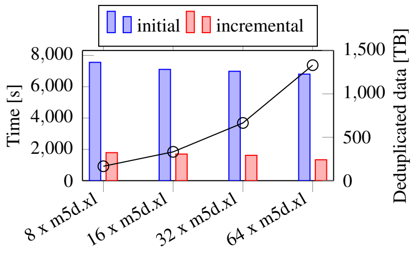


Figure 6: BatchDedup performance. The line and right y-axis show size of deduplicated data.

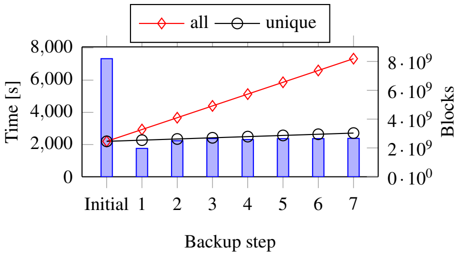


Figure 7: BatchDedup with growing data. The lines and right y-axis present number of blocks pre-deduplication (all) and post-deduplication (unique).

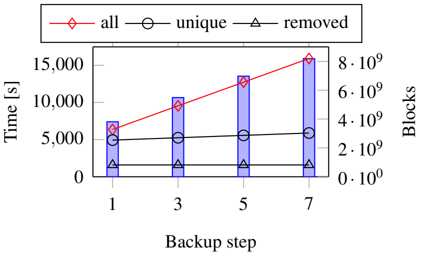


Figure 8: BatchGC performance. After 1-7 incremental steps, data from one incremental step was deleted (the removed blocks on right y-axis).

the local NVMe drive are underutilized, with peak per-node usage of respectively 350 MB/s of network bandwidth and 6% of disk utilization. We expect the computations can be further optimized, but the computation costs are already marginal compared to the cost of data storage. For instance, in the experiment with 32 instances, the second stage eliminates 191 TB of duplicates and costs below $1, which is less than 0.1% of monthly savings on storage. Similarly, the costs of accessing in-cloud metadata during processing are marginal, as both steps require roughly 250K GETs ($0.1) and 20K PUTs ($0.1), and transfer fees within one availability zone are free.

We also conducted a different experiment with multiple steps of incremental uploads in one configuration (8 instances). As shown in Fig. 7, the computation time increases close to linearly with the amount of non-duplicate data which is added to FingIdx in each experiment.

5.1.2 Batch Garbage Collection Processing

First, we evaluated BatchGC by removing a fraction of data uploaded in the first experiment from Section 5.1.1. Specifically, we removed the data uploaded in the first step to resemble removing the oldest backup. The processing took between 61 and 65 minutes in each verified configuration.

BatchGC, unlike BatchDedup, reads all PFRs, so we also verify that the processing time increases close to linearly with the size of both FingIdx and PFRs. The experiment shown in Fig. 8 had multiple incremental steps, and in BatchGC we removed data from one incremental step. The results confirm that, for data with many duplicates, BatchGC is more expensive than BatchDedup. However, BatchGC is expected to be executed less frequently, so both algorithms will have comparable total execution costs.

5.2 Strategies Evaluation

We evaluated how our garbage collection and storage type selection strategies behave in numerous workload simulations. The strategies optimize the costs of storing data for months and years, so we could not conduct these experiments in the public cloud, as it would take too long. Instead, we ran some initial experiments to confirm our understanding of the pricing model and features of the cloud, and based on the results, we implemented a simulator. The simulator calculates costs based on cloud pricing of storage, requests, transfer, and other factors, like the minimal storage duration.

Each experiment was conducted in many configurations of workload characteristics and system parameters. We present aggregated (minimal, maximal, and average) results, with values normalized to the result with the minimal cost.

5.2.1 Workload Characteristics

Our simulator allowed specifying the following factors to evaluate various backup workloads:

Data source was selected from the following two sets. Firstly, we generated synthetic workloads in which a given fraction of data was *modified* and *deleted* each day. Both types of modifications were applied in variable length *stream-contexts* (of size from 1 to 1024 blocks), so a given number of consecutive blocks was modified at once. The introduction of the stream-contexts was necessary, as data modified in small contexts are more fragmented, so reading data requires more requests. Secondly, FSL traces [73] were used, as they are real-world datasets that contain information on how the data of multiple users change over the years.

Retention policy specifies how long each file (backup) is stored. We analyzed guidelines related to retention policies [1,32,75] to generate realistic policies. Typically, each type of backup is stored for a longer time than its backup period (e.g., weekly backups are kept for four weeks). In our experiments, daily backups are kept for one week, weekly backups are kept for a month, monthly backups are kept for a year, and yearly backups are kept for five years. Based on that, we came up with three different policies: *keepAll* policy in which all types of backups are stored in the cloud, *dailyExcluded* in which daily backups are excluded (so only backups stored for at least a month are kept in the cloud), and *dailyOnly* in which only daily backups are kept in the cloud. The garbage collection was, in turn, executed every 7, 30, or 90 days. In all experiments, the simulation covered a period of 5 years.

Read patterns remarkably affect the total cost of ownership of data in the cloud. Unlike for writing data, we found no collected read traces for backup data. Similarly, there are no precise guidelines that describe typical backup read patterns. Therefore, we adapted a model in which each file is read with a given probability and verified the full spectrum of potential values.

5.2.2 Garbage Collection Strategies Evaluation

To evaluate how the proposed garbage collection strategies perform in different workloads, we conducted experiments with the pricing model of *AWS S3 Standard* as *hot* storage and *Glacier Instant Retrieval* as *cold* storage.[7] Experiments in which the storage types are mixed based on our strategy are denoted as *mixed*. Garbage collection strategies are denoted as follows: Strategy #1 is denoted as *onlyEmpty*, *less*{*25; 50; 75; 99*} denotes Strategy #2 with the T_{days} parameter such that the behavior is equivalent to reclaiming space when less than *25 / 50 / 75 / 99* percent of container capacity is used by live data, and Strategy #3 is denoted as *costBased*.

First, we evaluated a case in which there were no reads. As shown in Fig. 9, *onlyEmpty* strategy achieved the worst results. For *cold* and *mixed* storage, *costBased* strategy gave significantly better results than others (on average 1.4%-23%), whereas for hot storage (where the rewrite cost is marginal) it gave similar results to *less99*.

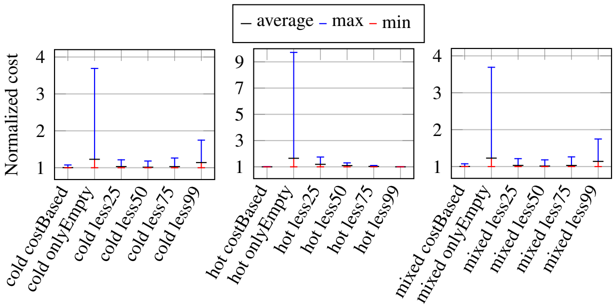


Figure 9: Garbage collection with different strategies.

In the next set of experiments, which included reads (with patterns explained in Section 5.2.3) and *mixed* storage, there are more differences between the strategies (Fig. 10). On average, *costBased* strategy is only 2.2% better, but comparing the worst cases, the difference is 24%. The analysis of the number of containers that are rewritten, deleted empty, or remain live at the end of the experiment, confirms that *onlyEmpty* has the largest number of containers that are live (Fig. 11).

The analysis of garbage collection strategies led to the question of how container sizes affect the costs, as smaller containers increase the probability of removing the entire container but also increase the number of PUT requests needed to store data initially or during container rewriting. As shown in Fig. 12, for *costBased* strategy, the lowest average cost is with 16 MB containers (4 MB and 64 MB are respectively 4.5% and 2% more expensive). The smallest, 1 MB containers were the most expensive, even with the *onlyEmpty* strategy, because of the cost of initial container generation (Fig. 13). Especially in *cold* storage, the cost of PUT requests is high (up to 40% of all costs with 1 MB containers).

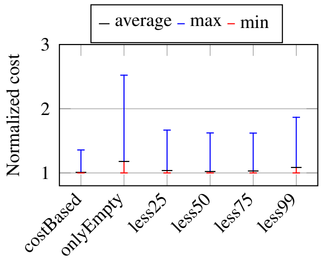


Figure 10: Garbage collection strategies with reads.

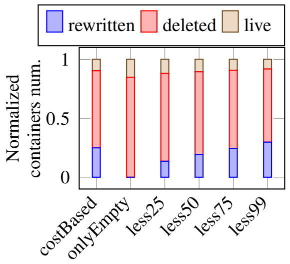


Figure 11: Breakdown of containers number.

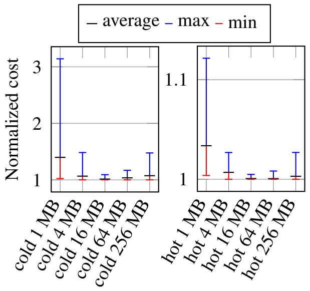


Figure 12: Garbage collection with varying container sizes.

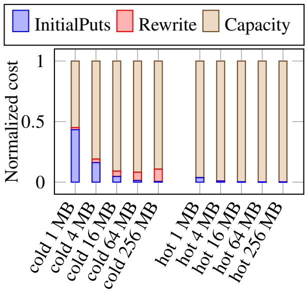


Figure 13: Cost breakdown with varying container sizes.

5.2.3 Storage Type Selection Evaluation

We evaluated our storage type selection strategies in workloads with varying read frequencies. For each experiment, there are 4 synthetically generated sets of files, and each set has a different read frequency: once a month, once a year, once a year with 1% probability, and once a year with 0.1% probability. All 4 sets were written together, just as in a storage system that keeps files with varying read frequencies. The experiments were conducted in series, and in each series, the read frequency was scaled by a factor from 0.001 to 10. Therefore, cases in which reads are virtually nonexistent, cases in which reads dominate the total cost, and cases in-between were evaluated. A real-world ratio between backup and recovery jobs is typically 100 : 1 [17] but varies depending on the system [16]. In our experiments, the ratio of backups to recov-

[7] At the moment of writing, cold storage had 4x/25x more expensive PUT/GET requests, 5.25x times cheaper storage costs, the minimum storage duration was 90 days, and an additional per-gigabyte retrieval cost for cold storage was equal to the fee for 3000 GET requests.

eries for scale factor 0.01 is $70 - 700 : 1$ (mean $= 216 : 1$) depending on the retention policy. Therefore, we expect results with scale factors 0.01 and 0.1 to reflect a typical use-case.

As shown in Fig. 14, on average *mixed* strategy gives 55% cost savings compared to *cold* if there are many reads and 70% compared to *hot* if there are hardly any reads. The breakdown of newly created containers (Fig. 15) confirms that data ends up in cold storage when there are hardly any reads, and in hot storage reads are frequent. The cost breakdown (Fig. 16) confirms that *mixed* strategy balances the high storage cost in *hot* and the expensive reads in *cold*.

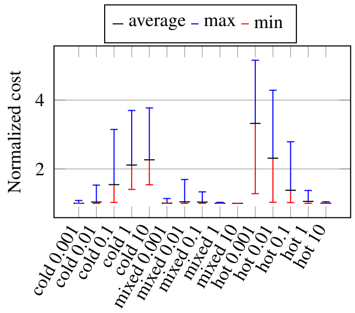


Figure 14: Storage type selection depending on the read frequency.

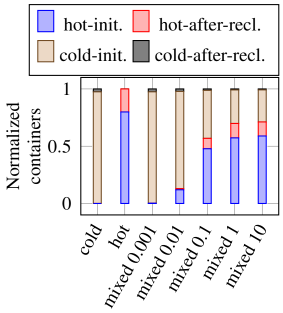


Figure 15: Containers created initially and after reclamation.

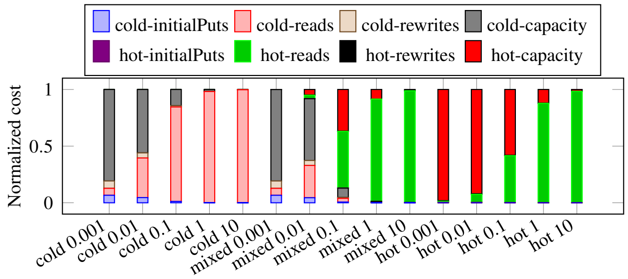


Figure 16: Cost breakdown with varying read frequencies.

We also evaluated how the changes in the predicted reference number affect the cost. Fig. 17 presents the normalized cost, depending on the selection of the expected reference number. Without predicting that more references will come in the future, the cost is higher on average by 11% (worst case 289%) compared to predicting 5-10 references, so we confirmed that predicting the number of references brings a significant cost reduction. The results with 3-10 references are very similar, so the slight inaccuracies in the expected number of references do not change the results much.

The mixed strategy depends on the expected frequency of reads, which may be incorrectly assessed. We conducted experiments with a significant prediction error (the value was underestimated and overestimated ten times). Even with such a large estimation error, the results are close to perfect (Fig. 18). Therefore, in all other experiments, we assumed perfect estimation to facilitate studying other experimental parameters.

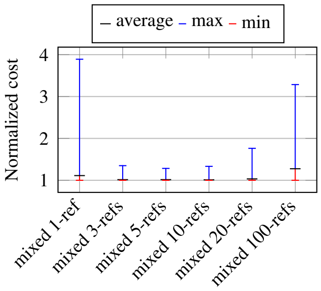


Figure 17: Cost of storing data depending on the expected number of references.

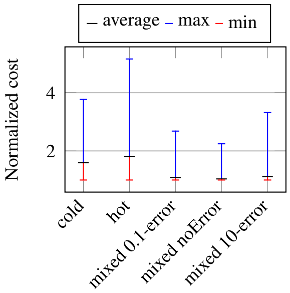


Figure 18: Cost of storing data depending on the error of frequency prediction.

5.2.4 Different Public Clouds

To confirm that our strategies are generally applicable to public clouds, we repeated most of the experiments with the pricing model of Google Cloud and Microsoft Azure. As our evaluation shows, mixing cold and hot storage reduces the costs for all three major providers (Fig. 19). The noticeable differences in gain between the cloud providers follow from the different ratios of costs, especially the cost of storing data and egress traffic. On average, keeping data only in hot storage is 61% more expensive, and keeping data only in cold storage is 30% more expensive than using the mixed strategy.

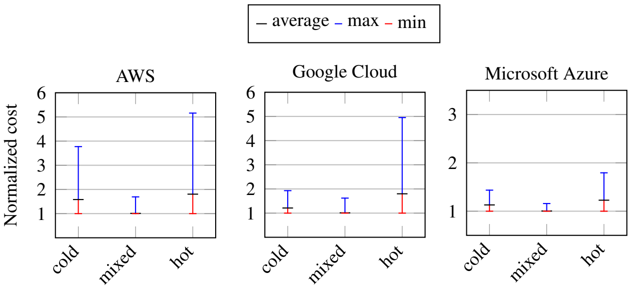


Figure 19: Storage type selection in different public clouds.

5.2.5 FSL Traces

Finally, we verified our strategies using the FSL traces [73]. Specifically, we used all data available with 64 KB chunking in *homes snapshots* dataset. The traces contain metadata of files chunked during writing, but they have no information about the read pattern. Therefore, for each user, we verified how our storage type selection works with a varying number of reads (restoring each backup with a frequency from 0.0001 to 1 time a month). As shown in Fig. 20, at the extreme read frequencies the mixed strategy keeps almost all the data in the cheapest of the two storage types. However, if the number of reads is in between, the mixed strategy works better than keeping data in a single type of storage, as depending on the data characterization a different decision should be made for each block. In particular, the characterization of the reference number of each block is important, as frequently referenced blocks are accessed more often. Therefore, mixing storage types can outperform keeping the data in one storage type, decreasing the cost by 26%-44%. This result shows that even when the restore frequency of each file is known in advance, relying on selecting one storage type can be significantly more expensive than using our mixed strategy.

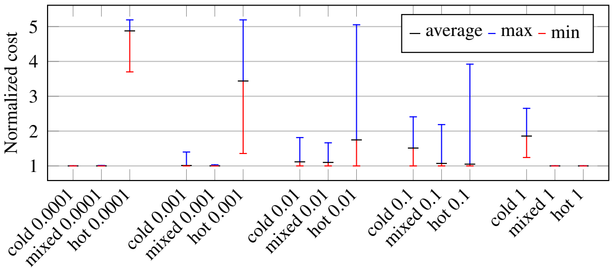


Figure 20: Total costs in experiments with FSL traces.

6 Related Work

Hierarchical storage is widely adapted, as storage devices offer a trade-off between cost, capacity, and performance [56]. Systems with storage tiers are actively researched, and in recent years, many publications have referred to tiering in the cloud [39, 46, 60]. Hsu et al. [38] propose an AI-based prediction model for classifying whether data is cold or hot. Liu et al. [45] describe an online algorithm for two-tier cloud storage, which works with no prior knowledge of future access frequencies. However, less attention is given to tiering techniques in the cloud in the context of deduplication.

MUSE [80] is a framework focused on providing SLA for deduplicated data focused on the primary storage use case, which is a different use case than storing backups. DD Tier [34] is tiering with deduplication that performs its computation in the local tier, hence imposing fundamental restrictions and limitations. First, deduplicating data from more than one local tier system is impossible, as each system performs deduplication on its own. Furthermore, all or at least a large fraction of metadata is needed locally to operate. Therefore, metadata are stored in both tiers, which not only increases storage capacity usage but also forces downloading a large amount of metadata to recover even a single file. Moreover, the resources for metadata storage and processing of the local tier are limited. As locally stored metadata can consume hundreds of terabytes of local storage, the size of the cloud tier is limited (to 2x the size of the local tier). Alike, deduplication and garbage collection algorithms cannot overuse scarce local resources, especially RAM, CPUs, and disk I/Os. Therefore, perfect hashing is used to decrease memory requirements below 3 bits per fingerprint, so extending it with techniques similar to our storage type selection is very difficult.

DD Tier introduces a technique for estimating how much space will be freed from the local tier after moving data to the cloud, and in recent years, significant research attention has been paid to the problem of selecting files for efficient data removal and migration in systems with deduplication [37, 42, 51]. As long as such methods do not require storing additional metadata locally, they can be used with InftyDedup.

A large number of publications explore security threats of deduplication in the cloud. Therefore, several methods of preventing particular attack types were proposed [21, 41, 43, 81]. Alike, side channels leaking information from deduplication storage have been studied [19, 20]. Most threats arise from the situation in which a public cloud provider implements deduplication between users. InftyDedup is meant to be used by a single organization, and writing to InftyDedup requires accessing the local tier, so the situation is much different. Still, some organizations might find the deduplication side-channels as a threat within the organization, and adding security mechanisms to InftyDedup can be required. Moreover, users of InftyDedup may not trust the cloud provider, so the local tier can encrypt data before storing them in the cloud. The structure of the data (information on block sizes and which blocks are referenced by which files) is still exposed to allow the computations, but the situation is similar in other tiering with deduplication solutions, as restoring blocks reveals the structure of files.

7 Conclusions

We presented InftyDedup, a novel, cloud-native approach to tiering to cloud for a storage system with deduplication. Compared to the state of the art, our architecture does not impose any limit on the size of the cloud tier and supports deduplication from multiple local tier systems. We implemented InftyDedup for a commercial storage system (HydraStor) and evaluated it in a public cloud (AWS). The evaluation confirmed the desired scalability of deduplication handling: our batch algorithms, designed to reduce cloud costs and harness dynamic resource allocation, were able to process metadata of multi-petabyte data collections for a couple of dollars.

To further decrease the cost of cloud storage, we proposed an extension to InftyDedup which moves chunks between hot and cold cloud stores based on their anticipated access patterns. Its evaluation with real-world traces showed that our deduplication-specific heuristic for adjusting the expected read frequency, which takes into account block reference counts, decreased the costs on average by 11%, and the overall solution achieved 26%–44% reductions. The algorithm requires minimal input from a system administrator and was demonstrated to retain its cost benefits even when the administrator's estimations were inexact.

Acknowledgments

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

References

[1] Acronis. Retention rules: how and when they work. 2022. https://kb.acronis.com/content/68304.

[2] 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.

[3] Yamini Allu, Fred Douglis, Mahesh Kamat, Philip Shilane, Hugo Patterson, and Ben Zhu. Backup to the future: How workload and hardware changes continually redefine data domain file systems. *Computer*, 50(7):64–72, 2017.

[4] Amazon Web Services, Inc. Amazon aurora - fully mysql and postgresql compatibile managed database service. 2023. https://aws.amazon.com/rds/aurora/?did=ap_card&trk=ap_card.

[5] Amazon Web Services, Inc. Amazon ebs pricing. 2023. https://aws.amazon.com/ebs/pricing/.

[6] Amazon Web Services, Inc. Amazon elastic container service pricing. 2023. https://aws.amazon.com/ecs/pricing/.

[7] Amazon Web Services, Inc. Amazon s3 pricing. 2023. https://aws.amazon.com/s3/pricing/.

[8] Amazon Web Services, Inc. Amazon s3 storage classes. 2023. https://aws.amazon.com/s3/storage-classes/.

[9] Amazon Web Services, Inc. Aws direct connect locations. 2023. https://aws.amazon.com/directconnect/locations/.

[10] Amazon Web Services, Inc. Aws lambda - faqs. 2023. https://aws.amazon.com/lambda/faqs/.

[11] Amazon Web Services, Inc. Aws snow family faqs. 2023. https://aws.amazon.com/snow/faqs/.

[12] Amazon Web Services, Inc. Best practices for cluster configuration. 2023. https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-instances-guidelines.html.

[13] Amazon Web Services, Inc. Billing for interrupted spot instances. 2023. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/billing-for-interrupted-spot-instances.html.

[14] Amazon Web Services, Inc. General purpose instances - network performance. 2023. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/general-purpose-instances.html#general-purpose-network-performance.

[15] Amazon Web Services, Inc. Link aggregation groups. 2023. https://docs.aws.amazon.com/directconnect/latest/UserGuide/lags.html.

[16] George Amvrosiadis and Medha Bhadkamkar. Identifying trends in enterprise data protection systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015.

[17] George Amvrosiadis and Medha Bhadkamkar. Getting back up: Understanding how enterprise data backups fail. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[18] Associate Research Director Andrew Smith, Research Manager; Archana Venkatraman. Enterprise data growth and adoption of cloud applications challenge traditional data protection strategies. 2021. https://afi.ai/r/US48310921.pdf.

[19] Frederik Armknecht, Colin Boyd, Gareth T Davies, Kristian Gjøsteen, and Mohsen Toorani. Side channels in deduplication: Trade-offs between leakage and efficiency. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.

[20] Andrei Bacs, Saidgani Musaev, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Dupefs: Leaking data over the network with filesystem deduplication side channels. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022.

[21] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In *Annual international conference on the theory and applications of cryptographic techniques*, 2013.

[22] Christophe Bertrand. Esg research report: The evolution of data protection cloud strategies. 2021. https://www.esg-global.com/research/esg-research-report-the-evolution-of-data-protection-cloud-strategies.

[23] Sriprasad Bhata. Introducing azure cool blob storage. Microsoft, 2016. https://azure.microsoft.com/en-us/blog/introducing-azure-cool-storage/.

[24] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, et al. Apache hive: From mapreduce to enterprise-grade big data warehousing. In *Proceedings of the 2019 International Conference on Management of Data*, 2019.

[25] Google Cloud. Cloud sql. 2023. https://cloud.google.com/sql.

[26] Google Cloud. Cloud storage pricing. 2023. https://cloud.google.com/storage/pricing#north-america.

[27] Google Cloud. Disk pricing. 2023. https://cloud.google.com/compute/disks-image-pricing#disk.

[28] Google Cloud. Storage classes. 2023. https://cloud.google.com/storage/docs/storage-classes#descriptions.

[29] Google Cloud. Vertex ai - google cloud's unified ml platform. 2023. https://cloud.google.com/vertex-ai.

[30] Biplob Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[31] 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.

[32] Druva. What is backup retention policy? how is it implemented? 2023. https://docs.druva.com/Knowledge_Base/inSync/Client/010_FAQ/What_is_Backup_Retention_Policy%3F_How_is_it_implemented%3F.

[33] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. Hydrastor: A scalable secondary storage. In *Proccedings of the 7th Conference on File and Storage Technologies (FAST 09)*, 2009.

[34] 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.

[35] Nnamdi Ekwe-Ekwe and Adam Barker. Location, location, location: exploring amazon ec2 spot instance pricing across geographical regions. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018.

[36] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[37] 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.

[38] Ying-Feng Hsu, Ryo Irie, Shuuichirou Murata, and Morito Matsuoka. A novel automated cloud storage tiering system through hot-cold data classification. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.

[39] Ryo Irie, Shuuichirou Murata, Ying-Feng Hsu, and Morito Matsuoka. A novel automated tiered storage architecture for achieving both cost saving and qoe. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, 2018.

[40] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference*, 2012.

[41] Sriram Keelveedhi, Mihir Bellare, and Thomas Ristenpart. Dupless: Server-aided encryption for deduplicated storage. In *22nd USENIX Security Symposium (USENIX Security 13)*, 2013.

[42] 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.

[43] Jingwei Li, Chuan Qin, Patrick P. C. Lee, and Jin Li. Rekeying for encrypted deduplication storage. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[44] 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.

[45] Mingyu Liu, Li Pan, and Shijun Liu. To transfer or not: An online cost optimization algorithm for using two-tier storage-as-a-service clouds. *IEEE Access*, 7:94263–94275, 2019.

[46] Yaser Mansouri and Abdelkarim Erradi. Cost optimization algorithms for hot and cool tiers cloud storage services. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.

[47] Dirk Meister and André Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009.

[48] Dirk Meister and André Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (ssd). In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[49] Microsoft. Azure storage pricing. 2023. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/#pricing.

[50] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2010.

[51] 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.

[52] Netapp. Cloud tiering documentation. 2023. https://docs.netapp.com/us-en/cloud-manager-tiering/pdfs/fullsite-sidebar/Cloud_Tiering_documentation.pdf.

[53] Veritas NetBackup. About netbackup worm storage support for immutable and indelible data. 2020. https://www.veritas.com/support/en_US/doc/25074086-143197427-0/v143250065-143197427.

[54] Veritas NetBackup. Veritas netbackup™ deduplication guide. 2021. https://www.veritas.com/support/en_US/doc/25074086-146020141-0/v145698641-146020141.

[55] Veritas NetBackup. Aws cloud storage with veritas netbackup. 2022. https://www.veritas.com/content/dam/www/en_us/documents/white-papers/WP_aws_cloud_storage_with_netbackup_long_term_retention_solution_V1259.pdf.

[56] Junpeng Niu, Jun Xu, and Lihua Xie. Hybrid storage systems: A survey of architectures and algorithms. *IEEE Access*, 6:13385–13406, 2018.

[57] Myoungwon Oh et al. Design of global data deduplication for a scale-out distributed storage system. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018.

[58] Oracle. Backing up file-system data. 2023. https://docs.oracle.com/cd/E91325_01/OBADM/osb_filesystem_backup.htm.

[59] Michael Paul. Cloud object storage deep dive – part two, implementation. Veeam Software, 2021. https://www.veeam.com/blog/cloud-object-storage-implementation.html.

[60] Ajaykrishna Raghavan, Abhishek Chandra, and Jon B Weissman. Tiera: Towards flexible multi-tiered cloud storage instances. In *Proceedings of the 15th International Middleware Conference*, 2014.

[61] Santhosh Rao, Nik Simpson, Michael Hoeck, and Jerry Rozeman. Magic quadrant for enterprise backup and recovery software solutions. 2021. https://www.gartner.com/en/documents/4003661.

[62] Meet Shah, Mohammedhasan Shaikh, Vishwajeet Mishra, and Grinal Tuscano. Decentralized cloud storage using blockchain. In *2020 4th International conference on trends in electronics and informatics (ICOEI)(48184)*, 2020.

[63] Pratima Sharma, Rajni Jindal, and Malaya Dutta Borah. Blockchain-based decentralized architecture for cloud storage system. *Journal of Information Security and Applications*, 62:102970, 2021.

[64] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[65] Veeam Software. Step 7. configure long-term retention. 2021. https://helpcenter.veeam.com/docs/backup/vsphere/backup_job_gfs_vm.html?ver=110.

[66] Veeam Software. 2022 data protection trends. 2022. https://go.veeam.com/wp-data-protection-trends-2022.

[67] Veeam Software. Restore operator activity. 2023. https://helpcenter.veeam.com/docs/one/reporter/restore_operator_activity.html?ver=110.

[68] Hewlett Packard Enterprise storage experts. Hpe cloud bank storage: A data protection solution you can bank on. 2017. https://community.hpe.com/t5/Around-the-Storage-Block/HPE-Cloud-Bank-

Storage-A-Data-Protection-Solution-You-Can-Bank/ba-p/6965903.

[69] Przemyslaw Strzelczak, Elzbieta Adamczyk, Urszula Herman-Izycka, Jakub Sakowicz, Lukasz Slusarczyk, Jaroslaw Wrona, and Cezary Dubnicki. Concurrent deletion in a distributed content-addressable storage system with global deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[70] Zhen Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, et al. A long-term user-centric analysis of deduplication patterns. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, 2016.

[71] Zhen "Jason" Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, and Erez Zadok. Cluster and single-node analysis of long-term deduplication patterns. *ACM Transactions on Storage (TOS)*, 14(2):1–27, 2018.

[72] Liyin Tang and Namit Jain. Join strategies in hive. *Hive Summit*, 2011.

[73] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[74] Lionel Sujay Vailshery. Cloud infrastructure services vendor market share worldwide from 4th quarter 2017 to 4th quarter 2021. 2022. https://www.statista.com/statistics/967365/worldwide-cloud-infrastructure-services-market-share-vendor/.

[75] Global Data Vault. Data backup: Developing an effective data retention. 2023. https://www.globaldatavault.com/blog/data-retention-policy-and-scheduled-backups/.

[76] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013.

[77] Marcia Villalba. Amazon s3 glacier is the best place to archive your data – introducing the s3 glacier instant retrieval storage class. Amazon Web Services, Inc., 2021. https://aws.amazon.com/blogs/aws/amazon-s3-glacier-is-the-best-place-to-archive-your-data-introducing-the-s3-glacier-instant-retrieval-storage-class/.

[78] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[79] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[80] Jianwei Yin, Yan Tang, Shuiguang Deng, Bangpeng Zheng, and Albert Y. Zomaya. Muse: A multi-tierd and sla-driven deduplication framework for cloud storage systems. *IEEE Transactions on Computers*, 70(5):759–774, 2021.

[81] Haoran Yuan, Xiaofeng Chen, Jin Li, Tao Jiang, Jianfeng Wang, and Robert H Deng. Secure cloud data deduplication with efficient re-encryption. *IEEE Transactions on Services Computing*, 15(1):442–456, 2019.

[82] Zerto. Maximize recovery achieve your best rtos and rpos. 2020. https://www.zerto.com/wp-content/uploads/2020/08/Fastest-RTO-and-RPO-in-the-Industry_Guide.pdf.

[83] Zerto. Deploy & configure zerto long-term retention amazon s3. 2022. https://www.zerto.com/page/deploy-configure-zerto-long-term-retention-amazon-s3/.

[84] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 08)*, 2008.

PERSEUS: A Fail-Slow Detection Framework for Cloud Storage Systems

Ruiming Lu[1]*, Erci Xu[3,1]*, Yiming Zhang[2]†, Fengyi Zhu[3], Zhaosheng Zhu[3],
Mengtian Wang[3], Zongpeng Zhu[3], Guangtao Xue[1]†, Jiwu Shu[2], Minglu Li[1,4], and Jiesheng Wu[3]

[1]Shanghai Jiao Tong University, [2]Xiamen University,
[3]Alibaba Inc., and [4]Zhejiang Normal University

Abstract

The newly-emerging "fail-slow" failures plague both software and hardware where the victim components are still functioning yet with degraded performance. To address this problem, this paper presents PERSEUS, a practical fail-slow detection framework for storage devices. PERSEUS leverages a light regression-based model to fast pinpoint and analyze fail-slow failures at the granularity of drives. Within a 10-month close monitoring on 248K drives, PERSEUS managed to find 304 fail-slow cases. Isolating them can reduce the (node-level) 99.99$^{\text{th}}$ tail latency by 48%. We assemble a large-scale fail-slow dataset (including 41K normal drives and 315 verified fail-slow drives) from our production traces, based on which we provide root cause analysis on fail-slow drives covering a variety of ill-implemented scheduling, hardware defects, and environmental factors. We have released the dataset to the public for fail-slow study.

1 Introduction

Large-scale storage systems are susceptible to various failures [2, 3, 5, 7, 21, 32, 34, 35, 41, 42, 48]. Both academia and industry have made great efforts on identifying [26, 28, 39], detecting [9, 10, 20, 27], and fixing [8, 13, 44, 46] different kinds of failures (e.g., fail-stop [2, 18, 31, 37], fail-partial [6, 38, 40], and Byzantine [14]) in the field.

Recently, the fail-slow failures [19], also known as gray failures [24] or limpware [16], have been receiving an increasing amount of attention [23, 29, 36]. In fail-slow failures, a software or hardware component (while functioning) delivers lower-than-expected performance. With faster hardware devices (e.g., Optane SSD [49] and Z-SSD [11]) and software stack (e.g., kernel bypassing [47]), the impact of fail-slow failures (e.g., caused by malfunctioning NANDs or unfit scheduling), which might be masked as noise previously, are more likely to be noticed. Recent studies [30, 36] indicate that annual fail-slow occurrences can be as frequent as annual fail-stop events (1%∼2%).

Accurately detecting fail-slow failures is challenging. Performance variations caused by internal factors (e.g., SSD garbage collection) or external factors (e.g., workload burst) can have similar symptoms as fail-slow failures. Unlike fail-stop failures where the criteria (e.g., software crash [2], data loss [34]) are well-defined, determining fail-slow failures is usually empirical in practice and thus inherently inaccurate. Moreover, fail-slow failures are often transient [19], making it difficult for on-site engineers to identify, let alone reproduce or reason the root causes.

Although several work [23, 29, 36] has attempted to detect fail-slow failures, they are impractical and inefficient for large-scale deployment in our production cloud environment. First, these techniques require source code access (e.g., static analysis in OmegaGen [29]) or software modification (e.g., modifying software timeouts in IASO [36]), while cloud vendors like us do not touch tenants' code. Even for in-house infrastructures, inserting certain code segments is still time-consuming, as the systems can run dozens of internal services with different software stacks. Second, existing techniques can only detect fail-slow failures at the node level (e.g., IASO), thus still requiring nontrivial manual efforts to locate the culprits [19].

In this paper, we share our experiences in developing a practical, fine-grained, and general fail-slow detection framework that is applicable to a wide range of of services and devices (with minor or no adjustment) in the Alibaba cloud data centers. We start with analyzing the characteristics of the known fail-slow failures in our fleet. We then discuss three unsuccessful attempts at identifying fail-slow failures in the field, including using an empirical threshold, performing peer-evaluation-based detection [22], and refactoring IASO [36].

With the lessons learned from our earlier efforts, we design and implement PERSEUS, a non-intrusive fail-slow detection framework. We first leverage classic machine learning techniques (PCA [1], DBSCAN [43], and polynomial regression [17]) to establish a mapping between latency variation and workload pressure. With the mapping, PERSEUS can automatically derive an accurate and adaptive threshold for each node to identify slow entries within the monitoring traces. Further, based on the slow entries, PERSEUS constructs the corresponding fail-slow events and utilizes a scoreboard mechanism to evaluate the severity of such events.

PERSEUS has been deployed in our cloud for over ten months, monitoring an increasing number of drives up to around 300K by now. PERSEUS has already identified more

*Equal contribution.

†Corresponding authors.

than 300 fail-slow drives. By isolating and/or replacing the identified fail-slow drives, we significantly reduce the node-level tail latency. The 95th, 99th, and 99.99th write latencies drop by 31%, 46%, and 48%, respectively.

We compare PERSEUS to previous fail-slow detection methods as follows. We assemble a large-scale fail-slow dataset (including 315 verified fail-slow drives and around 41K of their cluster-wise peer drives) from our production traces, and build a test benchmark based on the dataset. The benchmark evaluations indicate that PERSEUS outperforms all previous methods, achieving a precision of 0.99 and a recall of 1.00. We also evaluate the effectiveness of components and the sensitivity of parameters in PERSEUS. The results show that PERSEUS can serve as a non-intrusive (based on monitoring traces), fine-grained (per-drive), general (one set of parameters fits all) and accurate (high precision and recall) fail-slow detection framework for the cloud storage systems.

We have also analyzed the reasons for fail-slow failures and discover a wide variety of root causes including ill-implemented scheduling (e.g., unnecessary resource contention), hardware flaws (e.g., bad sectors for HDDs), and environmental factors (e.g., temperature and power).

This paper makes the following contributions.

- We share our lessons on detecting fail-slow failures in large-scale data centers from three unsuccessful attempts.
- We propose the design of PERSEUS, a non-intrusive, fine-grained and general fail-slow detection framework.
- We assemble a large-scale fail-slow dataset[1] and build a fail-slow test benchmark.
- We provide an in-depth root cause analysis of fail-slow failures from the perspective of various factors.

2 Background

2.1 System Architecture

In this paper, we explore fail-slow detection methods on a subset of Alibaba Internet Data Centers (IDCs). These IDCs span across the globe and each IDC includes multiple storage clusters. Atop each cluster, a distributed file system (DFS) is deployed to support a dedicated service (e.g., block storage, NoSQL, or big data analysis). Each cluster consists of tens of racks (at most 200), and each rack contains dozens of nodes (at most 48). There are three types of storage nodes: (1) *All-flash*: a node contains 12 NVMe SSDs to store data; (2) *Hybrid*: a node contains 60-120 HDDs for data storage and 2 SSDs as write cache; (3) *All-HDD*: a node contains 70-80 HDDs to store data. By default, data storage drives in each node are of the same model. At most three drives from the same node can be taken down for repairing at a time. Table 1 lists the basic information and distribution of the drives. Note that we name drive models as vendor-model. For example, I-A stands for model A of vendor I.

Class	Model	Vendor%	Total%	Layer /Type	Cap. (GB)
NVMe SSD	I-A	3.58	1.80	32L	1920
	I-B	6.96	3.49	64L	1920
	I-C	0.76	0.38	32L	3840
	I-D	82.57	41.38	64L	4000
	I-E	6.13	3.07	64L	7680
	II-A	52.24	2.40	48L	1920
	II-B	47.76	2.19	48L	3840
SATA HDD	III-A	100	13.28	CMR	12000
	IV-A	100	32.01	CMR	12000

Table 1: Summary of drive statistics in our dataset. (§2.1). ***Vendor%****: percentage of drive models in the same vendor;* ***Total%****: percentage of drive models in the total population;* ***Layer/Type****: number of stacking layers for 3D TLC NAND SSD, or recording type for HDD;* ***Cap.****: capacity.*

Service	#Entries (M)	Total%
Log service	0.58	0.16
Big data	2.05	0.57
E-commerce	4.04	1.13
Table storage	9.32	2.61
Stream processing	12.77	3.58
Database	13.61	3.81
Object storage	30.13	8.44
Data warehouse	31.86	8.92
Block storage	252.80	70.78
Total	**357.16**	**100.00**

Table 2: Cloud services and daily entries (§2.2). ***#Entries (M)****: number of entries in millions. Each entry has five fields (i.e., avg_latency, avg_throughput, drive_ID, node_UID, timestamp).*

2.2 Dataset Description

In our data centers, a monitoring daemon is placed in each node to collect operational statistics, mainly the latency and throughput of each drive. The daemons calculate the average statistics every 15 seconds and record them as time-series data entries. The daemons run three hours a day (from 9PM to 12AM). A drive generates 720 entries (= 180 min × 4 entries/min) per day. In total, we have compiled around 100 billion entries as our dataset. Table 2 lists the daily distribution of 9 cloud services.

2.3 Impact of Fail-Slow Failures

Fail-slow failures, especially the transient ones, can often be ignored or misinterpreted as performance variations. Additionally, storage stacks usually have multiple levels (software, firmware, and hardware) of fault tolerance, such as retry and redundancy, silently masking the fail-slow failures. However, fail-slow failures can have a much more significant impact on I/O performance in the wild. Next, we will use a representative example from our object storage service to demonstrate

[1] We release our dataset at https://tianchi.aliyun.com/dataset/144479.

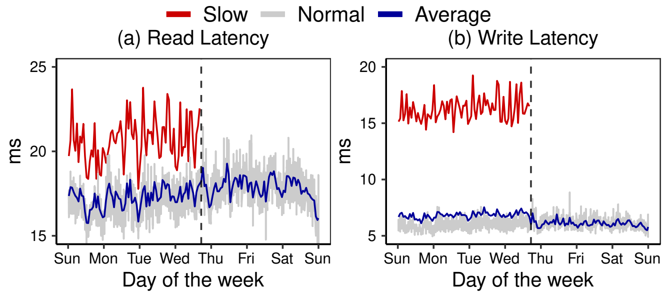


Figure 1: Fail-slow impact on I/O latency (§2.3). *The figures show (a) read and (b) write latency three days before and after isolating the fail-slow HDD (in red) in one node. Lines in grey refer to the latency distribution of normal peers from the same node. The vertical dashed line refers to the time when the fail-slow drive was taken down for replacement.*

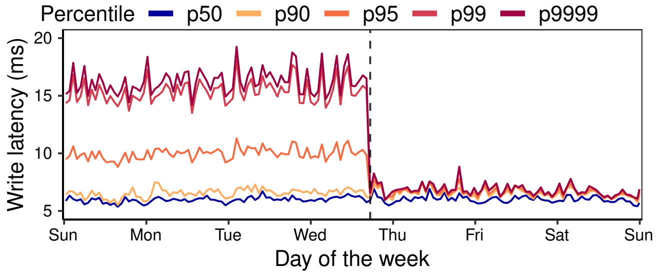


Figure 2: Fail-slow impact on tail latency (§2.3). *The figure shows the node-level write latency (of the same node as in Figure 1) at different percentiles. The vertical dashed line refers to the isolation time.*

the effect of fail-slow failures and consequently the importance of fail-slow failure detection.

Figures 1a and 1b show the read and write latency of a fail-slow HDD (the red line) against the latency of the peer HDDs (the gray lines) from the same node. The vertical dashed line indicates when the fail-slow drive was isolated (and taken down for replacement). While the fail-slow drive shows much higher read/write latency than the mean (the blue line) of all the drives (2.06∼3.65× higher for write and 1.01∼1.50× higher for read), its other metrics, such as IOPS, remain normal.

It was the utilization rate rather than the latency that directly led us to identify this fail-slow failure. Our monitoring system indicated that the victim node had been receiving much fewer writes than other nodes. The log analysis further suggested that the load balancer of the distributed object storage system prefers not to allocate writes to the victim node due to its abnormally high retry rate caused by high tail latencies. Figure 2 presents the 50th, 90th, 95th, 99th, and 99.99th percentile latency of the victim node before and after the dashed line when the fail-slow drive was isolated. Before isolating the fail-slow drive, the 99.99th tail latency is 2.43∼3.29× that of the median at the node level.

This example, along with other similar cases, shows that fail-slow failures impact not only the victim drive but also the entire node for a long period of time. This motivates us to explore effective measures for detecting fail-slow failures in our cloud.

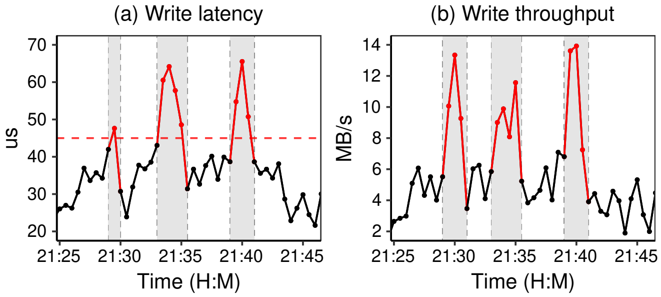


Figure 3: Sudden latency increase due to heavy load (§3.2). *The figures present the time series of (a) write latency and (b) write throughput of an NVMe SSD. Red points in grey boxes refer to time segments where drive latency is higher than a naive alarming threshold (i.e., 45 µs).*

3 Unsuccessful Attempts & Lessons

In this section, we first describe the design goals of the detection framework, and then discuss three unsuccessful early attempts. We conclude this section with a series of lessons to guide the design of PERSEUS.

3.1 Design Goals

From our perspective, a practical fail-slow detection framework should have the following properties.

- *Non-intrusive.* As cloud vendors, neither can we alter users' software nor require them to run specific modified versions of software stacks. Therefore, we can only rely on external performance statistics (e.g., drive latency) for detection.
- *Fine-grained.* Fail-slow root cause diagnosis can often be time-consuming (e.g., days to even weeks [19, 36]). We expect the framework to pinpoint the culprit.
- *Accurate.* The framework should have satisfying precision and recall to avoid unnecessary diagnosis on false positives.
- *General.* The framework can be deployed on both SSD and HDD clusters and quickly applied to different services (e.g., block/object storage and database) with minor adjustments.

3.2 Attempt 1: Threshold Filtering

Methodology. Intuitively, we can set up a hard threshold on drive latency to identify fail-slow drives based on the Service Level Objectives (SLOs). To avoid mislabeling due to one-off events such as SSD internal GC, we further specify a minimum slowdown span for a suspicious drive to be considered as fail-slow.

Limitation. Enforcing a hard threshold on device latency is clearly non-intrusive and fine-grained. However, the accuracy of threshold-identified fail-slow is low, as the latency is highly influenced by the workloads. Here, we use the latency and throughput traces of an NVMe SSD from the block storage service as an example. The left of Figure 3 illustrates the

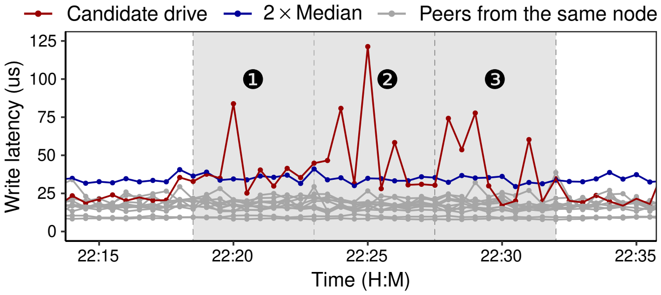


Figure 4: Peer evaluation (§3.3). *The figure shows peer evaluation results by comparing the candidate drive latency (in red) with node-level median latency (in blue) using a 5-minute sliding window. The candidate is slow (> 50% latency records higher than 2×Median) in three time windows (labeled with numbers in grey boxes).*

latency variation of the drive where the horizontal dashed line indicates the threshold (45μs). The right of Figure 3 is the corresponding throughput. We can see that three latency spikes occur at around 21:29, 21:34, and 21:40 as the latency increases up to 65μs. By comparing latency with throughput, it is clear that the workload pressure causes these spikes.

Hence, the dilemma is as follows. Setting a relaxed threshold easily mislabels normal performance variations as fail-slow events. Meanwhile, a strict one could leave many fail-slow cases undiscovered. Further, using a set of thresholds for different scenarios through fine-tuning can be fairly time-consuming as our experiments show that latency variation is a factor of drive models and workloads. In practice, we use threshold-based detection as a fail-safe measure like timeout.

3.3 Attempt 2: Peer Evaluation

Methodology. The problem of the first attempt is not having an adaptive threshold for detection. To address this problem, we explored the idea of peer evaluation [22,30]. The rationale behind this approach is that, with load balancing across the distributed storage system, drives from the same node should receive similar workload pressure. Since fail-slow failures are relatively rare [19] and the majority of drives in a node should be normal, we can identify the fail-slow drive by comparing the performance between drives from the same node.

Specifically, we first calculate the node-level median latency at each entry timestamp (every 15 seconds). We then evaluate whether there are drives constantly (more than half of the time) delivering abnormal performance—twice slower than the median in our case—during the monitoring window (e.g., 5 minutes). If so, the detection framework reports a fail-slow event, and the monitoring window moves forward to start the next round of evaluation.

Figure 4 provides an example of peer evaluation detection, including the fail-slow drive (the red line), the adaptive threshold (2×median, the blue line), and the normal drives (grey lines). The three shaded regions (numbered ❶, ❷, and ❸) indicate three monitoring windows when the victim drive experiences a fail-slow event.

Limitation. Peer evaluation can obtain an adaptive threshold (the blue line in Figure 4), but it requires more empirical parameters than threshold filtering, such as the slowdown degree and the monitoring window span, for tuning. Although it is possible to fine-tune the parameters on a few clusters for specific storage services, the effort would be prohibitively large if we want to extend peer evaluation to other drive models of different services. For example, it took on-site engineers two hours to fine-tune a cluster with around 300 nodes, and this set of parameters fails to work on another cluster even under the same service with the exact same models of drives.

3.4 Attempt 3: IASO-Based Model

Methodology. IASO is a fail-slow detection framework focusing on identifying performance-degrading nodes [36]). The design principle of IASO is to leverage software timeouts and convert them into informative metrics to benchmark fail-slow. However, directly using IASO is not suitable for us. First, IASO requires code changes (i.e., intrusive monitoring) to insert or modify certain code snippets of the running instances (e.g., Cassandra and ZooKeeper), thus leveraging software-level timeouts to identify fail-slow incidents[2]. Second, IASO is node-level detection, whereas our goal is device-level. Nevertheless, we re-factor IASO with our best effort. To avoid modifying the software, we reuse the fail-slow event reporting by peer evaluation (i.e., Attempt 2, see §3.3).

Limitation. The IASO-based model delivers rather unsatisfactory performance (see §5.4 for details) on our assembled benchmark, with a precision rate of only 0.48. We suspect the main reason is that using the fail-slow event reporting to replace the software timeout might not be effective. Moreover, we have explored other possible alternatives, such as replacing software timeouts with thresholds. However, the results are still unsatisfactory. Therefore, we assume IASO, even with refactoring, may not achieve our goals.

3.5 Guidelines for PERSEUS

The aforementioned methods are either labor-intensive (requiring extensive tuning) or ineffective in the field. In this subsection, we use a series of research questions as guidelines for designing our next fail-slow detection framework.

RQ1: What metrics should we use?

Throughout the early development of previous attempts, we mostly focused on the write performance (i.e., the latency/throughput of write) for two reasons. First, among the verified fail-slow cases, more than half of them only have a notable influence on writes. Even for the rest, the impact on read performance is always much smaller (similar to Figure 1). Second, fail-slow failures have more severe impacts on writes. In our storage systems, most clusters require the

[2] Note that the definition of "non-intrusive" is different in IASO (meaning low overhead introduced) from ours (meaning no code changes).

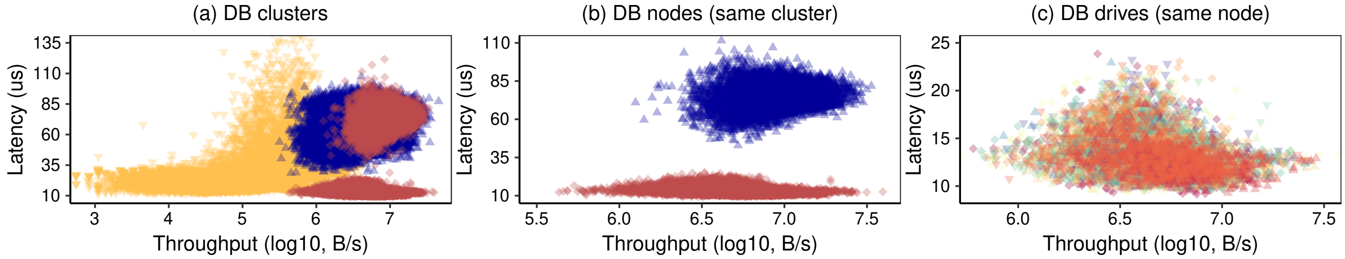


Figure 5: Distinct LvT distribution (§3.5). *The figures show the latency-vs-throughput (LvT) distribution of (a) three clusters (in yellow, blue and red) from database service, (b) two representative nodes (in blue and red) with clear-cut distribution from the same cluster (also from database service), and (c) drives (in distinct colors) in one node from database service. Throughput (unit: B/s) is scaled by log base 10 here and in related figures hereafter.*

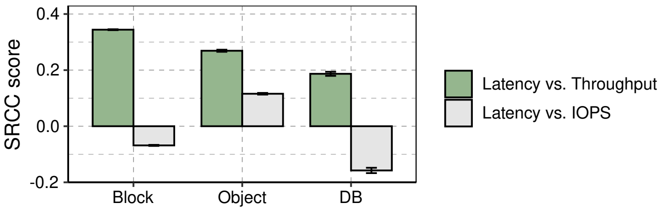


Figure 6: SRCC scores (§3.5). *The figure shows the SRCC scores between drive write latency and throughput/IOPS from three major services. Error bars refer to 95th percentile confidence intervals.*

write request to return after all three replicas ACKed (only <20% of our clusters require 2 replicas). Meanwhile, a read request will return as soon as one replica returns. In this case, fail-slow failure impacts the write request more often as one fail-slow write can lead to a slow write request in most cases while only 1/3 of the chances for a read request. Note that write reallocation cannot remedy for fail-slow write requests as the reallocation is triggered by a timeout (usually much longer than a fail-slow event). Nevertheless, we still evaluate fail-slow detection based on the read performance (see §5.5), and the results confirm the above assumptions.

RQ2: How to model workload pressure?

Simply depending on latency to detect fail-slow failures is unreliable. Our previous exploration has shown that workload pressure can significantly influence the latency variation. Further, we explore other indexes to model workload pressure to better understand the latency variation.

Analyzing Figure 3a and Figure 3b inspires us to check the possibility of using throughput or IOPS to model workload. Currently, per-drive I/O throughput (unit: byte/sec) and IOPS (unit: count/sec) are both available and stored in the same fashion as drive latency (see Section 2.2). In Figure 6, we measure the per-drive latency correlation with throughput and IOPS using Spearman's Rank Correlation Coefficient (SRCC [15]) across three representative services. A higher SRCC value indicates a stronger correlation. We can see that latency is more closely related to throughput than IOPS. Moreover, in certain services, latency is even negatively correlated with IOPS (e.g., block and database). Therefore, we decide to use throughput for modeling the workload pressure.

RQ3: How to automatically derive adaptive thresholds?

In Attempt 2 (§3.3), we discover that, though peer-evaluation can provide adaptive thresholds, this solution requires time-consuming tuning for different service types and drive models. Now, with workload pressure modeled by throughput, we are able to build the latency-vs-throughput (LvT) distribution. Then, we can use regression models on such distribution to define a statistically *normal* drive and subsequently use its upper bound as the adaptive thresholds for various environments.

To build such regression models, we need to determine the scope of drives to be included in the LvT distribution. The tradeoff is that including more samples (e.g., all drives from the service) can be more statistically confident but subject to a more diverse distribution—difficult to derive a clear upper bound. Therefore, we plot the distribution at three different scales in Figure 5 and discuss their pros and cons as follows.

Service-wise. In Figure 5a, we plot the LvT distribution of drives from three clusters (marked as red, yellow and blue) in the database service. First, we can observe that the clusters from the same service can have drastically different LvT distributions. For example, samples from the red cluster rarely overlap with those from the yellow cluster. This indicates that directly using all drives from the service to build the distribution is not applicable.

Cluster-wise. In Figure 5a, we notice that even samples from the same cluster can have clear-cut distributions (i.e., the two red regions). After statistical analysis, we discover that the disparity is widespread. For clarity, in Figure 5b, we plot the LvT distribution of drives from two different nodes (in red and blue) from the same cluster. The huge gap between distributions indicates that we also cannot rely on cluster-wise peer evaluation.

Node-wise. Finally, in Figure 5c, we plot the LvT distribution of drives from one all-flash node (12 SSDs). Each drive is represented with a color. We can see the colors are well clustered together which indicates drives from the same node follow a similar LvT distribution. Note that we also examine

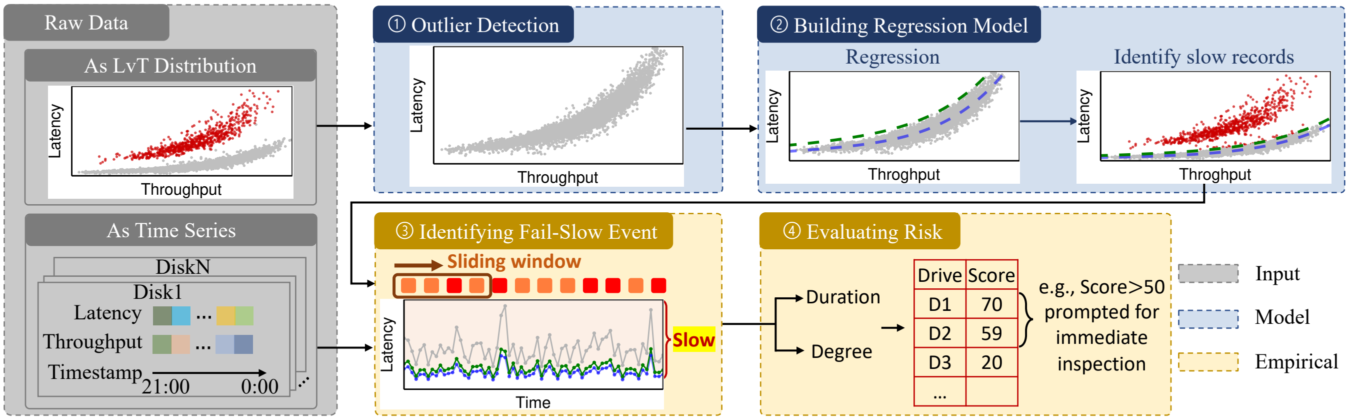


Figure 7: PERSEUS design diagram (§4.1). *From the raw data,* PERSEUS *seeks to distinguish the slow (in red) from the normal (in grey) by building a regression model (②) with preliminary outlier detection (①). With a sliding window,* PERSEUS *formulates consecutive slow records into slowdown events (③), and assigns risk scores based on slowdown duration and degree (④).*

nodes from other services and confirm that such behaviors persist across node configurations (e.g., all-flash or hybrid) and services (e.g., block or object storage). Thus, we decide to use the node-wise samples to build the LvT distribution.

RQ4: How to identify fail-slow without a criterion?

Unlike fail-stop failures, there are no clear criteria for detecting fail-slow drives. First, both the device (e.g., an SSD) and the software (i.e., users' code) can be a blackbox to the on-site engineers. Second, fail-slow failures can be temporal with varying symptoms. Moreover, the root causes of fail-slow failures can be too obscure to diagnose. As a result, we cannot exclude the possibility of mislabeling fail-slow failures.

Therefore, we rethink our strategy on fail-slow detection. Instead of relying on the framework to output a binary result (fail-slow or not), the detection tool should describe the likelihood of a drive to fail-slow. With sufficient accuracy, on-site engineers can focus on the most severe ones. While this may still leave some fail-slow drives undiscovered, it is acceptable as they behave like normal performance variations.

4 PERSEUS

With lessons from previous attempts, we propose PERSEUS, a non-intrusive, fine-grained and general fail-slow detection framework. The core idea of PERSEUS is building a polynomial regression on the node-level LvT distribution to automatically derive an adaptive threshold for each node. PERSEUS can use the threshold to formulate fail-slow events and further use a scoreboard mechanism to single out the drives with severe fail-slow failures. In this section, we first introduce the high-level workflow and then discuss the design of each step at length.

Our dataset can be viewed as a time-series dataset and each entry has five fields (i.e., avg_latency, avg_throughput, drive_ID, node_UID, timestamp). Every day, the monitoring proxy would gather 720 entries (180 minutes × 4 entries/min) from each drive (as raw dataset) and send them to PERSEUS for a four-step detection procedure.

4.1 High-Level Workflow

1. Outlier detection. For each node, PERSEUS first collects all the entries. Then, we use a combination of Principal Component Analysis (PCA [1]) and Density-Based Spatial Clustering of Applications with Noise (DBSCAN [43]) to identify and then discard outlier entries.

2. Building regression model. Based on the clean dataset (i.e., excluding the outliers), PERSEUS performs a polynomial regression to obtain the model and uses the prediction upper bound as a fail-slow detection threshold. Then, PERSEUS applies the model onto the raw dataset (i.e., including the outliers) to identify out-of-bound entries and mark them as slow entries.

3. Identifying fail-slow events. PERSEUS uses a sliding window and a slowdown ratio to identify consecutive slow entries and formulate corresponding fail-slow events.

4. Evaluating risk. Based on a risk-score mechanism [25], PERSEUS estimates the duration and degree of fail-slow events and assigns each drive a risk score based on daily accumulated fail-slow events. On-site engineers can then investigate the cases based on the severity.

4.2 Outlier Detection

Before applying regression models, a necessary pre-process is to root out noisy samples (i.e., outliers). While the LvT samples (i.e., <latency, throughput> pairs) are usually clustered together within a node (see RQ3 in §3.5), entries from fail-slow drives or under normal performance variations (e.g., internal GC) can still be deviating. Therefore, before building a polynomial regression model, we first screen out the outliers.

Using DBSCAN. Density-based clustering algorithms (measuring the spatial distance) are promising approaches for identifying the potentially distinctive groups (i.e., normal vs. slow). Initially, we employ DBSCAN [43] to label outliers.

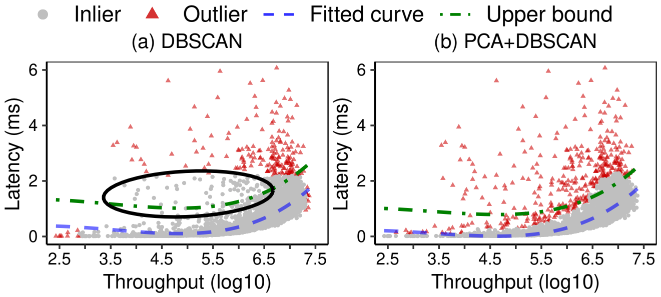


Figure 8: Outlier detection (§4.2). *The figures show the performance of the regression model based on two outlier detection schemes: (a) only DBSCAN and (b) PCA prior to DBSCAN. All data records come from the same node with one known fail-slow drive during the day.*

In a nutshell, DBSCAN groups points that are spatially close enough—distances between points is smaller than a minimum value. Note that <latency, throughput> pairs from long-term or permanent fail-slow drives can be clustered together but far away from the main cluster. Hence, we only keep one group with the most points for further modeling.

Adding PCA. Unfortunately, using the DBSCAN to sift through the raw dataset can have limited effectiveness. Here, we choose a sample node with one confirmed fail-slow drive to illustrate the limitation. In Figure 8a, we apply fine-tuned DBSCAN (outliers in red points) to the node's daily raw dataset and fit the rest of the data (grey points) to polynomial regression (with a fitted curve in blue and a 99.9% prediction upper bound in green dashed line). In this example, the DBSCAN algorithm only identifies 63.83% of slow entries.

The root cause is that the throughput and latency are positively correlated. Thus, the samples (i.e., <latency, throughput> pairs) can be skewed towards a particular direction. Hence, outliers (i.e., samples from fail-slow drives) can be mislabeled as inliers (see the black circle in Figure 8a). Therefore, we leverage Principal Component Analysis (PCA [1]) to transform the coordinates and penalize the outliers perpendicular to the skewed direction in order to reduce mislabeling. As a result, applying DBSCAN with PCA effectively detects 92.55% of slow entries (see Figure 8b).

Usage of outliers. Recall that in RQ4, we have discussed that just using binary detection cannot reflect the extent of slowdown. Therefore, we do not directly use the binary results of outlier detection, such as simply labeling outliers as slow entries (i.e., skipping §4.3) or fail-slow events (i.e., skipping §4.3 and §4.4). Rather, we exclude the outliers to build a better-fitted model for measuring slowdown degree of entries.

4.3 Regression Model

As normal drives within a node can have similar latency-vs-throughput mapping (i.e., well clustered together), we can use a regression model to describe the behavior of a "normal" drive and delineate the scope of variation for fail-slow detection. Classic regression models include linear, polynomial, and advanced ones like kernel regression. We do not use linear regression as the latency dependency on throughput is obviously nonlinear (e.g., see Figure 8). Moreover, advanced models (e.g., kernel regression) are unnecessary as the latency-vs-throughput mapping is primarily monotonic (i.e., latency increases along with the throughput). Polynomial regression is preferable as it handles nonlinearity while retaining model parsimony (i.e., achieving the desired goodness of fit with just enough parameters).

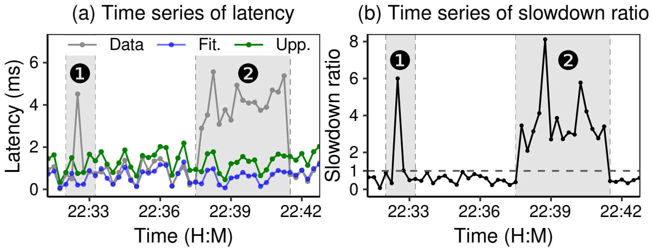


Figure 9: Identifying slowdown event (§4.4). *Figure (a) shows the original ("Data") time series of drive latency, together with the corresponding fitted values ("Fit.") and upper bounds ("Upp.") calculated from the regression model. Figure (b) shows the time series of slowdown ratio by dividing upper bound by original data. Records with a slowdown ratio higher than 1 are deemed as slow. The two grey boxes refer to a latency spike (❶) and a transient slowdown event (❷).*

4.4 Identifying Fail-Slow Event

Distinguishing slow entries. After obtaining the regression model, we can calculate a prediction upper bound to distinguish the slow entries, and use it to detect fail-slow events. For example, a 99.9% upper bound means that 99.9% of the variations are deemed normal. In practice, we use a combination of loose (i.e., 95%) and strict (i.e., 99.9%) upper bounds to avoid overfitting while identifying as many fail-slow drives as possible.

Formulating events. Next, we use both real and made-up examples to illustrate how to formulate fail-slow events. Figure 9a presents the drive latency (grey line), fitted values (blue line) and the 99.9% upper bound (green line). Slowdown Ratio (SR) is obtained from dividing drive latency by the upper bound, entry by entry (every 15 seconds). For example, let a drive's latency entries in one minute be [15, 20, 25, 10, 5] and the corresponding upper bound be [5, 5, 5, 5, 5]. The SR series would be [3, 4, 5, 2, 1]. Figure 9b presents the SR series of the candidate drive.

Next, we formulate fail-slow events by using a sliding window (similar to Attempt 2 in §3.3). The sliding window has a fixed length (i.e., a *minimum span*) and starts at the first entry. Within the span, if a certain *proportion* of SR series has a median SR value exceeding the *threshold*, PERSEUS would record that the drive has encountered a fail-slow event within the span and see if the event should be extended to the next entry.

Slowness	Duration (min)		
	Long-term ≥120	Moderate [60, 120)	Temporal [30, 60)
Severe (SR≥5)	Extreme	High	Moderate
Moderate (SR∈[2, 5))	High	Moderate	Low
Mild (SR∈[1, 2))	Moderate	Low	Minor

Table 3: Fail-slow risk matrix (§4.5). PERSEUS *assigns risk levels based on daily accumulated slowdown events. For example, drives at extreme risk for one day should experience a long-term slowdown (for 120-180 minutes in total) with a severe slowdown ratio (SR) on average (SR ≥5).*

As for the example above (i.e., an SR series of [3, 4, 5, 2, 1]), we set the *minimum span* as one minute (i.e., four entries), the *proportion* to be 50%, and the *threshold* to be 1. Then, the first four SR entries can form a fail-slow event as more than 50% of the SR values (i.e., 3, 4, 5, 2) have a higher median (i.e., 3.5) than the threshold (i.e., 1). For the same reason, this fail-slow event should include the fifth entry (i.e., 1).

In practice, we set the *minimum span* as 5 minutes, the *proportion* to be 50%, and the *threshold* to be 1, meaning the event should be slower than the upper bound. Our rationale is to only formulate fail-slow events under a persistent series of slowdown entries as one-off spike entries are likely to be acceptable performance variations. In Figure 9b, while both ❶ and ❷ have high SR values, only ❷ would be marked as a fail-slow event.

4.5 Risk Score

Recall our discussion on RQ4 and the usage of outlier detection, we also do not simply rely on the existence of fail-slow events to label the corresponding drive as "fail-slow." In fact, if we simply mark drives with one fail-slow event as fail-slow, we can easily obtain 6K such "fail-slow" cases on a bad day.

Therefore, we adopt the idea of establishing a risk score mechanism from performance regression testing [25]. In Table 3, slowdown duration and severity are classified into different risk levels (in shades of grey). For example, in our case, according to the daily slowdown span, the duration of fail-slow is classified into temporal (from 30 to 60 minutes), moderate (from 60 to 120 minutes) and long-term (from 120 to 180 minutes). Besides, based on the average slowdown ratio of the day, the slowness of fail-slow is evaluated as mild (1≤SR<2), moderate (2≤SR<5), or severe (SR≥5).

To examine fail-slow, a per-drive risk score is calculated by assigning different weights to risk levels:

$$\begin{aligned} Risk\ Score = {} & N_{extreme} \times 100 \ + \ N_{high} \times 25 \ + \\ & N_{moderate} \times 10 \ + \ N_{low} \times 5 + \ N_{minor} \times 1 \quad (1) \end{aligned}$$

$N_{extreme}$ *refers to #days at extreme risk level.*

If a drive whose risk scores exceed a minimum value (i.e., *min_score*) within the most recent *N* days, the drive will be recommended for immediate isolation and hardware inspection. Note that all drives in our fleet, HDDs and SSDs, use the same scoring mechanism.

Service	Device	#Node	#Fail-slow
Stream processing	NVMe SSD	47	1
Table storage	NVMe SSD	87	1
Big data	NVMe SSD	119	1
Data warehouse	NVMe SSD	663	1
Database	NVMe SSD	96	2
E-commerce	NVMe SSD	223	6
Log service	SATA HDD	34	36
Object storage	SATA HDD	1426	42
Block storage	NVMe SSD	734	225
Total	**-**	**3429**	**315**

Table 4: Test dataset size (§5.1).

5 Evaluation

5.1 Fail-slow Benchmark

One significant challenge of testing fail-slow detection frameworks is the lack of a benchmark. Existing fail-slow datasets [19, 36] only record high-level administrative information of fail-slow incidents and thus cannot be used for evaluation. Therefore, we build and release a large-scale fail-slow detection benchmark based on verified fail-slow drives and production-level traces.

Benchmark size. Table 4 presents a summary of our benchmark. Specifically, our dataset includes 886 million operational traces of 15 consecutive days from 41K drives and 25 clusters. Among them, 315 drives (237 SSDs and 78 HDDs) are verified fail-slow and thus labeled as positive; the rest are normal peer drives from the same clusters. Among the verified cases, 304 are detected by PERSEUS. All fail-slow drives are verified by either on-site engineers or manufacturers. Their root causes include software scheduling bugs, hardware defects, and environmental factors.

Workload heterogeneity. The benchmark covers 9 major services (see Table 4). These services can have drastically different I/O accessing patterns and subsequently various LvT distributions.

Benchmark setup. In our dataset, 252 fail-slow drives are caused by software scheduling (see Section 6.1). To avoid potential concerns that PERSEUS may be specific to the Alibaba stack, we set up two scopes: (1) the full test dataset, and (2) a subset excluding traces from clusters with software-induced fail-slow drives. We only show results with the highest evaluation scores in Table 5 and Table 7.

5.2 Test Candidates

We compare PERSEUS with three models based on our early explorations, namely threshold filtering (§3.2), peer evaluation (§3.3), and the IASO-based model (§3.4). In this section, we introduce their implementation and configuration details.

5.2.1 Threshold Filtering

We include both statistical and empirical thresholds.

Statistical bound. We derive the following statistics as the upper bounds: (1) an X^{th} percentile where X ranges from 75 to 99; or (2) an interquartile range ($IQR = 3rd\_quartile - 1st\_quartile$) [30]. Drives are classified as fail-slow if their median latency during the three-hour monitoring exceeds the upper bound.

Empirical bound. We manually set a latency upper bound for each node setup of each service based on the Service Level Objectives (SLOs) or the suggestions from on-site engineers (e.g., 300 μs for the *all-flash* setup in block storage service).

5.2.2 Peer Evaluation

Recall that the peer evaluation approach identifies a fail-slow drive if its latency is at least X times that of the node median for a minimum duration (denoted *min_dur*). To obtain the best performance, we explore different sets of parameters (i.e., X from 1.5 to 3 and *min_dur* from 0 to 150 minutes).

5.2.3 IASO-Based Model

We re-implement IASO [36] strictly following its original design and only modify parts when necessary. Since IASO detects fail-slow on a per-node basis, we label the results as true positive if the node contains a fail-slow drive. We list key implementation details as follows.

Epoch. The size of each `epoch`, instead of 5 seconds, is adjusted to 15 seconds (the finest granularity of our raw dataset).

Timeout. The `timeout` signals are converted to the number of *slow* drives in a node. During each `epoch`, the *slow* drives refer to the drives whose latency records (i) exceed pre-defined empirical bounds (Attempt 1 in §3.2) or (ii) are at least 2× the median latency of all drives from the same node (Attempt 2 in §3.3). The `response` is set as the total number of drives in each node.

DBSCAN configuration. IASO records peer scores (the higher the slower) among nodes, and only keeps one outlier with the highest score for its further mitigation procedure. Here, since we only evaluate the detection part of IASO, we retain all outliers classified by the DBSCAN and set different score thresholds to fine-tune IASO.

5.2.4 PERSEUS

The deployed PERSEUS adopts outlier detection (§4.2) and uses the combination of two prediction upper bounds (i.e., 95% and 99.9%) to formulate fail-slow events (§4.4). The monitoring period (N) is set as 15 days for both upper bounds, and the alert score (*min_score*) is set as 90 for the former and 40 for the latter (§4.5). Note that PERSEUS uses the same set of parameters for all node configurations (e.g., all-flash and hybrid) across different services (e.g., block/object storage and big data).

5.3 Evaluation Metrics

We adopt three evaluation metrics: precision rate, recall rate, and Matthews Correlation Coefficient (MCC [33]). The precision indicates the percentage of drives identified by a method is indeed a fail-slow one. The recall is the percentage of real fail-slow drives identified by a method. Since our test dataset is highly unbalanced (i.e., the positive-to-negative ratio is 1:137), we further adopt MCC as it evaluates binary classification models more fairly on imbalanced datasets and can offer a more informative and convincing score compared to other widely adopted metrics like accuracy and F1-score [12].

Metric	Thresh-Stat	Thresh-Emp	Peer Eval	IASO-Based	PERSEUS-Deployed
Full-set					
Precision	1.00	1.00	0.98	0.48	0.99
Recall	0.52	0.02	0.57	0.24	1.00
MCC	0.72	0.14	0.74	0.32	0.99
Subset (excluding software-induced)					
Precision	1.00	1.00	1.00	0.45	0.94
Recall	0.71	0.09	0.65	0.61	1.00
MCC	0.84	0.30	0.80	0.52	0.97

Table 5: Overall evaluation results (§5.4). *The table shows the best evaluation scores of threshold filtering based on statistical (**ThreshStat**) and empirical (**ThreshEmp**) bounds, peer evaluation (**PeerEval**), and the* IASO*-based model. For* PERSEUS*, we list the results of the deployed version. Each method is evaluated on both the full-set and the subset.*

5.4 Evaluation Results

Table 5 summarizes the performance results. For previous attempts (Thresh-Stat to IASO-Based columns), we choose the best performance. For PERSEUS, we use the results of the deployed version. The upper half includes the results from the full benchmark tests and the lower half includes results from the benchmark excluding the scheduling-induced failures.

Threshold-based. We can see that, in both the full-set and the subset, the two threshold-based approaches can achieve a precision of 100%. However, the recalls are subpar, especially the empirical threshold method. This is understandable as strict thresholds, on the one hand, can expose extremely slow drives which are highly likely to be caused by fail-slow failures. On the other hand, such methods can leave drives with only mild fail-slow symptoms undiscovered.

Peer-evaluation. Using peer evaluation to detect fail-slow failures also yields high precision but low recall. The reason is that it generally adopts an adaptive threshold (i.e., X times the node median) to formulate fail-slow events. Thus, it faces the same problem as threshold-based methods.

IASO-based. While we have tried our best effort on refactoring and fine-tuning IASO, its performance is rather disappointing. With an MCC score of 0.32, IASO even falls behind using a simple statistical threshold or using a peer-evaluation method. We believe there are two aspects of reasons. First, IASO heavily relies on software timeouts which can not be simply replaced with other metrics (e.g., node-level slow drives). Second, its algorithm is designed for node-level detection where a finer grained event (e.g., a fail-slow drive) may not trigger enough alerts.

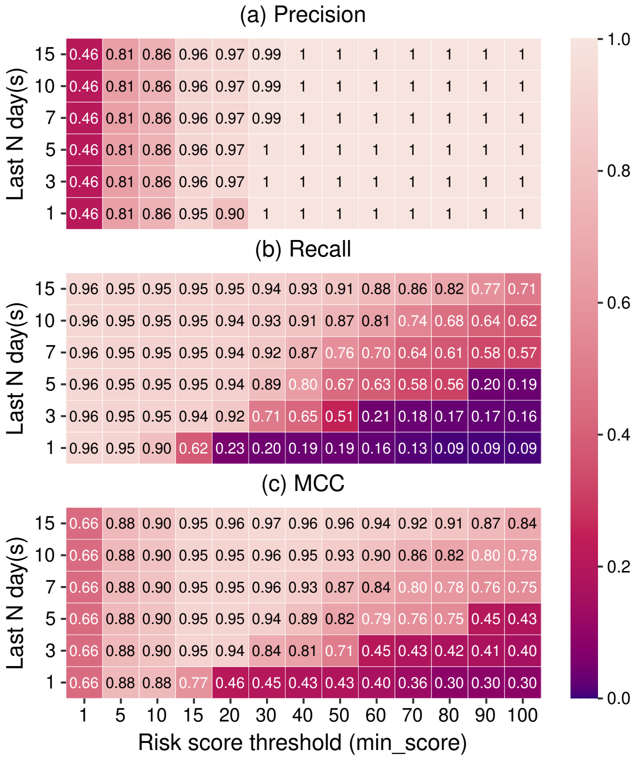


Figure 10: *min_score* **and** *N* **of PERSEUS (§5.5).** *The heatmaps show evaluation scores of* PERSEUS*-p999 in Table 7 under different min_score and N. The lighter the color, the higher the score, the better the result.*

PERSEUS. Table 5 shows that PERSEUS outperforms all previous attempts. The high precision and recall indicate that PERSEUS can successfully detect all fail-slow drives while rarely mislabeling normal as fail-slow. Therefore, we conclude that PERSEUS achieves our design goals as a fine-grained (per-drive), non-intrusive (no code changes), general (same set of parameters for different setups) and accurate (high precision and recall) fail-slow detection framework.

5.5 Effectiveness of PERSEUS Design

Now, we take a closer look at the effectiveness of procedures and sensitivity of parameters in PERSEUS. In Table 6, we list the main options or ranges of configurable parameters in PERSEUS. Next, we discuss the effectiveness of PERSEUS's procedures by enabling or disabling particular functionalities and explore different sets of parameters. The evaluation results are listed in Table 7. Note that the deployed version uses a combination of p95 and p999 upper bounds.

Outlier detection. By disabling the outlier detection, we can see that the precision is approximately the same, while the recall plummets to only 0.51. This indicates that, without outlier detection, PERSEUS may fail to distinguish samples from fail-slow drives, thus yielding a low recall.

PCA. Surprisingly, if we enable outlier detection but disable PCA, we find out the performance becomes even worse than simply without outlier detection. This experiment confirms the importance of correcting mislabeling samples via PCA.

Parameter	Range	Description
S1: Outlier detection (§4.2)		
PCA	On/Off	Transform the coordinates w.r.t. the principal components.
DBSCAN	On/Off	Density-based outlier detection.
S3: Identifying fail-slow event (§4.4)		
X	95∼99.9	Use the *X*% prediciton upper bound as the latency upper bound.
S4: Evaluating risk (§4.5)		
min_score	1∼100	Risk score threshold.
N	1∼15	Evaluate the risk score of the most recent *N* days.

Table 6: Evaluating PERSEUS's design choices (§5.5). *The table summarizes the parameter settings of evaluating* PERSEUS*'s design choices. PCA and DBSCAN are switched on and off to demonstrate their effectiveness.*

Metric	w/o Outlier	w/o PCA	p95	p99	p999	Deployed
Full-set						
Precision	0.98	0.55	0.99	1.00	1.00	0.99
Recall	0.51	0.43	0.99	0.93	0.93	1.00
MCC	0.71	0.49	0.99	0.96	0.96	0.99
Subset (excluding software-induced)						
Precision	0.95	0.36	0.94	0.98	1.00	0.94
Recall	0.82	0.91	0.95	0.92	0.95	1.00
MCC	0.88	0.57	0.95	0.95	0.98	0.97

Table 7: Evaluation results of PERSEUS's design choices in Table 6 (§5.5). *Only results based on the best sets of parameters with the highest MCC scores are shown.*

Prediction upper bounds. We set various prediction upper bounds from p95 to p999. The optimal one on the full-set is the p95 upper bound, with nearly perfect precision and recall both at 0.99. For the subset, a stricter bound of p999 is preferred as fail-slow in the subset is usually with severe slowdowns. In practice, the deployed version uses a combination of p95 and p999 upper bounds to strike a better balance on both benchmarks.

Risk score mechanism. Figure 10 evaluates the scoring mechanism. With a larger *min_score* and *N*, the precision usually increases while the recall decreases. This is because, as *N* becomes larger, drives—that have occasional but less severe slowdowns—could be misidentified with enough scores counted from more days.

Evaluating the read performance. Fail-slow failures impact the write performance more often than the read. Among the 315 fail-slow drives in our dataset, 49 are fail-slow in both write and read while 223 are only fail-slow in the write. Unfortunately, we do not have read traces of the remaining 43 fail-slow ones.

Runtime overhead. On Intel Xeon 8-core CPU 2.5GHz with 16GB RAM, the per-node execution time of PERSEUS is measured as 0.21±0.14 seconds. With low overheads, PERSEUS can detect fail-slow drives in tens of thousands of nodes each day on a single machine.

5.6 Benefit of Deployment

The most direct benefit of deploying PERSEUS is reducing tail latency. By isolating the fail-slow, node-level 95th, 99th and 99.99th write latencies are reduced by 30.67% (±10.96%), 46.39% (±14.84%), and 48.05% (±15.53%), respectively.

6 Root Cause Analysis

We further analyze the root causes of the 315 fail-slow drives in the test dataset. Among them, 216 SSDs and 36 HDDs are impacted by ill-implemented software scheduling. The remaining 42 HDDs and 21 SSDs are verified by our on-site engineers as hardware-related fail-slow failures and further sent back to vendors for detailed analysis. Due to the lengthy diagnosis process, we only obtain root causes for 15 drives (9 HDDs and 6 SSDs).

6.1 Ill-Implemented Scheduler

6.1.1 Case 1: In Open-Channel SSD Cluster

Symptom. In two clusters, PERSEUS has identified a total of 216 fail-slow drives constantly showing abnormal performance. This is unconventional as hardware-related fail-slow occurrences are usually rare and independent. Further investigation reveals that all detected fail-slow drives are with the same logical IDs, i.e., $disk_1$ and $disk_2$. After checking the per-node latency time series, we discover that the latency of individual drives in these nodes is positively correlated with their logical IDs, i.e., latency level: $disk_1 > disk_2 > \cdots > disk_{12}$. In other words, among those nodes, $disk_1$ is always the slowest, followed by $disk_2$ and so on.

Root cause. Each node in these two clusters is equipped with 12 open-channel SSDs (OC SSDs), whose Flash Translation Layer (FTL) is managed by the host. For each node, the host allocates 12 CPU cores to manage 12 OC SSDs, respectively (e.g., $core_0$-$core_{11}$ for SSD_0-SSD_{11}). The root cause is that the OS scheduler places system tasks on CPU cores by domain (a domain includes 6 CPU cores). Upon receiving a new system task (e.g., ps command), the scheduler first checks if the current core and last-selected core are idle. If not, starting from the first core in the domain, the scheduler iterates through all cores in order and attempts to preempt a core for running the task. As a result, OC drives with smaller IDs (e.g., $disk_1$) are more likely to be preempted and encounter fail-slow failures.

Fix. We modify the scheduler to no longer preempt the CPU cores assigned to the OC SSDs. After the fix, OC SSDs in these clusters no longer suffer fail-slow failures.

6.1.2 Case 2: In All-HDD Cluster

Symptom. In one cluster, PERSEUS has identified 36 fail-slow HDDs. This cluster is of an all-HDD setup with 76 HDDs in each node. There are three interesting facts about the distribution of the fail-slow HDDs from this cluster. First, the fail-slow failures always show up in fixed combinations of pairs. For example, if there are two fail-slow HDDs in a node, they would be $disk_0$ and $disk_{75}$. If there are 6 fail-slow HDDs, they would be $disk_{0-2}$ and $disk_{73-75}$. Second, all fail-slow HDDs are experiencing a similar level of slowdown. Third, in each node, the number of fail-slow drives is always twice the number of offline drives.

Root cause. In each node of this cluster, the OS assigns each HDD a thread to manage its I/O. The assignment follows a simple algorithm:

$$Thread\_ID = Disk\_ID \ \ mod \ \ \#Drives. \tag{2}$$

Therefore, as each node has 76 HDDs, normally $thread_0$ manages $disk_0$ and so on. However, when a drive is put offline, the number of drives changes and the assignment acts accordingly. For example, assume $disk_{20}$ crashes and the total number of drives now becomes 75. Then, $disk_0$ and $disk_{75}$ would share $thread_0$ ($0 \equiv 75$ mod 75) and thus suffer from fail-slow failures due to I/O contention. Similarly, two or three drives crash can result in corresponding two or three pairs of fail-slow occurrences.

Fix. We modify the assignment policy to only allow one HDD per thread and thus avoid the fail-slow occurrences.

6.2 Hardware Defects

Bad sector. Bad sectors are usually an artifact of physical damage (e.g., manufacturing defect or scratched by the read-/write head) [40]. To deal with them, disk firmware maintains a pool of spare sectors to reallocate the original data on bad sectors. Moreover, firmware remaps the logical address of bad sectors to the physical address of spare sectors. Upon host requests on an unmarked bad sector, the disk will suffer from long seek time (i.e., time spent for reallocating data). According to field events, three HDDs are reported to have a large number of bad sectors, resulting in repeated remapping and reallocating, and obviously fail-slow failures. Note that one can not simply infer the root causes based on the occurrences of such errors. The reason is that the hardware defects are usually neither necessary nor sufficient conditions for fail-slow failures.

Rotor eccentricity. Disk motor spins the platter at high speed (e.g., 7200 RPM for consumer-level HDDs). If the rotor in the spindle motor rotates with eccentricity, it will cause a lot of noise and vibration. For such disks, the read/write heads would frequently fail to locate targeted positions, resulting in considerable I/O delay. Two fail-slow HDDs are reported to suffer from slight rotor eccentricity in our field events.

Bad capacitors. SSD adopts a small amount of DRAM as an internal write-back cache to boost both read and write performance. If the DRAM capacitors are malfunctioning, SSD will be forced to stop using the cache since it is volatile (i.e., data loss upon power down), causing severe performance degradation. Instead, SSDs now only ACK after data have been directly flushed to the NAND successfully, incurring long latencies on writes. In total, four SSDs are found to have bad capacitors.

Read-only mode. Drives with severe errors (e.g., in the face of too many bad sectors) are reset to (temporal) read-only mode to prevent further data loss. Upon read-only mode, drives are blocked from executing any write command. As a result, two SSDs are found to be stuck in read-only mode.

6.3 Environment

Temperature and power are common sources of fail-slow incidents [19, 48]. According to field events, one fail-slow HDD suffered from temperature throttling due to high environmental temperature. Another three HDDs were related to insufficient power supply events.

7 Limitation

Multiple fail-slow occurrences. PERSEUS leverages an important precondition that fail-slow failures should be rare in the field. However, if a key component on the critical data path (e.g., HBA card) breaks down, all drives would be impacted and result in severe delays. In this case, PERSEUS may not be able to detect the performance anomalies as the LvT distribution can be skewed for all drives. At the moment, we are investigating the possibility to perform inter-node LvT distribution to enhance PERSEUS's ability on discovering multiple fail-slow occurrences within the same node.

Generalizability. Utilizing the LvT distribution to identify fail-slow drives also depends on the fact that all drives within the node have the same drive models and similar workloads. In our storage systems, drives have the same configuration, and multiple levels of load balancing assure that the workload on each drive is similar within the same node. While this is a common practice for large-scale storage systems [30, 32, 35, 48], it might not be the case for small-scale servers (e.g., private cloud), where drives in the same node can have drastically different workloads and configurations. Under such circumstances, the accuracy of PERSEUS can be affected.

Comprehensiveness. PERSEUS currently uses traces from 9PM to 12AM each day to reduce interference (see Section 2.2). It is possible that some fail-slow failures could only be triggered during a particular time window or under heavier workloads. We are working on designing a more efficient daemon to collect traces during busy hours for PERSEUS. Moreover, we are exploring other device-level metrics to enrich what PERSEUS can take as key inputs.

8 Related Work

Fail-slow failure study and diagnosis. As an emerging failure mode, fail-slow failure has received growing attention from academia and industry. Early literature mainly focuses on diagnosing fail-slow as an overlooked failure mode [16, 19, 24]. For example, Huang et al. define gray failure in the cloud with an abstract model [24]. Do et al. [16] measure system-level performance degradation brought by limpware, and address the necessity to develop limpware-tolerant systems. Gunawi et al. [19] perform qualitative analysis on 101 hardware-incident reports from various institutions and reveal the underlying fail-slow root causes in various types of hardware. Our motivational study in Section 2.3 specifically evaluates the fail-slow impact on drive performance (i.e., I/O latency). Moreover, our work provides a more diverse root cause analysis on fail-slow in storage devices.

Fail-slow failure detection. There have been a few studies addressing fail-slow detection and localization [4, 23, 29, 36, 45, 50]. For example, Panda et al. [36] convert software-level timeout signals into fail-slow metrics and adopt peer evaluation to detect fail-slow nodes. Huang et al. design Panorama to detect production failures by increasing the in-site observability [23]. Different from the above, PERSEUS detects fail-slow specifically in storage devices and is merely based on performance metrics like latency and throughput.

9 Conclusion

In this paper, we first share our unsuccessful attempts in developing robust and non-intrusive fail-slow detection for large-scale storage systems. We then introduce the design of PERSEUS, which utilizes classic machine learning techniques and scoring mechanisms to achieve effective fail-slow detection. Since deployment, PERSEUS has covered around 250K drives and successfully identified 304 fail-slow drives.

Acknowledgements

We would like to thank our shepherd Sangyeun Cho, and the anonymous reviewers for their insightful comments and suggestions. This research was supported by NSFC (62102424, 62072306), the Alibaba Innovation Research (AIR) program, National Key R&D Program of China (2022YFB4500302), Program of Hunan Postdoc Innovation (2021RC2069), and Program of Shanghai Academic Research Leader (20XD1402100). The authors would also like to thank Amber Bi and Shiming Wang for their feedbacks on early versions of this paper.

References

[1] Hervé Abdi and Lynne J. Williams. Principal component analysis. *WIREs Computational Statistics*, 2010.

[2] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated crash vulnerabilities. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[3] Jacob Alter, Ji Xue, Alma Dimnaku, and Evgenia Smirni. Ssd failures in the field: Symptoms, causes, and prediction models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2019.

[4] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[5] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[6] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.

[7] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawa, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the effects of disk-pointer corruption. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2008.

[8] Yu Cai, Yixin Luo, Saugata Ghose, and Onur Mutlu. Read disturb errors in mlc nand flash memory: Characterization, mitigation, and recovery. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015.

[9] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 1996.

[10] Wei Chen, S. Toueg, and M. Kawazoe Aguilera. On the quality of service of failure detectors. In *Proceedings of the 30th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2000.

[11] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, Jaehong Kim, Jaechun Park, Ki-Whan Song, Ki-Tae Park, Sangyeun Cho, Hwaseok Oh, Daniel D.G. Lee, Jin-Hyeok Choi, and Jaeheon Jeong. A flash memory controller for 15μs ultra-low-latency ssd using high-speed 3d nand flash with 3μs read time. In *Proceedings of the IEEE International Solid State Circuits Conference (ISSCC)*, 2018.

[12] David Chicco and Giuseppe Jurman. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC Genomics*, 2020.

[13] Brian Choi, Randal Burns, and Peng Huang. Understanding and dealing with hard faults in persistent memory systems. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, 2021.

[14] Allen Clement, Edmund Wong, Lorenzo Alvisi, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[15] GW Corder and DI Foreman. *Nonparametric Statistics: A Step-by-Step Approach.* Wiley, 2014.

[16] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*, 2013.

[17] Norman R Draper and Harry Smith. *Applied Regression Analysis*. Wiley, 1998.

[18] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

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

[20] Trinabh Gupta, Joshua B. Leners, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *Proceedings*

of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2013.

[21] Shujie Han, Patrick P. C. Lee, Fan Xu, Yi Liu, Cheng He, and Jiongzhou Liu. An in-depth study of correlated failures in production SSD-based data centers. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[22] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The tail at store: A revelation from millions of hours of disk and ssd deployments. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[23] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[24] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[25] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. Performance regression testing target prioritization via performance risk analysis. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[26] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing Closed-Source binary device drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC)*, 2010.

[27] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[28] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. Pcatch: Automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, 2018.

[29] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[30] 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 *In Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[31] Ao Ma, Fred Douglis, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. RAIDShield: Characterizing, monitoring, and proactively protecting against disk failures. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[32] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. A study of SSD reliability in large scale enterprise storage deployments. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[33] Brian Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 1975.

[34] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2015.

[35] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. Ssd failures in datacenters: What? when? and why? In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR)*, 2016.

[36] 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 *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[37] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application crash consistency and performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[38] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[39] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. SymDrive: Testing drivers without devices. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[40] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding latent sector errors and how to protect against them. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[41] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[42] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: A large-scale field study. In *Proceedings of the 2009 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009.

[43] Erich Schubert, Jörg Sander, Martin Ester, Hans Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: Why and how you should (still) use dbscan. *ACM Transactions on Database Systems*, 2017.

[44] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Maheswara Rao G. Uma, and Haryadi S. Gunawi. Pbse: A robust path-based speculative execution for degraded-network tail tolerance in data-parallel frameworks. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[45] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active device and link failure localization in data center networks. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[46] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS)*, 2012.

[47] Benjamin Walker. Spdk: Building blocks for scalable, high performance storage applications. In *Storage Developer Conference*, 2016.

[48] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and actions: What we learned from 10k ssd-related storage system failures. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[49] Jinfeng Yang, Bingzhe Li, and David J. Lilja. Exploring performance characteristics of the optane 3d xpoint storage technology. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 2020.

[50] Qiao Zhang, Guo Yu, Chuanxiong Guo, Yingnong Dang, Nick Swanson, Xinsheng Yang, Randolph Yao, Murali Chintalapati, Arvind Krishnamurthy, and Thomas Anderson. Deepview: Virtual disk failure diagnosis and pattern detection for azure. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance

Jinghuan Yu[1] Sam H. Noh[2,3*] Young-ri Choi[2] Chun Jason Xue[1]

[1]City University of Hong Kong [2]UNIST [3]Virginia Tech

Abstract

Log-Structure Merge-tree (LSM) based Key-Value (KV) systems are widely deployed. A widely acknowledged problem with LSM-KVs is write stalls, which refers to sudden performance drops under heavy write pressure. Prior studies have attributed write stalls to a particular cause such as a resource shortage or a scheduling issue. In this paper, we conduct a systematic study on the causes of write stalls by evaluating RocksDB with a variety of storage devices and show that the conclusions that focus on the individual aspects, though valid, are not generally applicable. Through a thorough review and further experiments with RocksDB, we show that data overflow, which refers to the rapid expansion of one or more components in an LSM-KV system due to a surge in data flow into one of the components, is able to explain the formation of write stalls. We contend that by balancing and harmonizing data flow among components, we will be able to reduce data overflow and thus, write stalls. As evidence, we propose a tuning framework called ADOC (Automatic Data Overflow Control) that automatically adjusts the system configurations, specifically, the number of threads and the batch size, to minimize data overflow in RocksDB. Our extensive experimental evaluations with RocksDB show that ADOC reduces the duration of write stalls by as much as 87.9% and improves performance by as much as 322.8% compared with the auto-tuned RocksDB. Compared to the manually optimized state-of-the-art SILK, ADOC achieves up to 66% higher throughput for the synthetic write-intensive workload that we used, while achieving comparable performance for the real-world YCSB workloads. However, SILK has to use over 20% more DRAM on average.

1 Introduction

LSM (Log-Structure Merge-tree based)-KV systems buffer their random updates in a memory batch to leverage the disk's high sequential write performance characteristic to support write-intensive workloads. These systems use background data movements to persist cached data, trim the redundant entries, and reshape the storage components to ensure IO performance. LSM-KVs are used in products such as NoSQL storage systems [17,25,41], data warehouses [52], time-series databases [32] and embedded storage engine in RDBMS [15, 48]. Although LSM-KVs can provide higher write throughput, they frequently encounter the write stall phenomenon when facing high write pressure workloads [16,46].

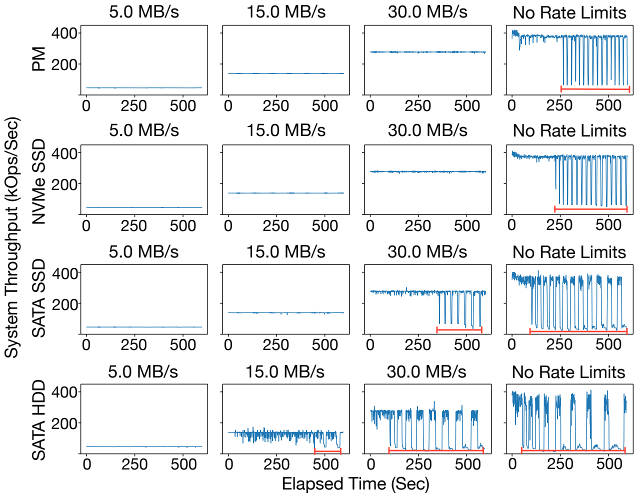


Figure 1: The figures show the throughput while running *fill-random* with increasing write pressure (represented as numbers on top of each graph) and for different storage devices. As writing pressure increases, system throughput on all devices shows patterns of sudden performance drop and even maintains long-term stalling states. Such significant performance drops are referred to as the write stall phenomenon.

The write stall phenomenon refers to the sudden drop in system throughput as marked by the red lines in Figure 1, which shows results for RocksDB running with write-intensive workloads on a wide range of storage devices, from a traditional HDD to a modern state-of-the-art Intel Optane DC PMM persistent memory device (denoted PM). Based on this figure, we

[*] This work was done while at UNIST.

identify two characteristics of the write stall phenomena: first, write stalls are *universal*, that is, they occur on all types of devices, though they may be triggered under different conditions, and second, write stalls are strongly *device dependent*, with their duration and rate of performance degradation influenced by various factors such as device type and write intensity. As a notable shortcoming of LSM-KV systems, the write stall phenomenon has been the subject of extensive research and attention in recent years [7, 43, 45, 46, 50, 58, 60].

This study presents ADOC (**A**utomatic **D**ata **O**verflow **C**ontrol), a framework with a goal of minimizing write stalls by harmonizing the flow of data between LSM-KV components. To this end, we first perform an extensive experimental study to analyze the occurrence pattern of write stalls. We find that previous studies [7, 43, 45, 46, 50, 58, 60] are conducted with particular settings, which make their analysis difficult to generalize. Many former studies conclude the cause of write stalls to be resource depletion, while we find that write stalls are triggered even when the hardware resources are sufficient. This indicates that write stalls not only happen in passive blocking situations, but also occur when the system proactively stalls the input stream in attempts to avoid further performance loss. We find that popular LSM-KVs like LevelDB and RocksDB already use active stalling strategies to avoid "Disk Overflow", which refers to the situation where flush or compaction jobs cannot keep up with the incoming write rate [28].

Through deeper analysis, we find the source of write stalls to be a more general form of disk overflow, which we refer to as "data overflow." Specifically, data overflow refers to the rapid expansion of one or more components in an LSM-KV system due to a surge in data flow into one of the components.

We categorize data overflows into three scenarios depending on the component that forms the data overflow. We also show how data overflow is able to explain the limitations that could not be explained with earlier studies. Based on these observations, we design and implement ADOC, an automatic tuning framework, to universally control and harmonize the data flow among LSM-KV components such that data overflow may be avoided. ADOC has the following four key features: 1) it improves performance by reducing write stalls through balanced use of resources; 2) it is a device transparent solution that improves performance for a wide range of devices, from traditional HDDs to state-of-the-art SSDs and PM devices; 3) it is an automatic tuning system that does not require human intervention, and 4) it is highly portable as it can be implemented by the native interfaces of LSM-KV systems.

Experimental results with RocksDB show that ADOC reduces the duration of write stalls by as much as 87.9% in the best case and improves performance by as much as 322.8% compared with the auto-tuned RocksDB, which takes a similar auto-tuning approach of ADOC. We also compare ADOC with the state-of-the-art LSM-KV SILK [7]. While there have been multiple novel LSM-KVs proposed more recently [11, 43, 44, 58], they mostly concentrated on making use of PM. As our target is a general-purpose LSM-KV that can accommodate all types of devices, we chose to compare it with SILK. Compared to SILK, ADOC achieves up to 66% higher throughput for the synthetic write-intensive workload that we used, while achieving comparable performance for the real-world YCSB workloads due to the higher read performance of SILK. However, SILK attains this performance at the expense of using 22.2% more main memory on average.

The source code of ADOC is available online [3].

2 Background

2.1 Advanced Storage Devices

Recent developments in storage technology have led to revolutionary advances in storage media. Two kinds of storage media, NVMe SSD and Persistent Memory, have entered the public realm and have been widely studied.

NVMe SSD: NVMe SSD refers to a class of SSD devices that are linked to the host via the PCIe bus and communicate with the host using the NVMe (Non-Volatile Memory express) protocol. NVMe is a communication interface and drivers designed for PCIe-based SSDs aim for efficient performance and interoperability. The command set in NVMe allows devices to directly communicate with the system CPU without an extra bus controller. Combined with the expanded command queue, NVMe provides much higher parallelism than conventional protocols like SATA and SAS.

Persistent Memory: Persistent memory (PM), or non-volatile memory (NVM), is a persistent medium that provides byte-addressability. The commercial PM product, Intel's Optane DC PM, can be deployed in Memory mode, as expanded memory, or App-direct mode, as a (block-device-like) storage device. Optane DC PM uses 3D XPoint technology, which, compared to traditional NVMe SSDs, offers lower write latency, provides byte addressing, and does not require garbage collection. However, as PM is attached to the memory bus, the IO processing of PM consumes more CPU resources. Also, as the PM device has limited bandwidth compared to DRAM, application bandwidth tends to quickly saturate as the number of threads increases [33, 56]. While Intel has announced the discontinuation of their Optane storage products effectively terminating 3D XPoint based products [20], other forms of NVM are still in development [35, 55]. Furthermore, we continue to see new developments such as CXL SSDs [35] that are expected to provide high-performing persistent storage similar to the Optane products.

2.2 Architecture of LSM-KVs

The majority of LSM-KVs follow the structure as that of RocksDB [25], which is one of the most popular LSM-KVs in industry and has been used as the platform of choice in many prior academic studies [7, 11, 14, 49, 58]. The structure consists of three major components and two data movement jobs to maintain these components as shown in Figure 2.

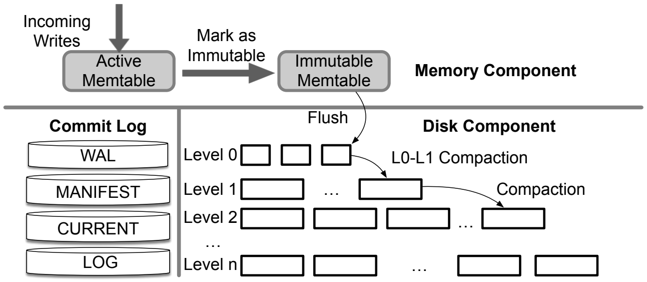


Figure 2: Architecture of RocksDB [15, 25, 48].

More specifically, the memory component caches the newest updates through an active and immutable Memtable [10, 17, 25, 41]. These Memtables are generally implemented with a skip list though other data structures such as a vector or hash table may be used. The commit log is the component that ensures system consistency. It is used to recover data when the system reboots from system failures. Finally, the disk component organizes the persisted data. Sorted String Tables (SSTable) that serve as the basic unit [46] are organized in a hierarchical manner in levels, starting from Level 0 to deeper (i.e., higher numbered) levels.

There are two types of background data movement jobs, flush and compaction. Flush moves the Immutable Memtable from the memory component to the disk component turning it into an SSTable in Level 0. When the capacity of a level reaches a certain threshold, compaction is triggered to merge SSTables in this level with SSTables in the next deeper level to form a set of new SSTables at the deeper level. Since LSM-KVs adopt out-of-place updates, there can be invalidated redundant data in SSTables. Compaction has the effect of removing some of these invalid redundant data.

Compaction can be further divided into Level 0-Level 1 (L0-L1) compaction and deeper level compaction. L0-L1 compaction is unique in that this activity cannot be executed in parallel with other L0-L1 compaction activities. This is because the SSTables in L0 can have overlapping keys as they are directly copied from the Immutable Memtable. In contrast, all SSTables at each level for Level 1 and deeper never have overlapping key ranges. Hence, deeper level compactions can occur in parallel.

2.3 Write Stall Issue

Previous studies have shown that the following three types of write stalls occur in modern LSM-KVs.

Memtable (MT) stall: This stall occurs when the memory component becomes full. For example, RocksDB sets the number of Memtables to 2, and when both are filled up, the system input is simply stopped resulting in a stall. This is known to be the most common case of write stalls [17, 41, 45].

Level 0-Level 1 Compaction (L0) stall: Similarly to MT stall, LSM-KVs will slow down or even stop the input stream when the number of L0 files reaches the set threshold, resulting in stalls. In RocksDB, the default slow-down threshold is 20, while at 36 the input stream is stopped. This type of control first appeared in LevelDB [17] but has been adopted by most subsequent implementations that use a similar compaction strategy [4, 25, 49].

Table 1: Specifications of Storage Devices.

Device	Product Name	Device Capacity	Sequential Bandwidth
PM	Optane DC PMM	512 GB	2300 MB/s
NVMe SSD	Samsung 970 PRO	1 TB	2700 MB/s
SATA SSD	Intel DC S4500	960 GB	490 MB/s
SATA HDD	Seagate ST1000DM010	1 TB	210 MB/s

Pending Input Size (PS) stall: LSM-KVs also slow down or stop the system when the pending input size of compaction jobs exceeds a certain threshold. Note that the pending input size not only refers to repeated and out-of-date entries in an SSTable, but also includes all the entries within the SSTable that is pending compaction. In RocksDB, the default pending compaction input size threshold for slowing down and stopping the system is 64GB and 128GB, respectively. Prior studies have used the term “compaction pending bytes” for this value [25, 58], while, in this study, we use the term “redundant data” instead. The aim of this control is to reduce the total amount of redundant data [15, 26, 28] or to avoid disk bandwidth bursts when compaction jobs happen in deep levels [45, 50].

3 Observations from Previous Studies

In this section, we use experimental observations with RocksDB to revisit previous studies. While these earlier studies provide valuable insight into the causes of write stalls, we show that they are all limited in that these insights are not able to explain many of the experimental results.

3.1 Experimental Settings

The experiments in this section are conducted on a server that follows the recommended configuration for installing the Optane DC PMM. It has two Intel Xeon(R) Gold 6230 processors with 2.10GHz frequency with a total of 40 cores (20 cores each) and is equipped with 128GB DDR4 DRAM. We consider four different storage devices as listed in Table 1, which shows the device types, the producer and product names, the capacity of the devices, and the sequential bandwidth as specified by the manufacturer or reported in an earlier study [29–31, 33].

All experiments are run on Ubuntu 18.04 LTS, running RocksDB 6.11 [24] compiled with CMake 3.10.2. We run the *fillrandom* workload in db_bench [23] issuing uniformly distributed random writes in each scheme for one hour, a time period sufficient to trigger all kinds of write stalls and maintain a trend in all devices. Each entry consists of a 16-byte key and a 1000-byte value. All experiments are evaluated under peak throughput since write stalls occur only when the write pressure is high enough as shown in Figure 1. Write stalls are observed with an embedded event listener provided

Table 2: Summary of confirmations and limitations on conclusions made by existing studies on write stalls.

Original Conclusion	Points we confirm	Limitations we find
Resource Exhaustion [7, 15, 34, 43, 51, 58, 60]	[C1]: High CPU utilization is a source of write stalls. Increasing background threads reduces CPU utilization and hence, reduces write stalls [34,43,51,60]. [C2]: Most devices show increased bandwidth usage and decreased CPU utilization when increasing the number of threads. The occurrence of write stalls increases when the number of threads exceeds a certain threshold [15]. [C3]: As modern devices provide much higher bandwidth and parallelism, the stall occurrence and duration on PM and NVMe SSD are much lower than those on SATA devices [7,43,58].	[L1]: Continued increase beyond a certain number of threads results in a continued decrease of (normalized) CPU utilization, but results in an increase in write stall duration. That is, reduced CPU utilization does not result in reduced write stalls. [L2]: Even with high CPU utilization, simply by increasing the batch size, write stalls may be reduced. That is, CPU utilization and write stalls do not correlate. [L3]: Modern devices can provide far more bandwidth than conventional devices, but write stalls may still occur before its bandwidth capacity is reached.
L0-L1 Compaction Data Movement [7,58]	[C4]: At early phases of execution, performance troughs in NVMe SSD and PM match the occurrence of compaction [7,58].	[L4]: Correspondence between performance troughs and L0-L1 compaction jobs diminishes over time, especially in the multi-threaded environment.
Deep Level Compaction Data Movement [45,49,50]	[C5]: The processing rate of flush jobs decrease when more threads are spawned for compaction jobs [45,49,50].	[L5]: As the number of threads increases, the occurrence of PS stalls that are caused by slow compaction decreases.

with RocksDB. This listener provides basic information such as the total duration and the number of occurrences of each type of write stall. All experimental results obtained are the average of three rounds of executions; the three rounds take over 240 hours to execute.

For the experiments, we mainly consider the impact of two parameters that have a strong effect on performance [13–15, 26]. The first is the number of threads that run concurrently in the system, which determines the resources that are allocated to each thread such as CPU time and bandwidth. In RocksDB, in particular, by default, a quarter of the threads are allocated for flush jobs (rounded down), while the rest perform compaction. The second parameter is batch size, which is the size used for both Memtable and SSTable. This value is critical for analyzing the behavior of LSM-KVs because 1) it controls the scheduling pattern and input scale of background jobs [6,7,14,43] and 2) it affects the data distribution at the various levels [11,13,14,22].

3.2 Limitations of Existing Studies

In this section, we discuss the limitations of earlier studies regarding write stalls. As we shall show, these studies tend to analyze the causes of write stalls in LSM-KV stores from a single component perspective. Through experimental observations, which we discuss below and summarize in Table 2, we show that these conclusions cannot be fully generalized.

Resource Exhaustion: Some earlier studies conclude that write stalls are caused by bandwidth congestion [7,45], while others consider CPU limitation as the root cause [43,58,60]. We revisit these conclusions, starting with CPU utilization.

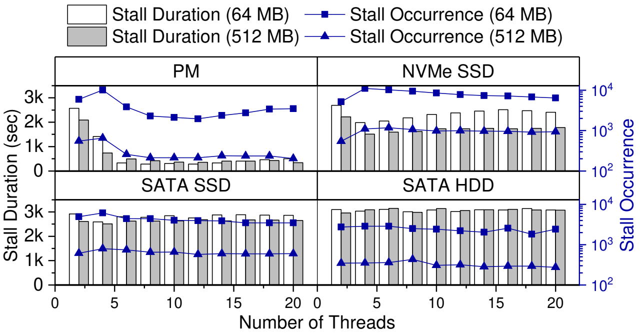


Figure 3: Duration and occurrences of write stalls as the number of threads is increased.

Figure 3 shows the stall occurrences and duration as the thread count increases. We observe that for PM and NVMe SSD, the stall duration decreases as the thread count increases (more notably with PM) until up to six threads. Also, as shown in Figure 4, up to six threads, the CPU utilization (normalized to the number of threads) remains relatively high for PM and NVMe SSD. Just based on these observations, one could conclude that the shortage of CPU resources, that is, high CPU utilization, is the cause for write stalls (Table 2 [C1]).

However, we also observe from Figure 3 that write stall occurrences start to drop, while the duration increases slightly, as the thread count increases beyond four (where CPU utilization decreases as shown in Figure 4) (Table 2 [L1]). This characteristic is particularly evident in the two advanced devices with higher bandwidth and parallelism. Based on these observations, our conclusion is that, while limited CPU capac-

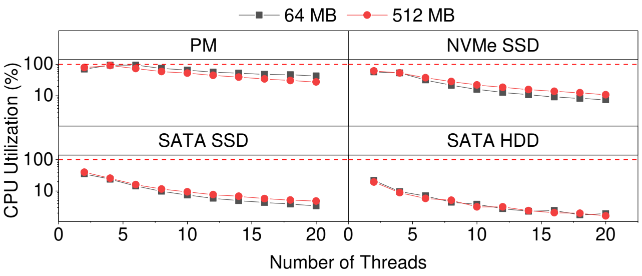


Figure 4: Comparison of normalized CPU utilization as threads are increased.

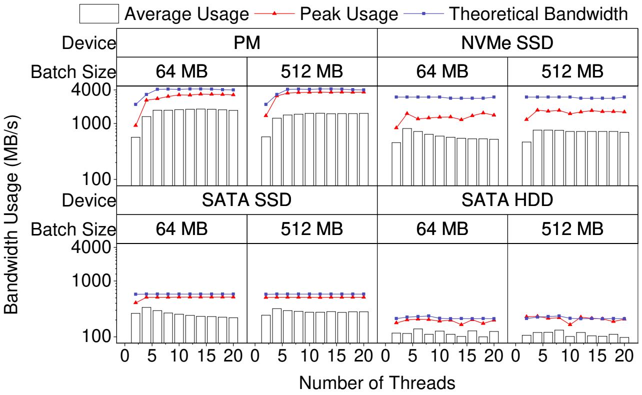


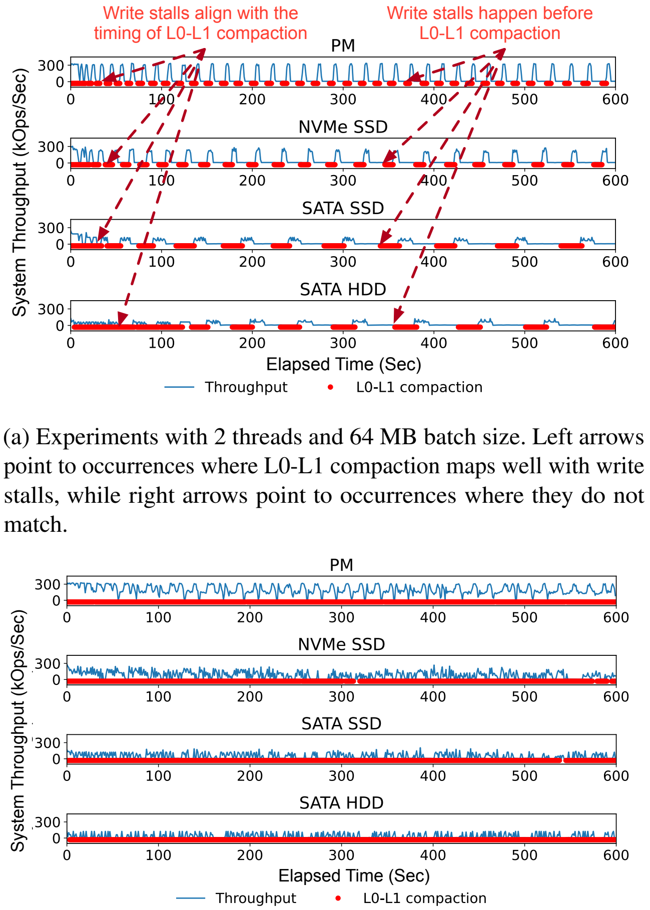
Figure 5: Comparison of bandwidth with an increasing number of threads and different batch sizes. (Note that y-axis is log scale.)

ity may be the cause of write stalls for some scenarios, this is difficult to generalize.

Moreover, when the batch size is increased from 64MB to 512MB in NVMe SSD and SATA SSD, even while CPU utilization does not show significant changes (Figure 4), we observe both duration and occurrence of write stalls decreases (Figure 3). In the PM case, we observe that beyond six threads, CPU utilization for 512MB batch size is lower than with 64MB, yet the stall duration is actually higher, except with over 15 threads and beyond. From these observations, we conclude that CPU utilization and write stall do not correlate well (Table 2 [L2]).

Other studies have pointed to the disk bandwidth limitation as the source of write stalls [7, 45, 58]. It is argued that with the increase in thread count the disk bandwidth will be overwhelmed leading to the stall problem (Table 2 [C2] and [C3]). However, the following observations showing that the system can be stalled even when disks are idle tell a different story.

Observe the theoretical bandwidth limit, which is the peak bandwidth observed when the device is flooded with requests generated from the FIO [2] tool with multiple threads, and the peak bandwidth used as the thread count increases in Figure 5. Although for the HDD and SATA SSD the peak reaches the theoretical bandwidth limit, for the NVMe SSD and PM, a large idle bandwidth gap remains, indicating that write stalls occur even if there is bandwidth to spare (Table 2 [L3]). In addition, if the write stall is due to insufficient bandwidth, devices should be under high write pressure for an elongated period. This should move the average bandwidth close to the theoretical value. However, we see in Figure 5 this is not so.



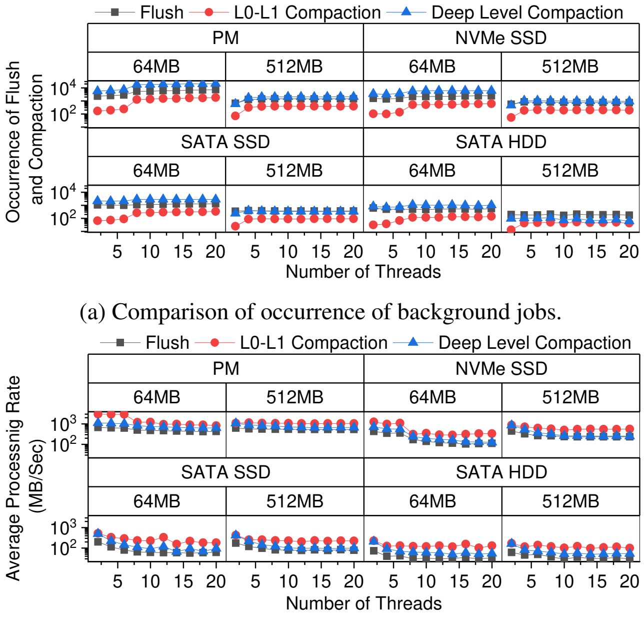
(a) Experiments with 2 threads and 64 MB batch size. Left arrows point to occurrences where L0-L1 compaction maps well with write stalls, while right arrows point to occurrences where they do not match.

(b) Experiments with 20 threads and 64 MB batch size. L0-L1 compaction is triggered much more frequently than those in (a), and the occurrences of write stalls show no relation with the L0-L1 compaction jobs.

Figure 6: Timing of L0-L1 compaction and throughput for the thread count of 2 and 20.

L0-L1 Compaction Data Movement: SSTables in L0 are unordered and their keys may overlap as they are generated by flush jobs. Thus, L0-L1 compaction, which takes all L0 files as its input, cannot be parallelized with other L0-L1 compaction jobs [16]. Former studies have taken this unique limitation as the direct cause of write stalls [7, 58].

We observe from Figure 6, which plots the process timing of L0-L1 compaction with instantaneous throughput, that this earlier conclusion is partially true. Specifically, when running with two threads (Figure 6(a)), initially, we observe system throughput dropping immediately as L0-L1 compaction is triggered, as designated by the dashed arrows on the left. This is the regularity observed by Yao et al. [58] (Table 2 [C4]). However, as the system continues to process the input stream, this correspondence disappears, as with the apparent misalign-



(a) Comparison of occurrence of background jobs.

(b) Average processing rate of background jobs.

Figure 7: Comparison of occurrences of background jobs (flush, L0-L1 compaction, and deep level compaction) and their average processing rate as the number of threads and batch size are increased, measured for one hour of execution.

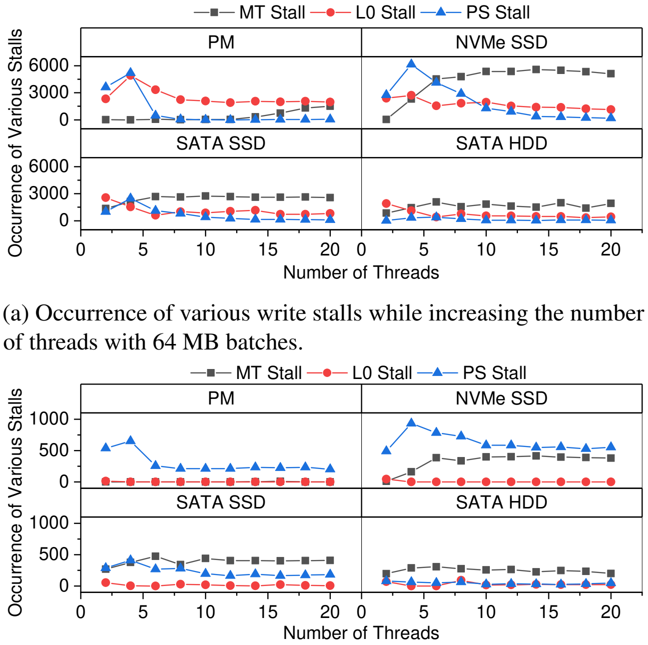
ments as designated by the dashed arrows on the right. Moreover, when the thread count is increased to 20, as shown in Figure 6(b), L0-L1 compaction occurs much more frequently than for the 2-threaded case, while showing no evident mapping relation between the timing of L0-L1 compaction and write stalls (Table 2 [L4]).

Deep Level Compaction Data Movement: Yet another set of earlier studies contend that the high resource consumption of deep level compaction jobs that are competing with other background jobs is the cause of write stalls [7, 45, 49, 50]. Again, this is partially true, as increasing the number of threads does lead to more frequent compaction jobs, as shown in Figure 7(a), and the average processing rate of background jobs decreases, as shown in Figure 7(b) (Table 2 [C5]).

However, if the conflicting compaction jobs were the main source of write stalls, we should have observed the occurrence of PS stalls increase just like how the slow flush rate increased MT stalls. Instead, as we observe in Figure 8, the occurrences of PS stall decrease as the number of threads increases and the batch size increases, which is contrary to the earlier conclusion.

4 Data Overflow

As seen from the previous section, earlier studies focus on individual aspects that could be the cause of write stalls. Our analysis of modern LSM-KVs reveals a more general source of write stalls, that is, what we refer to as *data overflow*. In this section, we explain the formation of data overflow and use data overflows to explain the limitations observed in Section 3.



(a) Occurrence of various write stalls while increasing the number of threads with 64 MB batches.

(b) Occurrence of various write stalls while increasing the number of threads with 512 MB batches.

Figure 8: Breakdown of various write stalls.

4.1 Data Overflow in Modern LSM-KVs

Data overflow refers to the rapid expansion of one or more components in an LSM-KV system due to a surge in data flow into one of the components. It happens when the processing rates of different background jobs do not match each other. We identify three types of data overflow, namely, Memory Overflow, Level 0 Overflow, and Redundancy Overflow, as shown in Figure 9. We now describe these in more detail.

Memory Overflow (MMO): MMO occurs when the system input rate surpasses the Immutable Memtable flush rate. Consequently, as the Immutable Memtable cannot be flushed in time, there will not be enough space in the memory component to absorb new data. In most modern LSM-KVs, upon MMO, the system stops receiving input as there is no room to buffer the incoming updates. This results in an MT stall.

Level 0 Overflow (L0O): L0O occurs when the processing rate of L0-L1 compaction is not able to match the flush rate. This results in the number of SSTables in Level 0 to rise. In modern LSM-KVs, the input stream is stopped or slowed when this value reaches a certain threshold so that the accumulated data may be consumed resulting in an L0 stall.

Redundancy Overflow (RDO): RDO occurs when the working efficiency of compaction threads cannot match the rate in which redundant data is generated. In modern LSM-KVs, when the size of redundant data reaches a threshold, the system will slow down or stop the input, and wait for the compaction threads to clear out the accumulated redundancy. Such action results in PS stalls.

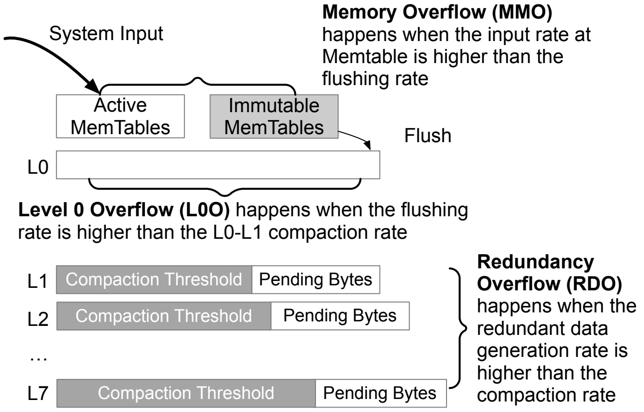

System Input
Memory Overflow (MMO) happens when the input rate at Memtable is higher than the flushing rate
Active MemTables
Immutable MemTables
Flush
L0
Level 0 Overflow (L0O) happens when the flushing rate is higher than the L0-L1 compaction rate
L1
Compaction Threshold
Pending Bytes
L2
Compaction Threshold
Pending Bytes
...
L7
Compaction Threshold
Pending Bytes
Redundancy Overflow (RDO) happens when the redundant data generation rate is higher than the compaction rate

Figure 9: Data overflow scenarios in modern LSM-KVs.

4.2 Explaining the Unexplained

In Section 3, we showed how conclusions made in earlier studies could not explain the reason behind write stalls for some portions of the extensive experimental results that we had obtained (Table 2). Here, we present how such incongruities can be explained with data overflow. To observe write stalls in this section, we make use of the LOG file that RocksDB provides. While the embedded listener used in the previous section provides only basic information, the LOG file provides more detailed information of each write stall, including the stall type, the limited input rate, and the exact timestamp.

Resource Exhaustion: First, consider how write stalls occurred under low utilization of resources (3.2, Table 2 [L1], [L2] and [L3]). The key underlying reason is that all three kinds of data overflow will stop or slow down the input before the system reaches the hardware limitation. Let us elaborate.

Firstly, when the flush rate is not high enough to persist the incoming requests in time, MMO will stop the input. Figure 7(b) shows that flush jobs are being allocated the least bandwidth among the background jobs and thus, the average flush rate monotonically decreases with the number of threads. This is because as the number of threads increases, more threads are forced to share the limited bandwidth, resulting in less bandwidth being allocated to the flush threads. This results in the Immutable Memtable not being flushed fast enough, which is the most common reason for write stalls that occur in SATA HDD as well as other devices when there are too many threads.

Secondly, L0O occurs as compaction of SSTables in L0 cannot keep pace with flush jobs, resulting in the number of SSTables in L0 reaching its threshold, and thus, the input stream being stopped or slowed. As direct evidence, Figure 10(a) shows how the occurrences of write stalls, marked by the vertical blue lines, correspond to the peak in the number of L0 SST files. Since L0-L1 compaction jobs are not being executed in parallel, increasing the number of threads will not help in reducing L0O as the processing rate does not increase. Hence, the system is stalled even when there is enough CPU resources.

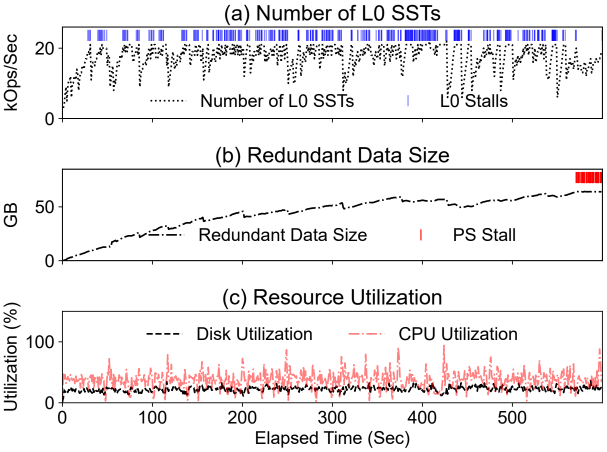


Figure 10: System statistics (first 600 seconds) when running the *fillrandom* workload with 4 background threads with NVMe SSD, which shows the most occurrences of write stalls.

Lastly, we show how RDO makes LSM-KVs stall when both CPU utilization is low and there is disk bandwidth to spare. Figure 10(b) shows how the redundant data tend to accumulate as the deep level compaction cannot keep pace with L0-L1 compaction. Eventually, RDO occurs when the redundant data size reaches 64GB, which is the default PS stall threshold, at which point the system slows the input, represented by the red lines. Note, however, that at these stall points, both the CPU and bandwidth utilization are low (and stable) as any other points in execution (Figure 10(c)), showing how RDO can occur despite low resource usage.

L0-L1 Compaction Data Movement: We next discuss why L0-L1 compaction does not align with the occurrence of write stalls (Table 2 [L4]). First, consider the misalignment in Figure 6(a), when there are only two threads. In the early stages of execution, as data in the deeper level have not been accumulated, L0-L1 compaction is easily assigned a thread. Hence, we see a nice alignment of the compaction with the write stall. However, as execution continues, data starts to accumulate and more deep level compaction requests get to be made. With only a limited number of threads, this dwindles the chance of L0-L1 compaction from being assigned a thread. Thus, L0-L1 compaction and write stalls start to misalign.

Now consider the situation when the number of threads is 20. Here, we have enough threads to always assign for L0-L1 compaction. However, with a large number of threads, bandwidth for flush jobs diminish, and thus, the processing rate of flush jobs becomes much lower (Figure 7(b)) causing more frequent MMO (Figure 8). That is, the frequent write stalls here are due to MT stalls, and L0O hardly occurs showing no relation to the L0-L1 compaction jobs as shown in Figure 6(b).

Deep Level Compaction Data Movement: Finally, while earlier studies concluded that PS stalls increase with thread count, which was true for up to four threads, we also saw

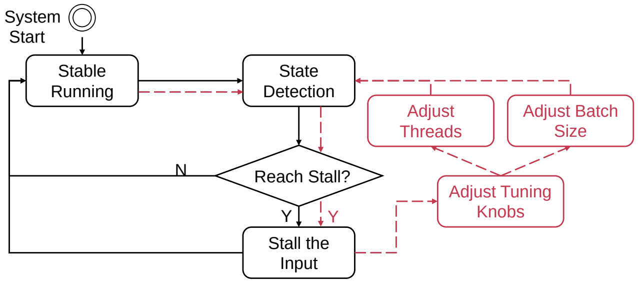


Figure 11: The solid black diagrams show the default control flow of RocksDB. ADOC extends this with the red diagrams (including the dashed arrows) to adjust tuning knobs during execution.

the opposite as more threads were added (Figure 8 and Table 2 [L5]). This can be explained with RDO. The initial spike in PS stalls (up to four threads) shown in Figure 8 is due to the flush threads obtaining sufficient bandwidth. In addition, with this increase to four, occurrences of L0O are decreased as more threads are allocated. This results in more data being crowded into the deeper levels and overwhelming the processing rate of deep level compaction jobs. This results in a significantly increase of PS stalls (Figure 8). However, beyond this threshold of 4 (for most cases), the flush rate is reduced and consequently, the data redundancy rate also diminishes. This results in less need to process deep level compaction jobs, which finally results in less RDO.

5 Automatic Data Overflow Control

The goal of our study is to minimize (and eventually remove) the effects of write stalls on LSM-KVs. To this end, we develop a framework, which we call **ADOC** (**A**utomatic **D**ata **O**verflow **C**ontrol), that controls the dataflow such that data overflow may be minimized. Dataflow is controlled by online tuning of the number of threads and the batch size as these values have a strong influence on dataflow, as seen from discussions in Section 4, and most LSM-KVs [15,48,52] provide APIs to adjust these two values without rebooting the system.

We develop ADOC under two principles. The first is device transparency. Instead of targeting optimizations to a particular storage device, as new storage devices will continue to evolve [35,54], ADOC should be able to tune itself to reduce write stalls irrespective of the underlying storage device. For this, ADOC monitors the flow of data amongst the components independent of the specific performance parameters of the underlying device. Then, the thread count and batch size are adjusted to control the processing rate and scheduling frequency of background jobs, thereby controlling the data flow within the LSM-KV.

The second principle is ease of portability. As shown in Figure 11, the design of ADOC is a straightforward extension of the RocksDB control flow mechanism. Unlike MatrixKV and similar approaches [14,45,50,58] that change the compaction strategy or SILK [7] and auto-tuned RocksDB [39] that adjust the internal thread scheduling and IO process, ADOC does not disrupt the internal architecture of the LSM-KV system making it highly portable.

In our current implementation in RocksDB, we make modifications to only two classes. One is the `Options` class, which controls whether the ADOC tuner will be enabled or not and records the instantaneous information of the system in a shared C++ vector. The other is the `tuner` class itself, which will periodically wake the tuner threads to perform tuning actions. In total, we add around 300 lines of code (LOC) to implement ADOC with 250 LOC for the tuner and 50 LOC to collect system states.

We now discuss the triggering mechanism and the actions that we employ to adjust these parameters. Recall that we identified data overflow as the source of write stalls. Thus, for every time window T_w, ADOC monitors for data overflow and takes action accordingly as explained below. In our current implementation, we set T_w to one second based on empirical observations; values larger are not agile enough to quickly detect the overflows leading to deteriorated performance for state-of-the-art high-performing storage devices, while values smaller could incur overhead as well as lead to fluctuations due to responding too quickly.

MMO: ADOC determines that MMO is occurring simply when the active Memtable is filled before the Immutable Memtable gets flushed. Upon MMO detection, ADOC increases the flush rate by reducing the number of threads, which will have the effect of reserving more bandwidth for the flush jobs, and increasing the batch size to increase the processing rate.

L0O: Determining whether L0O is occurring follows the same logic as RocksDB, that is, when the number of L0 files exceeds the threshold, which is 20 by default in RocksDB. Upon L0O detection, ADOC will increase the number of threads. This has the effect of improving the chance of L0-L1 compaction being assigned a thread and decreasing the flush rate to ease the overflow. The batch size, in this case, is unchanged as increasing it will increase the load on L0-L1 compaction, and decreasing it will generate more L0 files, both leading to more L0 stalls.

RDO: Like L0O, determining if RDO is occurring follows the same logic as RocksDB, that is, when the total redundant data size exceeds the threshold, which is 64GB by default in RocksDB. Upon detection of RDO, ADOC will increase the number of threads and decrease the batch size. The former is to increase the rate of deep level compaction (and also reduce flush rate) and the latter is to allow the scheduler to generate more fine-grained compaction jobs as small and dense compaction jobs can help improve the efficiency of redundancy reduction.

If multiple overflows are detected in T_w, we choose to handle the overflow in L0O, RDO, MMO order based on our experimental observations. Also, when turning the tuning knobs, we take the approach used by the Additive In-

Table 3: Schemes Evaluated

Name	Description
RocksDB-DF	RocksDB default setting
RocksDB-AT	RocksDB with auto-tuner on
SILK-D	SILK with RocksDB default setting
SILK-P	SILK setting set as in SILK paper [7]
SILK-O	SILK optimized to our setting (Section 3)
ADOC	RocksDB that enables ADOC tuner

crease/Multiplicative Decrease (AIMD) algorithm [9], best known for its use in TCP congestion control [40]. The reason for using AIMD is the fitness between the algorithm and ADOC's working scenario, that is, gently increasing the tuning knob to explore the suitable configuration and rapidly removing the over-allocated resources to avoid resource competition. In detail, we increase the number of threads by 2 and the batch size by 64MB, which is the default thread number and batch size value of RocksDB, while when decreasing, the values are reduced by half. After the adjustment, the number of threads that are allocated for flush jobs will also be adjusted to a quarter of the total number, just as the default setting.

6 Evaluation

6.1 Experiment Setups

Basic Settings: We use the same hardware and software setups as described in Section 3.1. For the basic setup, we follow that of SILK and set the maximum batch size to 512MB [7]. All schemes, including ADOC but excluding SILK, are based on RocksDB v7.5.3, the latest version as of this submission. SILK is built based on RocksDB 5.7.1 (early 2018) and our attempt to port SILK to more recent versions failed due to compatibility issues as considerable optimizations have been made since RocksDB v6 [27]. Thus, all performance measurements for SILK are done on RocksDB v5.7.1. In one of our experiments, we also show the results of ADOC ported on v5.7.1, which show some discrepancies with the results of v7.5.3, but overall, are quite similar in trend.

Schemes Compared: The schemes that we evaluate are as listed in Table 3. There are two settings of RocksDB, three settings of SILK, and ADOC. For RocksDB, we have RocksDB-DF with the default configuration, that is, two background working threads and 64MB batch size, and RocksDB-AT, an auto-tuner enabled version. RocksDB-AT automatically changes the threshold of the rate-limiter, which limits the number of IO operations generated by background threads [1] based on the IO pressure of background threads. RocksDB-AT also adjusts the allocation rate of each thread based on the thread priority to avoid starving compaction jobs, which has lower priority than flush jobs.

As for SILK, the three different configurations are as follows. The first is SILK that runs with the same configuration as the default configuration, which we refer to as SILK-D (D for default). The second is SILK-P (P for paper), which refers to SILK that runs with the same settings as mentioned in its original paper [7]. The configuration of SILK-P is of 4 background threads and 128MB batch size. The third configuration is SILK-O (O for optimal), which we believe to be the best performing setting in our experimental platform, with 8 background threads and 512MB batch size, which were obtained manually through exhaustive tuning attempts where we considered ten (2, 4, 8,..., 18, 20) thread and two (256MB and 512MB) batch configurations. We did not consider batch sizes 64MB and 128MB as when their results were observed for SILK-D and SILK-P, we found larger batches to be clearly better. In addition to the thread and batch size settings, the SILK implementation makes use of particular hard-coded settings. One is the allocation of bandwidth that is hard-coded into the db_bench tool. To faithfully configure our three versions accordingly, we set the configuration flags such that a quarter of the entire media bandwidth (Table 1) is reserved for compaction jobs while the rest is reserved for flush jobs. This ratio preserves the ratio used in the original paper. Additionally, as in the original implementation, we disable L0O by setting the number of L0 SST files threshold, which slows the input stream when reached, to an extremely large value.

6.2 Microbenchmark Performace

In this section, we make use of the same microbenchmark workload used in Section 3, that is, the db_bench random filling benchmark. We consider the three performance measures, namely, the throughput, the stall duration, and 99^{th} tail latency for the first 3600 seconds of execution.

Throughput: Figure 12 compares the system throughput for all the schemes that we consider. A few notable observations can be made as follows. First, ADOC shows the best performance over all devices. It shows 66.7%, 37.8%, 31.0%, and 55.1% higher average throughput over the next best performing scheme, SILK-O, for PM, NVMe SSD, SATA SSD, and SATA HDD devices, respectively. Second, the three variations of SILK show performance in SILK-O, SILK-P, and SILK-D order. Among these, the best performing SILK-O does considerably better for high-end devices, but not much so for low-end devices. Most notably, we observe that the configuration of SILK, the best of which is not straightforward to find, has a considerable effect on overall performance. Finally, RocksDB-DF and RocksDB-AT fare comparably with SILK-D and SILK-P, but worse than SILK-O. While RocksDB-AT automatically decides the bandwidth usage of different background jobs [39], the results show that this is insufficient in bringing out the best performance. Consequently, we find that RocksDB-AT performs better than RocksDB-DF for NVMe SSD and SATA SSD, which concurs with the fact that RocksDB is optimized for flash devices [11, 15, 25], but performs worse for PM and SATA HDD. It is also limited in that the user needs to provide the bandwidth information.

For clarity, hereafter, we omit the results for RocksDB-DF

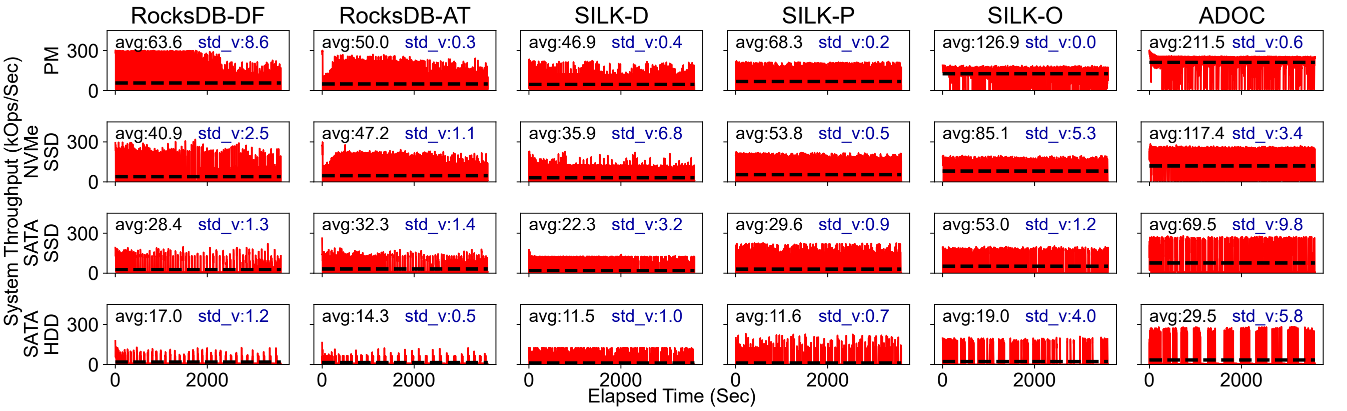


Figure 12: The solid red lines represent the instantaneous throughput during a single run of 3600 seconds, while the dotted black lines represent the average throughput of this run. The numbers shown within the box are the average throughput and the standard deviation of the three runs for each case.

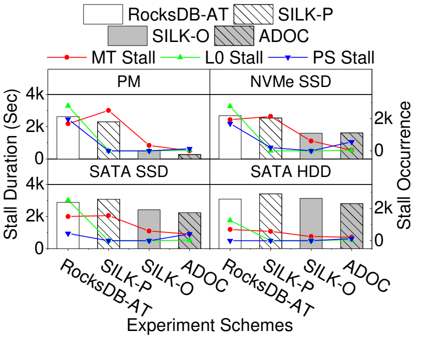


Figure 13: Stall duration (bar) and occurrences (lines).

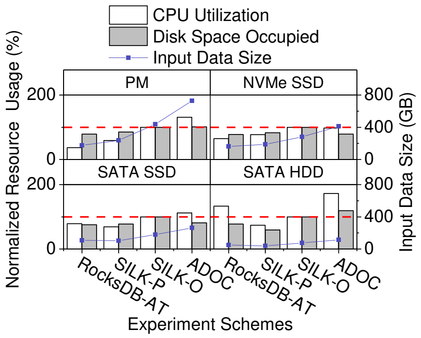


Figure 14: Comparison of CPU utilization, disk space occupied, and input data size with the *fillrandom* workload.

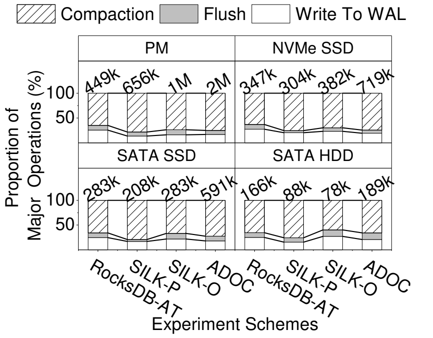


Figure 15: Proportion of major operation occurrences, with numbers representing total occurrences.

and SILK-D, the two low-performing schemes.

Stall Duration: Figure 13 compares the write stall duration for the various schemes. The bars indicate the total stall time, while the scattered lines mark the average occurrences of the three different sources of write stalls.

ADOC reduces the stall duration compared to SILK-O for PM, SATA SSD, SATA HDD by 45.2%, 8.7%, and 10.3%, respectively. However, for NVMe SSD, stalls seem to be elongated by 1.5%. Looking at the sources of the stalls, we observe that for MT and L0O stalls, ADOC is the lowest. For the PS stall, however, ADOC seems to be doing worse than both SILK schemes, seemingly negating the reduction of the other stalls. However, one must take into consideration the fact that these are measurements taken for the same 3600 seconds. Since the throughput of ADOC is considerably higher than the other schemes, ADOC takes in much more data from the input stream, specifically, 66.9%, 46.9%, 46.5%, 53.5% more data than SILK-O with PM, NVMe SSD, SATA SSD, and SATA HDD, respectively, as shown by 'Input Size' in Figure 14. This results in more data being accumulated into the deeper levels resulting in particularly higher PS stalls (Figure 13). Also, we observe that these additional PS stalls have a positive effect on the space amplification of the system. That is, the Disk Space Occupied results in Figure 14 show that despite ADOC processing a much higher volume of input data, the system does not occupy significantly more space compared to the other schemes. Even on the HDD, which has the worst compaction performance, ADOC accepts 53.5% more input data than SILK-O, yet the disk space occupied is only 19.4% larger. However, we also see from Figure 14 that this results in higher CPU utilization for ADOC. For example, with HDD, ADOC spends 72.5% more CPU time than SILK-O. Figure 15 shows the breakdown of the major operations for each scheme, with the total number of operations shown on top of each bar. These numbers were obtained by sampling the call stack of RocksDB using the *perf* [5, 18] tool at 99Hz sampling frequency. While not exact, these numbers provide an estimate of the CPU time spent for the operations as have been used in other studies [8, 19, 61], as the CPU time will be proportional to the operation count. We observe that the proportion of each operation is relatively stable for schemes on each device. However, we also observe that the operation count for ADOC is considerably higher than SILK-O for all devices, with the largest difference being 143.6% higher with HDD. This is because ADOC accepts more input data than the other schemes, with the additional inputs generating more

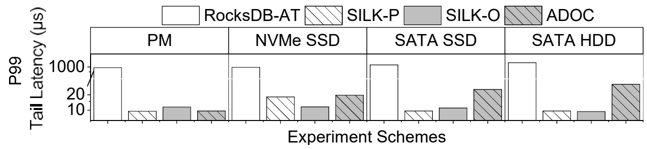


Figure 16: Average 99th tail latency in *fillrandom* workload.

Table 4: Data distribution and the composition of request types for the six YCSB workloads. (RMW: read-modify-write)

Workload	Distribution	Request Composition
A	Zipfian	50% Update 50% Read
B	Zipfian	95% Read 5% Update
C	Zipfian	100% Read
D	latest	5% Insert 95% Read
E	uniform	5% Insert 95 %Seek
F	Zipfian	50% Read 50% RMW

data movement jobs, and also because the occurrences are correlated to the CPU utilization of the related functions [18]. In addition, ADOC and SILK use different approaches to reserve bandwidth for flush jobs. SILK puts the compaction jobs to sleep when facing bandwidth congestion, while with ADOC there is at least one compaction job working at all times. As a result, ADOC shows higher CPU utilization and operation count than SILK.

Tail Latency: Figure 16 shows the 99th tail latency results obtained over 3600 seconds of execution. We observe that both SILK schemes do well in terms of tail latency, which was the target performance measure of SILK. Compared to SILK-O, ADOC does better for PM, but is higher by 70.1%, 131.2%, and 242.9% for NVMe SSD, SATA SSD, and SATA HDD, respectively. Again, however, we note that ADOC generates over twice the number of requests than SILK-P and 50% more than SILK-O, meaning that it faces edge cases (e.g., foreground GC, conflict I/O request in half duplex bus, etc.) much more often leading to higher tail latency.

6.3 Macro Benchmark

In this section, we consider real-world workloads using the YCSB benchmark. The YCSB benchmark [12] is a popular benchmark tool that generates workloads following real-world data characteristics. We run the six workloads with characteristics as shown in Table 4, executing them in the suggested order [12,49], that is, execute the loading stage first, followed by the run stages A, B, C, D, and F. Then, we reload the data and execute workload E. We load 50M entries (10B keys and 1000B values) during the load stage and then execute each run stage for one hour.

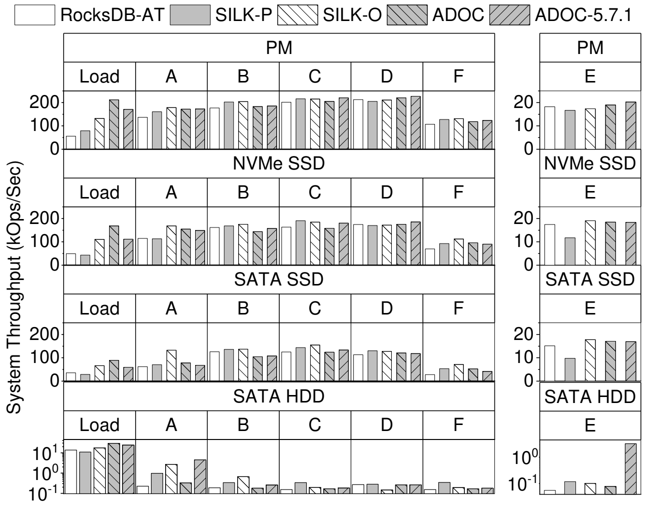


Figure 17: Comparison of system throughput in different stages of YCSB workloads.

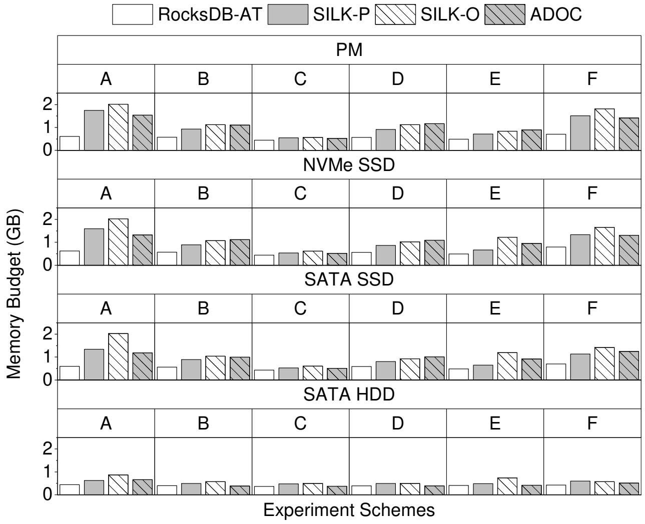


Figure 18: Comparison of main memory footprint.

Figure 17 shows the throughput of all schemes. (Note that there is a ADOC-5.7.1 scheme that has been added to the results. We elaborate on this later.) Comparing the auto-tuning systems, RocksDB-AT and ADOC, we find that the latter performs 7.6% to 11.4% better over all the devices considered. Comparing SILK-O and ADOC, the two gives-and-takes. For Load, a purely write workload, ADOC beats SILK-O. However, for workloads YCSB-A, -B, -C, and -F, SILK-O performs better. For YCSB-D and -E, the winner depends on the device. Recall that the RocksDB versions on which SILK-O and ADOC run differ. To remove the version effect, we also run ADOC on RocksDB 5.7.1. These results, denoted, ADOC-5.7.1 in Figure 17, show that the version difference has some effect on ADOC performance, with the older version performing better in the majority of run stages, most notably for YCSB-A and -E on the HDD.

The main reason SILK-O does well, despite the fact that ADOC performs considerably better than SILK-O for writes as was shown with the microbenchmarks, may be attributed to the large memory usage. As shown in Figure 18, we find that SILK-O uses as much as 76.8% more memory than ADOC. Larger memory allows more requests to be serviced from the

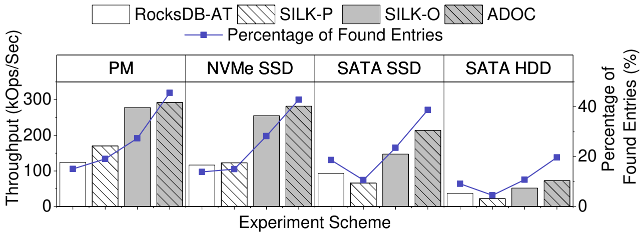


Figure 19: Throughput for the *read-while-writing* workload.

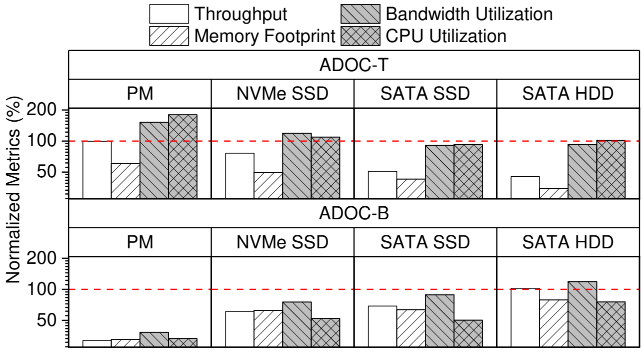


Figure 20: Comparison of system throughput and resource consumption with different tuning knobs triggered, the values shown are normalized to the value of ADOC.

data buffered in memory benefiting not only read-intensive but also update-intensive workloads like YCSB-A and -F.

Finally, the tail latency results (not shown) do not reveal any surprises; overall, SILK-O does best, but gives-and-takes between SILK-P and ADOC for particular workloads.

In conclusion, the performance results show that SILK-O and ADOC are comparable. However, recall that for SILK, performance varies considerably depending on the initial setting and that the "optimal" SILK-O setting was manually obtained. This is in contrast to ADOC being an online tuning system that does not need human intervention.

As a supplement to the macro benchmark, we also evaluate the system under the *read-while-writing* workload in db_bench that uses two threads, the reader and writer, to generate 50M write and read requests, respectively. Note that as the reader and writer are running concurrently, the reader thread may request entries that have not yet been persisted by the writer; we show the percentage of found entries along with the throughput of the various schemes in Figure 19. The results show that ADOC achieves 5.2% to 45.3% higher throughput than SILK-O, and 95% to 135% higher than RocksDB-AT.

6.4 Number of Threads versus Batch Size

Throughout our discussions, we considered the number of threads and batch size as our tuning parameters. In this section, we consider the effect of each parameter on ADOC. For this, we consider the *fillrandom* workload in Section 6.2 on two versions of ADOC, ADOC-T and ADOC-B, the former that only tunes the number of threads and the latter that only tunes the batch size.

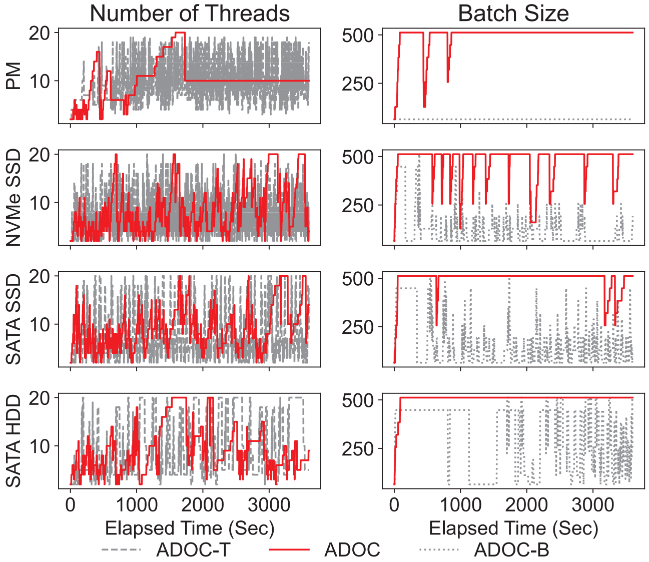


Figure 21: Tuning actions during one-hour execution.

Figure 20 shows the throughput and resource usage of the two schemes relative to ADOC initially set with the default values. Observing the throughput for ADOC-T, we see that it performs well for PM, but that the throughput deteriorates with slower devices. More specifically, since ADOC-T does not adjust the batch size, its memory footprint is smaller, but this comes at the price of performance. The disk bandwidth and CPU utilization are also high since the batch size is relatively small, resulting in more frequent data movement.

In contrast, we see that ADOC-B is almost the complete opposite. It performs well for HDD, but throughput deteriorates as the devices get more powerful. Since it does not increase the number of threads, the memory footprint, the CPU utilization, and the bandwidth utilization are all relatively low as fewer data movement jobs get triggered. That is, while resources are abundant, there are not enough workers to take advantage of them. However, for the HDD where bandwidth is limited, having few threads allow these threads to make full use of the bandwidth incurring less write stalls.

Figure 21 shows how the number of threads and the batch size are adjusted when only one parameter is considered with ADOC-T and ADOC-B, respectively, versus when both are considered with ADOC. The results show that tuning both parameters together reduces the frequency and fluctuations of the adjustments resulting in a much more stable setting.

In conclusion, we find that the number of threads and batch size have a complementing effect that stabilizes the tuning process, consequently resulting in better performance.

7 Related Work

Write Stall Issues: LSM-KV experiences periodical performance drops when facing heavy writing pressure due to write stalls. SILK focuses on the long tail latency problem caused by write stalls. To remedy this problem, it adjusts the priorities of background threads and adds rate limiters to these background threads [7]. It ensures the L0-L1 compaction will be handled in a timely manner to avoid disk overflow. Other studies propose new scheduling strategies to align the background compaction jobs in different levels and align the start time of each compaction according to the available resources [45, 50]. MatrixKV observes the shortcoming of the original SSTable format and points to the slow L0-L1 compaction as the root cause of write stalls with PM devices [58]. It redesigns the format of SSTables for NVM and proposes a new compaction scheme between the first two levels, which they call column-compaction. It also reduces the depth of LSM trees to reduce write amplification.

Deploying LSM-KVs on New Devices: Recent studies also focus on the design of LSM-KVs to fit in new storage devices, especially on PM. NoveLSM discusses several possible solutions that accelerate LSM-KV such as storing parts of the persisted component on PM [37]. It also proposes an in-place update solution to replace the compaction jobs in shallower levels. SLM-DB further takes advantage of PM by building a global B+-tree index in PM [36]. This global index helps in organizing the entire DB into a single level and uses selective compaction to reduce write amplification. ListDB deploys the entire LSM-KV in a DRAM-PM only system and considers the NUMA sensitivity of PM and the overhead of multiple copying of commit logs [38]. It develops a NUMA-aware skip list to replace SSTs. This saves the overhead of merge-sorting in Memtables and entry copying commit logs. It also uses in-place updates to reduce bandwidth utilization and the write amplification problem caused by conventional compaction jobs.

Other approaches to improve LSM-KV systems have been proposed. In particular, as new devices with much higher bandwidth and parallelism than conventional devices become prevalent, software overhead becomes more significant, and thus, approaches to reduce this overhead have been made. P2KVS notices the long waiting time on WAL lock [44]. It adds an accessing layer that batches the incoming requests and dispatches them into different KV instances, efficiently increasing the scalability of LSM-KVs on NVMe SSD. Studies such as KVell [43] and SpanDB [11] notice the high software overhead in conventional IO interfaces and replace the interfaces with more efficient ones like libasync or SPDK [57]. To further eliminate the high IO stack overhead, some of the studies try to reduce the duplicated operations between devices and the LSM-KV. FlashKV and LOCS use Open Channel SSDs (OCSSD) to directly control the IO process of LSM-KVs from the user-level [53, 59]. Other studies like KVSSD and iLSM try to integrate LSM-KVs and the FTL (Flash Translation Layer) of SSDs to bypass the IO stack and achieve lower operation latency [42, 55].

Parameter Tuning of LSM-KVs: There are studies that notice the performance of LSM-KVs can be strongly influenced by the configuration setting. Monkey extrapolates the worst-case scenarios for various operations and designs a configuration tuning framework to tune memory allocation policies and read/write performance for specific workloads based on these scenarios [13]. Dostoevsky further discusses the impact of different compaction strategies and develops a mixed compaction strategy that determines the input according to the input level [14]. Rafiki [47] and TiKV [21] use offline training methods such as Deep Neural Networks (DNN) to study the best setting combination according to the performance of the entire workload. They achieve better throughput but are limited to workload characteristics. Endure concludes that the tuning method on the worst cases is the Nominal Tuning Problem and provides a system that uses a robust tuning method to improve the tuning effect when facing uncertain workloads [22].

8 Conclusion

In this paper, we studied the write stall phenomena in LSM-KVs by revisiting earlier studies. We showed that the conclusions that focus on the individual aspects, though valid, are not generally applicable. Through a thorough review and further experiments on a modern LSM-KV, we showed that data overflow, which refers to the rapid expansion of one or more components in an LSM-KV system due to a surge in data flow into one of the components, is able to explain the formation of write stalls. Our contention was that by balancing and harmonizing data flow among components, we will be able to reduce data overflow and thus, write stalls.

We proposed a tuning framework called ADOC (Automatic Data Overflow Control) to adjust the system configurations rather than simply waiting for the overflowed data to be consumed as is done by default in RocksDB. Experimental results with RocksDB showed that ADOC improves throughput by as much as 322.8% compared with the auto-tuned RocksDB, which takes a similar auto-tuning approach to ADOC. Compared to the manually optimized state-of-the-art SILK [4], ADOC achieves up to 66% higher throughput for the synthetic write-intensive workloads, while achieving comparable performance for the real-world YCSB workloads. However, SILK attains this performance at the expense of using 22.2% more main memory on average.

Acknowledgement

We would like to thank our shepherd Sudarsun Kannan and the anonymous reviewers for their constructive comments that helped improve the paper. This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2022-2021-0-01817) supervised by the IITP (Institute for Information Communications Technology Planning Evaluation), and also partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CityU 11217020).

References

[1] Auto-tuned rate limiter. http://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html. Accessed on 2022-12-29.

[2] fio - flexible i/o tester. https://fio.readthedocs.io/en/latest/fio_doc.html. Accessed on 2022-01-07.

[3] Github - supermt/feat_7.11. https://github.com/supermt/FEAT_7.11. Accessed on 2022-01-07.

[4] HyperLevelDB: A fork of LevelDB intended to meet the needs of HyperDex while remaining compatible with LevelDB. https://github.com/rescrv/HyperLevelDB. Accessed on 2022-09-09.

[5] Perf wiki. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed on 2022-12-29.

[6] 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 2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, 2017.

[7] 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 2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[8] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. Midas: Systematic Kernel TOCTTOU Protection. In *Proceedings of 31st USENIX Security Symposium (USENIX Security 22)*, pages 107–124, Boston, MA, August 2022. USENIX Association.

[9] Lin Cai, Xuemin Shen, Jianping Pan, and Jon W. Mark. Performance Analysis of TCP-friendly AIMD Algorithms for Multimedia Applications. *IEEE Transactions on Multimedia*, 7(2):339–355, 2005.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[11] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *Proceedings of 19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 17–32, 2021.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. page 143–154, New York, NY, USA, 2010.

[13] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, page 79–94, New York, NY, USA, 2017.

[14] 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 *Proceedings of the 2018 International Conference on Management of Data*, page 505–520, New York, NY, USA, 2018.

[15] 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 19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, 2021.

[16] Facebook. Leveled Compaction. https://github.com/facebook/rocksdb/wiki/Leveled-Compaction. Accessed on 2022-09-09.

[17] Sanjay Ghemawat and Jeff Dean. Google: Leveldb. https://github.com/google/leveldb, 2022. Accessed on 2022-09-09.

[18] Brendan Gregg. The flame graph: This visualization of software execution is a new necessity for performance profiling and debugging. *Queue*, 14(2):91–110, mar 2016.

[19] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the idiosyncrasies of real persistent memory. *Proc. VLDB Endow.*, 14(4):626–639, feb 2021.

[20] Tom's Hardware. Intel Kills Optane Memory Business, Pays $559 Million Inventory Write-Off. https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good. Accessed on 2022-09-09.

[21] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.*, 13(12):3072–3084, aug 2020.

[22] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty, 2021.

[23] Facebook Inc. Benchmarking tools | RocksDB. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools. Accessed on 2022-09-09.

[24] Facebook Inc. facebook/rocksdb. https://github.com/facebook/rocksdb/tree/v6.11.4. Accessed on 2022-09-09.

[25] Facebook Inc. RocksDB | A Persistent Key-Value store | RocksDB. https://rocksdb.org/. Accessed on 2022-09-09.

[26] Facebook Inc. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide. Accessed on 2022-09-09.

[27] Facebook Inc. rocksdb/HISTORY.MD. https://github.com/facebook/rocksdb/blob/main/HISTORY.md. Accessed on 2022-09-09.

[28] Facebook Inc. Write Stalls. https://github.com/facebook/rocksdb/wiki/Write-Stalls. Accessed on 2022-09-09.

[29] Intel Inc. Intel SSD DC S4500 Series 960GB 2.5in SATA 6Gbs 3D1 TLC Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/series/120535/intel-ssd-dc-s4500-series.html. Accessed on 2022-09-09.

[30] Samsung Inc. 970 PRO | Consumer SSD | Samsung Semiconductor. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970pro/. Accessed on 2022-09-09.

[31] Seagate Inc. BarraCuda Hard Drives. https://www.seagate.com/products/hard-drives/barracuda-hard-drive/, 2022. Accessed on 2022-09-09.

[32] InfluxData. InfluxDB: Purpose-Built Open Source Time Series Database | InfluxData. https://www.influxdata.com/. Accessed on 2022-09-09.

[33] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Soh, Zixuan Wang, Yi Xu, Subramanya Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. https://arxiv.org/abs/1903.05714, 03 2019.

[34] Yichen Jia and Feng Chen. From Flash to 3D XPoint: Performance Bottlenecks and Potentials in RocksDB with Storage Evolution. In *Proceedings of 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 192–201, 2020.

[35] Myoungsoo Jung. Hello Bytes, Bye Blocks: PCIe Storage Meets Compute Express Link for Memory Expansion (CXL-SSD). HotStorage '22, page 45–51, 2022.

[36] 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 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, 2019.

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

[38] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, Carlsbad, CA, July 2022. USENIX Association.

[39] Andrew Kryczka. Auto-tuned Rate Limiter | RocksDB. https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html. Accessed on 2022-09-09.

[40] Chengdi Lai, Ka-Cheong Leung, and Victor O.K. Li. Design and analysis of TCP AIMD in wireless networks. In *Proceedings of Wireless Communications and Networking Conference (WCNC)*, pages 1422–1427, 2013.

[41] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

[42] Chang-Gyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Youngjae Kim, Jungki Noh, Woosuk Chung, and Kyoung Park. ilsm-ssd: An intelligent lsm-tree based key-value ssd for data analytics. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 384–395, 2019.

[43] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. KVell: the Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the*

27th ACM Symposium on Operating Systems Principles, pages 447–461, 2019.

[44] Ziyi Lu, Qiang Cao, Hong Jiang, Shucheng Wang, and Yuanyuan Dong. P<sup>2</sup>KVS: A Portable 2-Dimensional Parallelizing Framework to Improve Scalability of Key-Value Stores on SSDs. page 575–591, 2022.

[45] Chen Luo and Michael J Carey. On performance stability in LSM-based storage systems (extended version). https://arxiv.org/abs/1906.09667, 2019.

[46] Chen Luo and Michael J Carey. LSM-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[47] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference(Middleware '17)*, page 28–40, 2017.

[48] Yoshinori Matsunobu, Siying Dong, and Herman Lee. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[49] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles(SOSP '17)*, page 497–514, 2017.

[50] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 217–228, 2012.

[51] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. FPGA-based Compaction Engine for Accelerating LSM-tree Key-Value Stores. In *Proceedings of 2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1261–1272, 2020.

[52] Mehul Nalin Vora. Hadoop-HBase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, pages 601–605, 2011.

[53] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[54] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *Proceedings of 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[55] Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang. KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 563–568, 2018.

[56] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of 18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, 2020.

[57] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. Spdk: A development kit to build high performance storage applications. In *Proceedings of 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, 2017.

[58] 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 2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31, 2020.

[59] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. Flashkv: Accelerating kv performance with open-channel ssds. *ACM Trans. Embed. Comput. Syst.*, 16(5s), sep 2017.

[60] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *Proceedings of 18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 225–237, Santa Clara, CA, 2020.

[61] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *Proceedings of 2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, Carlsbad, CA, July 2022. USENIX Association.

FUSEE: A Fully Memory-Disaggregated Key-Value Store

Jiacheng Shen[1]*, Pengfei Zuo[2], Xuchuan Luo[3], Tianyi Yang[1],
Yuxin Su[4], Yangfan Zhou[3], and Michael R. Lyu[1]

[1]*The Chinese University of Hong Kong,* [2]*Huawei Cloud,* [3]*Fudan University,* [4]*Sun Yat-sen University*

Abstract

Distributed in-memory key-value (KV) stores are embracing the disaggregated memory (DM) architecture for higher resource utilization. However, existing KV stores on DM employ a *semi-disaggregated* design that stores KV pairs on DM but manages metadata with monolithic metadata servers, hence still suffering from low resource efficiency on metadata servers. To address this issue, this paper proposes FUSEE, a ***FU****lly memory-di****S****aggr****E****gated* KV Stor***E*** that brings disaggregation to metadata management. FUSEE replicates metadata, *i.e.*, the index and memory management information, on memory nodes, manages them directly on the client side, and handles complex failures under the DM architecture. To scalably replicate the index on clients, FUSEE proposes a client-centric replication protocol that allows clients to concurrently access and modify the replicated index. To efficiently manage disaggregated memory, FUSEE adopts a two-level memory management scheme that splits the memory management duty among clients and memory nodes. Finally, to handle the metadata corruption under client failures, FUSEE leverages an embedded operation log scheme to repair metadata with low log maintenance overhead. We evaluate FUSEE with both micro and YCSB hybrid benchmarks. The experimental results show that FUSEE outperforms the state-of-the-art KV stores on DM by up to 4.5 times with less resource consumption.

1 Introduction

Traditional in-memory key-value (KV) stores on monolithic servers have recently been ported to the disaggregated memory (DM) architecture for better resource efficiency [60, 73]. Compared with monolithic servers, DM decouples the compute and memory resources into independent network-attached compute and memory pools [3, 23, 25, 38, 47, 54, 55, 65]. KV stores on DM can thus enjoy efficient resource pooling and have higher resource efficiency.

However, constructing KV stores on DM is challenging because the memory pool generally lacks the compute power to manage data and metadata. Existing work [60] proposes a *semi-disaggregated* design that stores KV pairs in the disaggregated memory pool but retains metadata management on monolithic servers. In such a design, the KV pair storage enjoys high resource utilization due to exploiting the DM architecture, but the metadata management does not. Many additional resources are exclusively assigned to the metadata servers in order to achieve high overall throughput [13, 53, 69].

To achieve full resource utilization, it is critical to bring disaggregation to the metadata management, *i.e.*, building a *fully memory-disaggregated* KV store. The metadata, *i.e.*, the index and memory management information, should be stored in the memory pool and directly managed by clients rather than metadata servers. However, it is non-trivial to achieve a fully memory-disaggregated KV store due to the following challenges incurred from handling complex failures and the weak compute power in the memory pool.

1) Client-centric index replication. To tolerate memory node failures, clients need to replicate the index on memory nodes in the memory pool and guarantee the consistency of index replicas. In existing replication approaches, *e.g.*, state machine replication [33, 46, 50, 62] and shared register protocols [5, 7, 43], the replication protocols are executed by server-side CPUs. These protocols cannot be executed on DM due to the weak compute power in the memory pool. Meanwhile, if clients simply employ consensus protocols [36, 46, 50] or remote locks [60], the KV store suffers from poor scalability due to the explicit serialization of conflicting requests [4, 11, 64, 70].

2) Remote memory allocation. Existing semi-disaggregated KV stores manage memory spaces with monolithic metadata servers. However, in the fully memory-disaggregated setting, such a server-centric memory management scheme is infeasible. Specifically, memory nodes cannot handle the compute-heavy fine-grained memory allocation for KV pairs due to their poor compute power [25, 60]. Meanwhile, clients cannot efficiently allocate memory spaces because multiple RTTs are required to modify the memory management information stored on memory nodes [38].

3) Metadata corruption under client failures. In semi-disaggregated KV stores, client failures do not affect metadata because the CPUs of monolithic servers exclusively modify metadata. However, clients directly access and modify metadata on memory nodes in the fully memory-disaggregated setting. As a result, client failures can leave partially modified metadata accessible by others, compromising the correctness of the entire KV store.

To address these challenges, we propose FUSEE, a fully memory-disaggregated key-value store that has efficient index replication, memory allocation, and fault-tolerance on DM.

*Work mainly done during the internship at Huawei Cloud.

First, to maintain the strong consistency of the replicated index in a scalable manner, FUSEE proposes the SNAPSHOT replication protocol. The key to achieving scalability is to resolve write conflicts without involving the expensive request serialization [7]. SNAPSHOT adopts three simple yet effective conflict-resolution rules on clients to allow conflicts to be resolved collaboratively among clients instead of sequentially. Second, to achieve efficient remote memory management, FUSEE employs a two-level memory management scheme that splits the server-centric memory management process into compute-light and compute-heavy tasks. The compute-light coarse-grained memory blocks are managed by the memory nodes with weak compute power, and the compute-heavy fine-grained objects are handled by clients. Finally, to deal with the problem of metadata corruption, FUSEE adopts an embedded operation log scheme to resume clients' partially executed operations. The embedded operation log reuses the memory allocation order and embeds log entries in KV pairs to reduce the log-maintenance overhead on DM.

We implement FUSEE from scratch and evaluate its performance using both micro and YCSB benchmarks [15]. Compared with Clover and pDPM-Direct [60], two state-of-the-art KV stores on DM, FUSEE achieves up to 4.5 times higher overall throughput and exhibits lower operation latency with less resource consumption. The code of FUSEE is available at `https://github.com/dmemsys/FUSEE`.

In summary, this paper makes the following contributions:

- A fully memory-disaggregated KV store with disaggregated metadata and data that is resilient to failures on DM.
- A client-centric replication protocol that uses conflict resolution rules to enable clients to resolve conflicts collaboratively. The protocol is formally verified with TLA+ [35] for safety and the absence of deadlocks under crash-stop failures.
- A two-level memory management scheme that leverages both memory nodes and clients to efficiently manage the remote memory space.
- An embedded operation log scheme to repair the corrupted metadata with low log maintenance overhead.
- The implementation and evaluation of FUSEE to demonstrate the efficiency and effectiveness of our design.

2 Background and Motivation

2.1 The Disaggregated Memory Architecture

The disaggregated memory architecture is proposed to address the resource underutilization issue of traditional datacenters composed of monolithic servers [25, 38, 47, 54, 55, 65]. DM separates CPUs and memory of monolithic servers into two independent hardware resource pools containing compute nodes (CNs) and memory nodes (MNs) [55, 60, 64, 73]. CNs have abundant CPU cores and a small amount of memory as local caches [64]. MNs host various memory media, *e.g.*, DRAM and persistent memory, to accommodate different application requirements with weak compute power. CPUs in CNs directly access memory in MNs with fast remote-access interconnect techniques, such as one-sided RDMA (remote direct memory access), Omni-path [16], CXL [42], and Gen-Z [14]. Each MN provides `READ`, `WRITE`, and atomic operations, *i.e.*, compare-and-swap (`CAS`) and fetch-and-add (`FAA`), for CNs to access memory data. Besides, MNs own limited compute power (*e.g.*, 1-2 CPU cores) to manage local memory and establish connections from CNs, providing CNs with the `ALLOC` and `FREE` memory management interfaces. Without loss of generality, in this paper, we consider CNs accessing MNs using one-sided RDMA verbs.

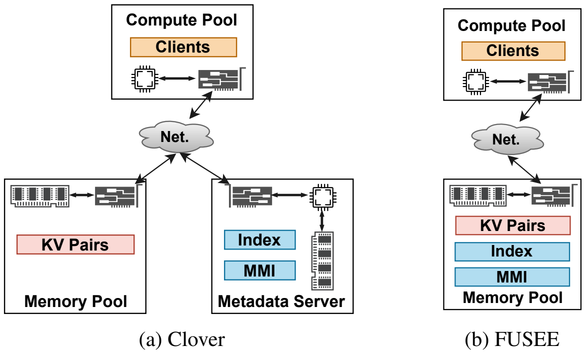


Figure 1: Two architectures of memory-disaggregated KV stores. (a) The semi-disaggregated architecture (Clover [60]). (b) The fully disaggregated architecture proposed in this paper.

2.2 KV Stores on Disaggregated Memory

Clover [60] is a state-of-the-art KV store built on DM. It adopts a semi-disaggregated design that separates data and metadata to lower the ownership cost and prevent the compute power of data nodes from becoming the performance bottleneck. As shown in Figure 1a, Clover deploys clients on CNs and stores KV pairs on MNs. It adopts additional monolithic metadata servers to manage the metadata, including *memory management information (MMI)* and the *hash index*. For `SEARCH` requests, clients look up the addresses of the KV pairs from metadata servers and then fetch the data on MNs using `RDMA_READ` operations. For `INSERT` and `UPDATE` requests, clients allocate memory blocks from metadata servers with RPCs, write KV pairs to MNs with `RDMA_WRITE` operations, and update the hash index on the metadata servers through RPCs. To prevent clients' frequent requests from overwhelming the metadata servers, clients allocate a batch of memory blocks one at a time and cache the hash index locally. As a result, Clover achieves higher throughput under read-intensive workloads with less resource consumption.

However, the semi-disaggregated design of Clover cannot fully exploit the resource efficiency of the DM architecture due to its monolithic-server-based metadata management. On the one hand, monolithic metadata servers consume additional resources, including CPUs, memory, and RNICs. On the other hand, many compute and memory resources have to

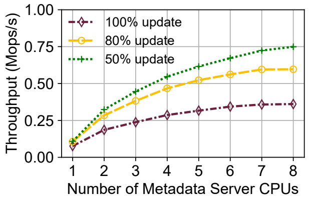


Figure 2: The throughput of Clover with an increasing number of metadata server CPUs.

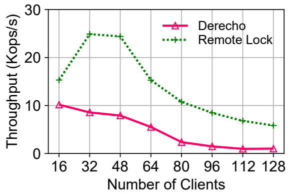


Figure 3: The throughput of Derecho [27] and lock-based approaches.

be reserved and assigned to the metadata server of Clover to achieve good performance due to the CPU-intensive nature of metadata management [13, 53, 69]. To show the resource utilization issue of Clover, we evaluate its throughput with 2 MNs, 64 clients, and a metadata server with different numbers of CPU cores. We control the number of CPU cores by assigning different percentages of CPU time with cgroup [10]. As shown in Figure 2, Clover has a low overall throughput with a small number of CPU cores assigned to its metadata server. At least six additional cores have to be assigned until the metadata server is no longer the performance bottleneck.

To attack the problem, FUSEE adopts a *fully memory-disaggregated* design that enables clients to directly access and modify the hash index and manage memory spaces on MNs, as shown in Figure 1b. Compared with the semi-disaggregated design, resource efficiency can be improved because client-side metadata management eliminates the additional metadata servers. The overall throughput can also be improved because the computation bottleneck of metadata management no longer exists.

3 Challenges

This section introduces the three challenges of constructing a fully memory-disaggregated KV store, *i.e.*, index replication, remote memory allocation, and metadata corruption.

3.1 Client-Centric Index Replication

The index must be replicated to tolerate MN failures. Strong consistency, *i.e.*, linearizability [26], is the most commonly adopted correctness standard for data replication because it reduces the complexity of implementing upper-level applications [1, 7, 12]. Linearizability requires that operations on an object appear to be executed in some total order that respects the operations' real-time order [26]. The key challenge of achieving a linearizable replicated hash index under the fully memory-disaggregated setting comes from the client-centric computation nature of DM.

First, existing replication methods are not applicable in the fully memory-disaggregated setting due to their server-centric nature. State machine replication (SMR) [33, 44, 46, 49, 50, 59, 62] and shared register protocols [7, 43] are two major replication approaches that achieve linearizability. However, both approaches are designed with a server-centric assumption that a data replica is exclusively accessed and modified by the CPU that manages the data. First, the SMR approaches consider the CPU and the data replica as a state machine and achieve strong consistency by forcing the state machines to execute deterministic KV operations in the same global order [49, 50]. Server CPUs are extensively used to reach a consensus on a global operation order and apply state transitions to data replicas. Second, shared register protocols view the CPU and the data replica as a shared register with `READ` and `WRITE` interfaces. Linearizability is achieved with a last-writer-wins conflict resolution scheme [43] that forces a majority of shared registers to always hold data with the newest timestamps. Shared register protocols also heavily rely on server-side CPUs to compare timestamps and apply data updates. The challenge with the server-centric approaches is that in the fully memory-disaggregated scenario, there is no such management CPU because all clients directly access and modify the hash index with one-sided RDMA verbs.

Second, naively adopting consensus protocols or remote locks among clients results in poor throughput due to the expensive request serialization. To show the performance issues of consensus protocols and remote locks, we store and replicate a shared object on two MNs and vary the number of concurrent clients. We use a state-of-the-art consensus protocol Derecho [27] and an RDMA `CAS`-based spin lock to ensure the strong consistency of the replicated object. As shown in Figure 3, both Derecho and lock-based approaches exhibit poor overall throughput and cannot scale with the growing number of concurrent clients.

3.2 Remote Memory Allocation

The key challenge of managing DM is where to execute the memory-management computation. There are two possible DM management approaches [38], *i.e.*, compute-centric ones and memory-centric ones. The compute-centric approaches store the memory management metadata on MNs and allow clients to allocate memory spaces by directly modifying the on-MN metadata. Since the memory management metadata are shared by all clients, clients' accesses have to be synchronized. As a result, compute-centric approaches suffer from the high memory allocation latency incurred by the expensive and complex remote synchronization process on DM [38]. The memory-centric approaches maintain all memory management metadata on MNs with their weak compute power. Such approaches are also infeasible because the poor memory-side compute power can be overwhelmed by the frequent fine-grained KV allocation requests from clients. Although there are several approaches that conduct memory management on DM, they all target page-level memory allocation and rely on special hardware, *i.e.*, programmable switches [38] and SmartNICs [25], which are orthogonal to our problem.

3.3 Metadata Corruption

In fully memory-disaggregated KV stores, crashed clients can leave partially modified metadata accessible by other

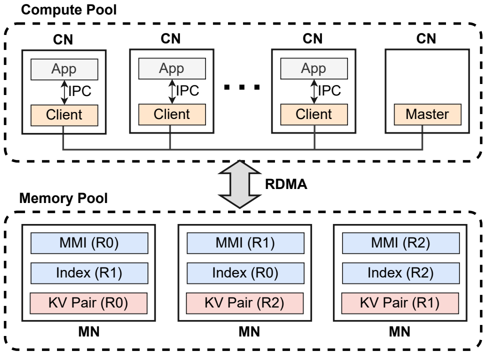


Figure 4: The FUSEE overview *(MMI, Index, and KV pairs have multiple replicas, i.e., R_0, R_1, and R_2. R_0 is the primary replica.).*

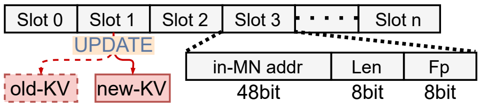


Figure 5: The structure of an index replica.

healthy clients. Since the metadata contains important system state, metadata corruption compromises the correctness of the entire KV store. First, crashed clients may leave the index in a partially modified state. Other healthy clients may not be able to access data or even access wrong data with the corrupted index. Second, crashed clients may allocate memory spaces but not use them, causing severe memory leakage. Hence, in order to ensure the correctness of the KV store, the corrupted metadata has to be repaired under client failures.

4 The FUSEE Design

4.1 Overview

As shown in Figure 4, FUSEE consists of clients, MNs, and a master. Clients provide `SEARCH`, `INSERT`, `DELETE`, and `UPDATE` interfaces for applications to access KV pairs. MNs store the replicated memory management information (MMI), hash index, and KV pairs. The master is a cluster management process responsible only for initializing clients and MNs and recovering data under client and MN failures.

FUSEE replicates both the hash index and KV pairs to tolerate MN failures. We adopt RACE hashing (Section 4.2) to index KV pairs and propose the SNAPSHOT replication protocol to enforce the strong consistency of the replicated hash index (Section 4.3). A two-level memory management scheme is adopted to efficiently allocate and replicate variable-sized KV pairs (Section 4.4). FUSEE uses logs to handle the corrupted metadata under client failures and adopts an embedded operation log scheme to reduce the log maintenance overhead (Section 4.5). Other optimizations are introduced in Section 4.6 to further improve the system performance.

4.2 RACE Hashing

RACE hashing is a one-sided RDMA-friendly hash index. As shown in Figure 5, it contains multiple 8-byte slots, with each storing a pointer referring to the address of a KV pair, an 8-bit fingerprint (Fp), *i.e.*, a part of the key's hash value, and the length of the KV pair (Len) [73]. For `SEARCH` requests, a client reads the slots of the hash index according to the hash value of the target key and then reads the KV pair on MNs according to the pointer in the slot. For `UPDATE`, `INSERT`, and `DELETE` requests, RACE hashing adopts an *out-of-place modification* scheme. It first writes a KV pair to MNs and then modifies the corresponding slot in the hash index to the address of the KV pair atomically with an `RDMA_CAS`. Nevertheless, RACE hashing only supports a single replica.

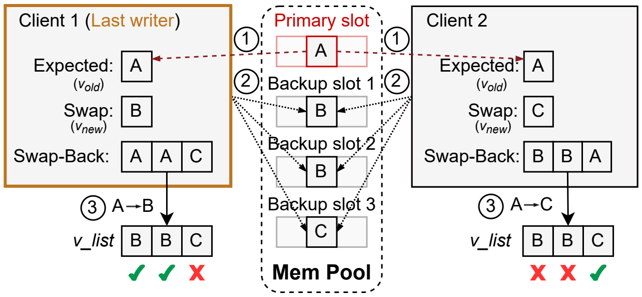


Figure 6: The SNAPSHOT replication protocol.

4.3 The SNAPSHOT Replication Protocol

In FUSEE, multiple clients concurrently read or write the same slot in the replicated hash index to execute `SEARCH` or `UPDATE` requests, as shown in Figure 6. To efficiently maintain the strong consistency of slot replicas in the replicated hash index, FUSEE proposes the SNAPSHOT replication protocol, a client-centric replication protocol that achieves linearizability without the expensive request serialization.

There are two main challenges to efficiently achieving linearizability under the fully memory-disaggregated setting. First, how to protect readers from reading incomplete states during read-write conflicts. Second, how to resolve write-write conflicts without expensively serializing all conflicting requests. To address the first challenge, SNAPSHOT splits the replicated hash index into a single primary replica and multiple backup replicas and uses backup replicas to resolve write conflicts. Hence, incomplete states during write conflicts only appear on backup replicas and the primary replica always contains the correct and complete value. Readers can simply read the contents in the primary replica without perceiving the incomplete states. To address the second challenge, SNAPSHOT adopts a last-writer-wins conflict resolution scheme similar to shared register protocols. SNAPSHOT leverages the *out-of-place modification* characteristic of RACE hashing that conflicting writers always write different values into the same slot because the values are pointers referring to KV pairs at different locations. Three conflict-resolution rules are thus defined based on the values written by conflicting writers in backup replicas, which enable clients collaboratively to decide on a single last writer under write conflicts.

Algorithm 1 shows the `READ` and `WRITE` processes of the SNAPSHOT replication protocol. Here we focus on the execution of SNAPSHOT when no failure occurs and leave the

Algorithm 1 The SNAPSHOT replication protocol

```
 1: procedure READ($slot$)
 2:     $v$ = RDMA_READ_primary($slot$)
 3:     if $v$ = FAIL then deal with failure
 4:     return $v$
 5: procedure WRITE($slot, v_{new}$)
 6:     $v_{old}$ = RDMA_READ_primary($slot$)
 7:     $v\_list$ = RDMA_CAS_backups($slot, v_{old}, v_{new}$)
 8:     // Change all the $v_{old}$s in the $v\_list$ to $v_{new}$s.
 9:     $v\_list$ = change_list_value($v\_list, v_{old}, v_{new}$)
10:     $win$ = EVALUATE_RULES($v\_list$)  ▷ The last writer returns
        the winning rule while other writers return LOSE.
11:     if $win$ = Rule_1 then
12:         RDMA_CAS_primary($slot, v_{old}, v_{new}$)
13:     else if $win \in$ {Rule_2, Rule_3} then
14:         RDMA_CAS_backups($slot, v\_list, v_{new}$)
15:         RDMA_CAS_primary($slot, v_{old}, v_{new}$)
16:     else if $win$ = LOSE then
17:         repeat
18:             sleep a little bit
19:             $v_{check}$ = RDMA_READ_primary($slot$)
20:             if notified failure then goto Line 24
21:         until $v_{check} \neq v_{old}$
22:         if $v_{check}$ = FAIL then goto Line 24
23:     else if $win$ = FAIL then
24:         deal with failure
25:     return
```

discussion of failure handling in Section 5. We call the slots in the primary and backup hash indexes primary slots and backup slots, respectively.

For READ operations, clients directly read the values in the primary slots using RDMA_READ. For WRITE operations, SNAPSHOT first resolves write conflicts by letting conflicting writers collaboratively decide on a last writer with three conflict resolution rules and then let the decided last writer modify the primary slot. Figure 6 shows the process that two clients simultaneously WRITE the same slot. The corresponding algorithms are shown in Algorithms 1 and 2. Clients first read the value in the primary slot as v_{old} (①). Then each client modifies all backup slots by broadcasting RDMA_CAS operations (②) with v_{old} as the expected value and v_{new} as the swap value. On receiving an RDMA_CAS, the RNICs on MNs atomically modify the value in the target slot only if v_{old} matches the current value in the slot. Since all writers initiate RDMA_CAS operations with the same v_{old} and different v_{new}s and all backup slots initially hold v_{old}, the atomicity of RDMA_CAS ensures that each backup slot can only be modified once by a single writer. As a result, the values in all backup slots will be fixed after each of them has received one RDMA_CAS from one writer [1]. Meanwhile, since an RDMA_CAS returns the value in the slot before it is modified, all clients

Algorithm 2 The rule evaluation procedure of SNAPSHOT

```
 1: procedure EVALUATE_RULES($v\_list, slot, v_{new}, v_{old}$)
 2:     $v_{maj}$ = The majority value in $v\_list$
 3:     $cnt_{maj}$ = The number of $v_{maj}$ in $v\_list$
 4:     if FAIL $\in v\_list$ then
 5:         return FAIL
 6:     else if $cnt_{maj}$ = Len($v\_list$) then
 7:         return Rule 1 if $v_{maj} = v_{new}$ else LOSE
 8:     else if $2 * cnt_{maj} >$ Len($v\_list$) then
 9:         return Rule 2 if $v_{maj} = v_{new}$ else LOSE
10:     else if $v_{new} \notin v\_list$ then
11:         return LOSE
12:     $v_{check}$ = RDMA_READ($slot$)
13:     if $v_{check}$ = FAIL then
14:         return FAIL
15:     else if $v_{check} \neq v_{old}$ then
16:         return FINISH
17:     else if $min(v\_list) = v_{new}$ then
18:         return Rule 3
19:     return LOSE
```

can perceive the new values in the backup slots (③) through the return values of the broadcast of RDMA_CAS operations. The return values are denoted as v_list in Algorithm 1.

With v_list, SNAPSHOT defines the following three rules to let conflicting clients collaboratively decide on a last writer:

> **Rule 1**: A client that has successfully modified all the backup slots is the last writer.
>
> **Rule 2**: A client that has successfully modified a majority of backup slots is the last writer.
>
> **Rule 3**: If no last writer can be decided with the former two rules, the client that has written the minimal target value (v_{new}) is considered as the last writer.

The three rules are evaluated sequentially as shown in Algorithm 2. **Rule 1** provides a fast path when there are no conflicting modifications. **Rule 2** preserves the most successful CAS operations to minimize the overhead of executing atomic operations on RNICs when conflicts are rare [29]. Finally, **Rule 3** ensures that the protocol can always decide on the last writer. To ensure the uniqueness of the last write, a client issues another RDMA_READ to check if the primary slot has been modified (Line 12, Algorithm 2) before evaluating **Rule 3**. If the primary slot has not been modified, then the RDMA_CAS_backups (Line 7, Algorithm 1) of the client must happen before the last writer modifies the primary slot. Hence, it is safe to evaluate **Rule 3** because the v_list must contain the value of the last writer if it has already been decided. Otherwise, **Rule 3** will not be evaluated because the modification of the primary slot means the decision of a last writer. Relying on the three rules, a unique last writer can be decided without any further network communications. For example, in Figure 6, Client 1 is the last writer according to **Rule 2**. Client 1 then modifies the backup slots that do not yet contain its proposed value using RDMA_CASes and then modifies the primary slot. Other conflicting clients iteratively READ the value in the

[1] That the process that all conflicting clients broadcast RDMA_CASes to modify backup slots is just like taking a snapshot, which is why the replication protocol is named SNAPSHOT.

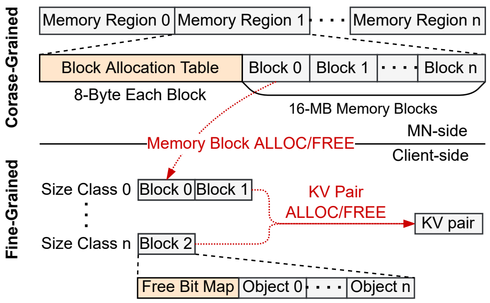


Figure 7: The two-level memory management scheme.

primary slot and return success after finding the change in the primary slot. The primary slot may remain unmodified only under the situation when the last writer crashed, which will be discussed in Section 5.

Correctness. The SNAPSHOT replication protocol guarantees linearizability of the replicated hash indexes with last-writer-wins conflict resolution like shared register protocols [7, 43]. We briefly demonstrate the correctness of SNAPSHOT using the notion of the linearizable point of KV operations. A formal proof is shown in the extended version of this paper [56]. A linearizable point is a point when an operation atomically takes effect in its invocation and completion [26]. For READ, the linearizable point happens when it gets the value in the primary slot. For WRITE operations, the linearizable point of the last writer happens when it modifies the primary slot. Linearizable points of other conflicting writers appear instantly before the last writer modifies the primary slot. Conflicts between readers and the last writer are resolved by RNICs because the last writer atomically modifies the primary slot using RDMA_CAS operations and readers access the primary slot using RDMA_READ operations.

Performance. SNAPSHOT guarantees a bounded worst-case latency when clients WRITE the hash index. Under the situation when **Rule 1** is triggered, 3 RTTs are required to finish a WRITE operation. Under situations when **Rule 2** or **Rule 3** is triggered, 4 or 5 RTTs are required, respectively.

4.4 Two-Level Memory Management

Memory management is responsible for allocating, replicating, and freeing memory spaces for KV pairs on MNs. As discussed in Section 3.2, the key challenge of DM management is that conducting the management tasks solely on clients or on MNs cannot satisfy the performance requirement of frequent memory allocation for KV requests. FUSEE addresses this issue via a two-level memory management scheme. The key idea is to split the server-centric memory management tasks into compute-light coarse-grained management and compute-heavy fine-grained management run on MNs and clients.

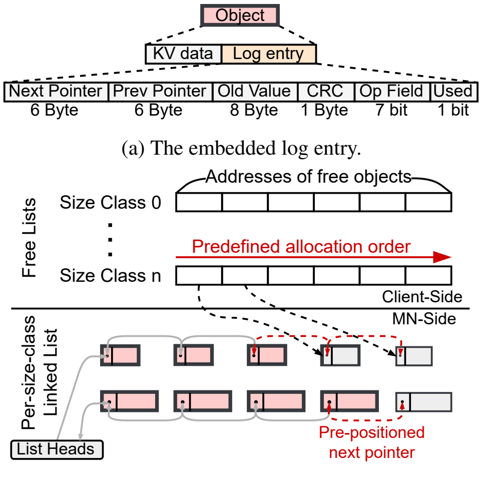
FUSEE first replicates and partitions the 48-bit memory space on multiple MNs. Similar to FaRM [18], FUSEE shards the memory space into 2GB memory regions and maps each region to r MNs with consistent hashing [32], where r is the replication factor. Specifically, consistent hashing maps a region to a position in a hash ring. The replicas are then stored at the r MNs successively following the position and the primary region is placed on the first of the r MN.

Figure 7 shows the two-level memory allocation of FUSEE. Allocating a memory space for a KV pair happens before writing the KV pair, as introduced in Section 4.1. The first level is the coarse-grained MN-side memory block allocation with low computation requirements. Each MN partitions its local memory regions into coarse-grained memory blocks, *e.g.*, 16 MB, and maintains a block allocation table ahead of each region. For each memory block, the block allocation table records a client ID (CID) that allocates it. Clients allocate memory blocks by sending ALLOC requests to MNs. On receiving an ALLOC request, an MN allocates a memory block from one of its primary memory regions, records the client ID in the block allocation tables of both primary and backup regions, and replies with the address of the memory block to the client. The coarse-grained memory allocation information is thus replicated on r MNs and can survive MN failures. The second level is the fine-grained client-side object allocation that allocates small objects to hold KV pairs. Specifically, clients manage the blocks allocated from MNs exclusively with slab allocators [6]. The client-side slab allocators split memory blocks into objects of distinct size classes. A KV pair is then allocated from the smallest size class that fits it.

The allocated objects can be freed by any client. To efficiently reclaim freed memory objects on client sides, FUSEE stores a free bit map ahead of each memory block, as shown in Figure 7, where each bit indicates the allocation state of one object in the memory block. The free bit map is initialized to be all zeros when a block is allocated. To free an object, a client sets the corresponding bit to '1' in the free bit map with an RDMA_FAA operation. By reading the free bit map, clients can easily know the freed objects in their memory blocks and reclaim them locally. FUSEE frees and reclaims memory objects periodically using background threads in a batched manner to avoid the additional RDMA operations on the critical paths of KV accesses. The correctness of concurrently accessing KV pairs and reclaiming memory spaces is guaranteed by RACE hashing [73], where clients check the key and the CRC of the KV pair on data accesses.

4.5 Embedded Operation Log

Operation logs are generally adopted to repair the partially modified hash index incurred by crashed clients. Conventional operation logs record a log entry for each KV request that modifies the hash index. The log entries are generally written in an append-only manner so that the order of log entries reflects the execution order of KV requests. The recovery process can thus find the crashed request and fix the corrupted metadata by scanning the ordered log entries. However, constructing operation logs incurs high log maintenance overhead on DM because writing log entries adds remote memory accesses on the critical paths of KV requests.



(a) The embedded log entry.

(b) The organization of the embedded operation log.

Figure 8: The embedded operation log.

To reduce the log maintenance overhead on DM, FUSEE adopts an *embedded operation log* scheme that embeds log entries into KV pairs. The embedded log entry is written together with its corresponding KV pair with one `RDMA_WRITE` operation. The additional RTTs required for persisting log entries are thus eliminated. However, by embedding log entries in KV pairs, the execution order of KV requests cannot be maintained because the log entries are no longer continuous. To address this problem, the embedded operation log scheme maintains per-size-class linked lists to organize the log entries of a client in the execution order of KV requests. As shown in Figure 8b, a per-size-class linked list is a doubly linked list that links all allocated objects of the size class in the order of their allocations. The object allocation order reflects the execution order of KV requests because all KV requests that modify the hash index, *e.g.*, `INSERT` and `UPDATE`, need to allocate objects for new KV pairs. For `DELETE`, FUSEE allocates a temporary object recording the log entry and the target key and reclaims the object on finishing the `DELETE` request. FUSEE stores the list heads on MNs during the initialization of clients, which will be accessed during the recovery process of clients (Section 5).

An embedded log entry is a 22-byte data structure stored behind KV pairs, as shown in Figure 8a. It contains a 6-byte *next pointer*, a 6-byte *prev pointer*, an 8-byte *old value*, a 1-byte *CRC*, a 7-bit *opcode*, and a *used* bit. The *next pointer* points to the next object of the size class that will be allocated and the *prev pointer* points to the object allocated before the current one. The *old value* records the old value of the primary slot for recovery proposes, which will be discussed in Section 5. The 1-byte *CRC* is adopted to check the integrity of the *old value* under client failures. The *operation field* records the operation type, *i.e.*, `INSERT`, `UPDATE`, or `DELETE`, so that the crashed operation can be properly retried during recovery. The *used bit* indicates if an object is in-use or free. Storing the *used bit* at the end of the entire object can be used to check the integrity of an entire object. This is because the order-preserving nature of `RDMA_WRITE` operations ensures that the used bit is written only after all other contents in the object have been successfully written.

FUSEE efficiently organizes per-size-class linked lists by co-designing the linked list maintenance process with the memory allocation process. As shown in Figure 8b, for each size class, a client organizes the addresses of remote free objects locally as a free list. Since an object is always allocated from the head of a local free list, the allocation order of each size class is pre-determined. Based on the pre-determined order, for each allocation, a client pre-positions the *next pointer* to point to the free object in the head of the local free list and the *prev pointer* to point to the last allocated object of the size class. Both the *next pointer* and the *prev pointer* are thus known before each allocation and the entire log entry can be written to MNs with the KV pair in a single `RDMA_WRITE`.

Combined with the SNAPSHOT replication protocol, the execution process is shown as follows. First, for each writer, a log entry with an empty *old value* and *CRC* is written with the KV pair in a single `RDMA_WRITE`. Then, for the last writer of the SNAPSHOT replication protocol, the *old value* is modified to store the old value of the primary slot before the primary slot is modified. For other non-last writers, the used bits in their corresponding KV log entries are reset to '0' after finding the modification of the primary slot.

4.6 Optimizations

Adaptive index cache. Index caching is widely adopted on RDMA-based KV stores to reduce request RTTs [60, 66–68]. For a key, the index cache caches the remote addresses of the replicated index slots and the addresses of the KV pairs locally. With the cached KV pair addresses, `UPDATE`, `DELETE`, and `SEARCH` requests can read KV pairs in parallel with searching the hash index, reducing an RTT on cache hits. To guarantee cache coherence, an invalidation bit is stored together with each KV pair, which is used by clients to check whether the KV pair is valid or invalid. However, by accessing the index cache, invalid KV pairs (*e.g.*, outdated) can be fetched into clients, causing read amplification.

To attack the read amplification issue, FUSEE adaptive bypasses the index cache by distinguishing read-intensive and write-intensive keys. For each cached key, FUSEE maintains an access counter and an invalid counter which increases by 1 each time the key is accessed or found to be invalid. A client calculates an *invalid ratio* $I = \frac{\text{invalid counter}}{\text{access counter}}$ for each cached key. The index cache is bypassed when accessing a key with $I > threshold$ because the key is write-intensive and the cached key address points to an invalid KV pair with high probability. The *invalid ratio* can adapt to workload changes, *i.e.*, a write-intensive key becomes read-intensive, because the access counter of the key keeps increasing while the invalid counter stops. Besides, the adaptive scheme does not affect the `SEARCH` latency for most cases since only write-intensive keys bypass the cache.

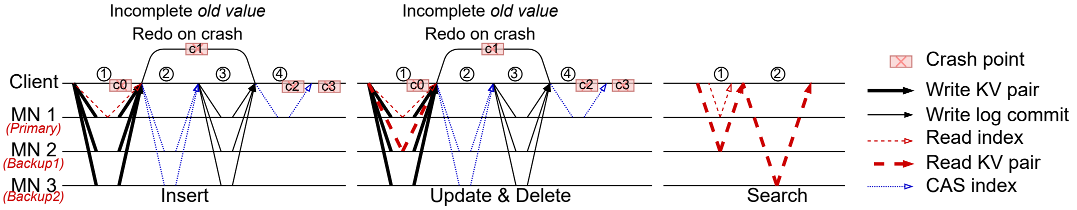


Figure 9: The workflows of different KV requests. `INSERT`: *① write the KV pair to all replicas and read the primary index slot. ② CAS all backup slots. ③ write the old value to the log header. ④ CAS the primary slot.* `UPDATE` *&* `DELETE`: *① write the KV pair, read the primary slot, and read the KV pair according to the index cache. ② CAS backup slots. ③ write the old value to the log header. ④ CAS the primary slot.* `SEARCH`: *① read the primary slot and the KV pair according to the index cache. ② read the KV pair on cache misses.*

RDMA-related optimizations. KV requests require multiple remote memory accesses. FUSEE adopts doorbell batching and selective signaling [29] to reduce RDMA overhead. Figure 9 shows the procedures for executing different KV requests. Each request consists of multiple phases with multiple network operations. For each phase, FUSEE adopts doorbell batching [29] to reduce the overhead of transmitting network operations from user space to RNICs and selective signaling to reduce the overhead of polling RDMA completion queues. Consequently, each phase only incurs 1 network RTT. For `INSERT`, `DELETE`, and `UPDATE` requests, four RTTs are required in general cases. For `SEARCH` requests, at most two RTTs are required and only one RTT is required in the best case due to the index cache.

5 Failure Handling

Similar to existing replication protocols [33, 59, 62], FUSEE relies on a fault-tolerant master with a lease-based membership service [24] to handle failures. The master maintains a membership lease for both clients and MNs so that clients always know alive MNs by periodically extending their leases. The failures of both clients and MNs can be detected by the master when they no longer extend their leases. Master crashes are handled by replicating the master with state machine replication [24, 59, 62]. We formally verify FUSEE in TLA+ [35] for safety and absence of deadlocks under MN failures and more details are shown in the extended version [56].

5.1 Failure Model

We consider a partially synchronous system where processes, *i.e.*, clients and MNs, are equipped with loosely synchronized clocks [20, 24, 33]. FUSEE assumes *crash-stop* failures, where processes, *i.e.*, clients and MNs, may fail due to crashing and their operations are non-Byzantine.

Under this failure model, FUSEE guarantees linearizable operations, *i.e.*, each KV operation is atomically committed in a time between its invocation and completion [26]. All the objects of FUSEE are durable and available under an arbitrary number of client crashes and at most $r-1$ MN crashes, where r is the replication factor.

5.2 Memory Node Crashes

MN crashes lead to failed accesses to KV pairs and hash slots. For accesses to KV pairs, clients can access the backup replicas according to the consistent hashing scheme.

The complication comes from the unavailable primary and backup slots that affect the normal execution of index `READ` and `WRITE` operations. FUSEE relies on the fault-tolerant master to execute operations on clients' behalves under MN failures. We first introduce how clients `READ`/`WRITE` the replicated slots and then introduce the master's operations.

When executing index `WRITE` under MN crashes, FUSEE allows the last writer decided by the SNAPSHOT replication protocol to continue modifying all *alive* slots to the same value. Other writers send RPC requests to the master and wait for the master to reply with a correct value in the replicated slots. Under situations when no last writer can be decided, the master decides the last writer and modifies all the index slots on behalf of clients. For `READ` operations, executions are not affected under the following two cases. First, if the primary slot is still alive, clients can read the primary slot normally. Second, if the primary slot crashes, clients read all *alive* backup slots. If all *alive* backup slots contain the same value, reading this value is safe because there are no write conflicts. Otherwise, clients use RPCs and rely on the master to return a correct value for the crashed slot. Since `READ` operations are only affected under write conflicts, most `READ` can continue under the read-intensive workloads that dominate in real-world situations [9, 71].

On detecting MN crashes, the master first blocks clients from further modifying the crashed slots with the lease expiration. The master then acts as a representative last writer that modifies all *alive* slots to the same value. Specifically, the master selects a value v in an *alive* backup slot and modifies all *alive* slots to v. Since the SNAPSHOT protocol modifies the backup slots before the primary slot, the values in the backup slots are always newer than the primary slot. Hence, the master choosing a value from a backup slot is correct because it proceeds the conflicting write operations. In cases where all backup slots crash, the master selects the value in the primary slot. Clients that receive old values from the master retry their write operations to guarantee that their new value is written. The master then writes the *old value* in the operation log header to prevent clients from redoing operations when recovering from crashed clients (Section 5.3). Finally, the master reconfigures new primary and backup slots and returns the selected value to all clients that wait for a reply. After the reconfiguration of the primary and backup slots, all KV requests can be executed normally without involving the master. During the whole process, only accesses to the crashed slots are affected and the blocking time can be short thanks to the microsecond-scale membership service [24].

5.3 Client Crashes

Crashed clients may result in two issues. First, their allocated memory blocks remain unmanaged, causing memory leakage. Second, other clients may be unable to modify a replicated index slot if the crashed client is the last writer. The master uses embedded operation logs to address these two issues.

The recovery process is executed in the compute pool and consists of two steps, *i.e.*, memory re-management and index repair. Memory re-management restores the coarse-grained memory blocks allocated by the client and the fine-grained object usage information of the client. The recovery process first gets all memory blocks managed by the crashed client by letting MNs search for their local block allocation tables. Then the recovery process traverses the per-size-class linked lists to find all used objects and log entries. With the used objects and the allocated memory blocks, the recovery process can easily restore the free object lists of the crashed client. Hence, all the memory spaces of the crashed client are re-managed.

The index repair procedure then fixes the partially modified hash index. FUSEE deems all requests at the end of per-size-class linked lists as potentially crashed requests. For incomplete log entries, *i.e.*, the *used bit* at the end of the log entry is not set, the client must crashes during writing the KV pair (c0 in Figure 9). The object is directly reclaimed without further operation since the writing of the object has not been completed. For a log entry with an incomplete *old value* according to the *CRC* field, FUSEE redoes the request according to the *operation field* and the KV pair. Under this situation, either the request belongs to the last writer that crashed before committing the log (c1 in Figure 9), or it belongs to other non-last writers. In the first case, the values in the backup slots may not be consistent and the primary slot has not been modified to a new value. Redoing the request can make the backup and primary slots consistent. In the second case, since the request of crashed non-last writers has not returned to clients, redoing the request does not violate the linearizability. For a request with a complete *old value*, the request must belong to a last writer. However, the request may finish (c3) or crash before the primary slot is modified (c2). The recovery process checks the value in the primary slot (v_p) and the value in the *old value* (v_{old}) to distinguish c2 from c3. If $v_p = v_{old}$, the request crashed before the primary was modified because v_{old} records the value before index modification. Since all backup slots are consistent, the recovery process modifies the primary slot to the new value and finishes the recovery. Otherwise, the request is finished and no further operation is required. After recovering the request, the master asynchronously checks content in the v_{old}s in log entries of the crashed client to recover its batched free operations.

5.4 Mixed Crashes

In situations where clients and MNs crash together, FUSEE recovers the failures separately. FUSEE first lets the master recover all MN crashes and then starts the recovery processes for failed clients. KV requests can proceed because the master acts as the last writer for all blocked KV requests. No request is committed twice because the master commits the operation logs on clients' behalves.

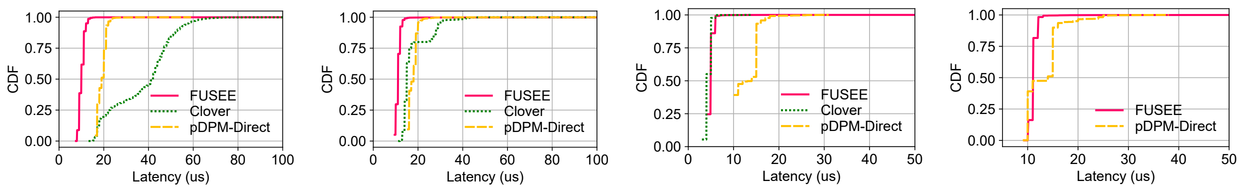
6 Evaluation

6.1 Experiment Setup

Implementation. We implement FUSEE from scratch in C++ with 13k LOC. We implement RACE hashing carefully according to the paper due to no available open-source implementations. Coroutines are employed on clients to hide the RDMA polling overhead, as suggested in [30, 73]. The design of FUSEE is agnostic to the lower-level memory media of memory nodes, *i.e.*, any memory node with either persistent memory (PM) or DRAM that provides `READ`, `WRITE`, and 8-byte `CAS` interfaces is compatible. We adopt monolithic servers with RNICs and DRAM to serve as MNs like Clover [60] since we do not have access to smartNICs and PM. Specifically, we start an MN process on a monolithic server to register RDMA memory regions and serve memory allocation RPCs with a UDP socket. MN processes serve memory allocation requests with UDP sockets. Since the socket *receive* is a blocking system call, the process will be in the blocked state with no CPU usage most of the time.

Testbed. We run all experiments on 22 physical machines (5 MNs and 17 CNs) on the APT cluster of CloudLab [19]. Each machine is equipped with an 8-core Intel Xeon E5-2450 processor, 16GB DRAM, and a 56Gbps Mellanox ConnectX-3 IB RNIC. These machines are interconnected with 56Gbps Mellanox SX6036G switches.

Comparison. We compare FUSEE with two state-of-the-art KV stores on DM, *i.e.*, pDPM-Direct and Clover [60]. pDPM-Direct stores and manages the KV index and memory space on the clients. It uses a distributed consensus protocol to ensure metadata consistency and locks to resolve data access conflicts. We extend the open-source version of pDPM-Direct to support string keys for fair comparison in our evaluation. Clover is a semi-disaggregated KV store that adopts monolithic servers to manage memory spaces and a hash index. All `UPDATE` and `INSERT` requests have to go through the metadata server, requiring additional compute power. For both pDPM-Direct and Clover, client-side caches are enabled following their default settings. To show the effectiveness of SNAPSHOT and the adaptive index cache, we implement FUSEE-CR and FUSEE-NC, two alternative versions of FUSEE. FUSEE-CR replicates index modifications by sequentially `CAS`ing all replicas to enforce sequential accesses. FUSEE-NC is the version of FUSEE without a client-side



(a) `INSERT` latency CDF. (b) `UPDATE` latency CDF. (c) `SEARCH` latency CDF. (d) `DELETE` latency CDF.

Figure 10: The CDFs of different KV request latency under the microbenchmark.

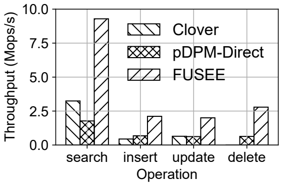


Figure 11: The throughputs of microbenchmark.

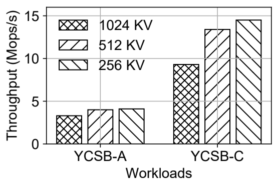


Figure 12: The throughput of FUSEE under different KV sizes.

cache. For all these methods, we evaluate their throughput and latency with both micro and YCSB [15] benchmarks.

Since the open-source version of Clover and pDPM-Direct only support one index replica, we compare FUSEE with these two approaches with a single index replica and two data replicas in the microbenchmark (Section 6.2) and YCSB performance (Section 6.3) evaluations. When evaluating FUSEE with a single index replica, the embedded log is constructed, but the commit of the log is skipped since committing the log is used to ensure the consistency of multiple index replicas. The performance of FUSEE with multiple replicas is evaluated in the fault-tolerance evaluation (Section 6.4).

6.2 Microbenchmark Performance

We use microbenchmarks to evaluate the operation throughput and latency of the three approaches. For FUSEE and pDPM-Direct, we use 16 CNs and 2 MNs. For Clover, we use 17 CNs and 2 MNs because it needs an additional metadata server, consuming 8 more CPU cores and an additional RNIC. We do not use multiple metadata servers for Clover because the current open-source implementation of Clover only supports a single metadata server. We run 128 client processes on the 16 CNs, where each CN holds 8 clients. The `DELETE` of Clover is not tested because Clover does not support it.

Latency. To evaluate the latency of KV requests, we use a single client to iteratively execute each operation $10,000$ times. Figure 10 shows the cumulative distribution functions (CDFs) of the request latency. FUSEE performs the best on `INSERT` and `UPDATE`, since the SNAPSHOT replication protocol has bounded RTTs. FUSEE has a little higher `SEARCH` latency than Clover since FUSEE reads the hash index and the KV pair in a single RTT, which is slower than only reading the KV pair in Clover. FUSEE has slightly higher `DELETE` latency than pDPM-Direct because FUSEE writes a log entry and reads the hash index in a single RTT, which is slower than just reading the hash index in pDPM-Direct.

Throughput. Figure 11 shows the throughput of the three approaches. The throughput of pDPM-Direct is limited by its remote lock, which causes extensive lock contention as the number of clients grows. For Clover, even though it consumes more hardware resources, *i.e.*, 8 additional CPU cores and an RNIC, the scalability is still lower than FUSEE. This is because the CPU processing power of the metadata server bottlenecks its throughput. On the contrary, FUSEE improves the overall throughput by eliminating the computation bottleneck of the metadata server and efficiently resolving conflicts with the SNAPSHOT replication protocol.

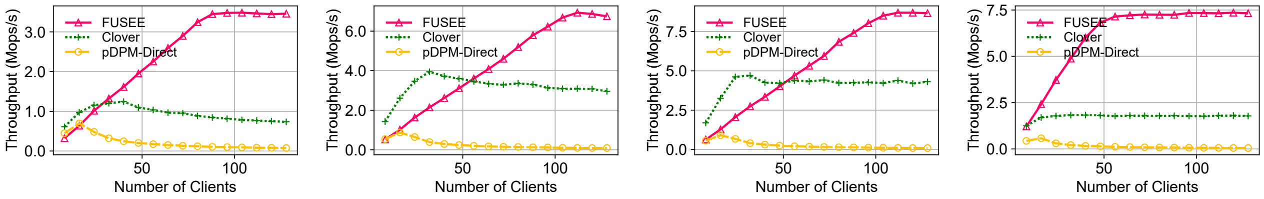
6.3 YCSB Performance

For YCSB benchmarks [15], we generate $100,000$ keys with the Zipfian distribution ($\theta = 0.99$). We use 1024-byte KV pairs, which is representative of real-world workloads [9, 15, 17]. The hardware setup is the same as microbenchmarks.

YCSB Throughput. Figure 13 shows the throughput of three approaches with different numbers of clients. Clover performs the best under a small number of clients since adopting the metadata server simplifies KV operations. Compared with Clover, pDPM-Direct and FUSEE require more RDMA operations to resolve index modification conflicts. As the number of clients grows, the throughput of Clover and pDPM-Direct does not increase because the throughput is bottlenecked by the metadata server and the lock contention, respectively. Compared with Clover, FUSEE scales better with the growing number of clients while consuming fewer resources. Compared with pDPM-Direct, FUSEE improves the throughput by avoiding lock contention. When the number of clients reaches 128, the throughput of FUSEE is 4.9× and 117× higher than Clover and pDPM-Direct, respectively.

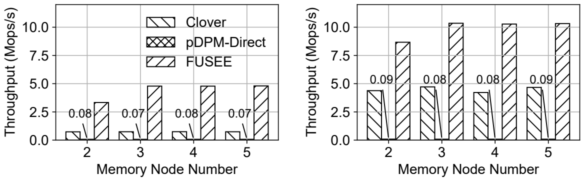
Figure 14 shows the throughput of the three approaches with a write-intensive workload (YCSB-A) and a read-intensive workload (YCSB-C) when varying numbers of MNs from 2 to 5 using 128 clients. The throughput of pDPM-Direct and Clover does not increase due to being limited by lock contention and the limited compute power of the metadata server, respectively. As for FUSEE, the throughput improves as the number of memory nodes increases from 2 to 3. There is no further throughput improvement because the total throughput is limited by the number of compute nodes.

Figure 12 shows the throughput of FUSEE under smaller KV sizes. Since the throughput of FUSEE is limited by the bandwidth of MN-side RNICs, the YCSB-C throughput of FUSEE increases by 44.1% and 55.9% with 512B and 256B



(a) A (SEARCH:UPDATE = 0.5:0.5). (b) B (SEARCH:UPDATE = 0.95:0.05). (c) C (100% SEARCH). (d) D (SEARCH:INSERT = 0.95:0.05).

Figure 13: The scalability of FUSEE under different YCSB workloads.



(a) YCSB-A throughput. (b) YCSB-C throughput.

Figure 14: The throughput with different numbers of MNs.

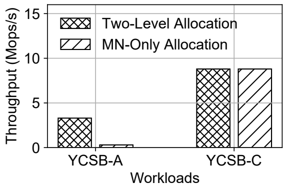


Figure 17: The throughput of different memory allocation methods.

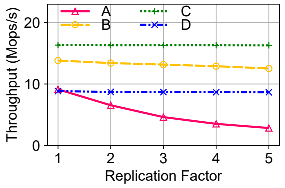


Figure 18: YCSB throughput under different replication factors.

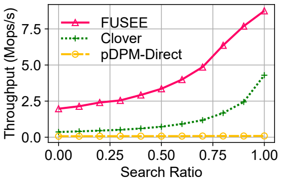


Figure 15: Throughput under different SEARCH-UPDATE ratios.

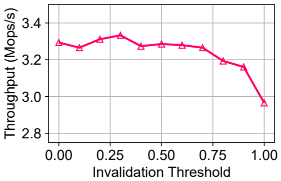


Figure 16: Throughput under different adaptive cache thresholds.

KV pairs, respectively. The performance of FUSEE is not affected by the dataset size because the performance depends only on the number of RTTs of KV requests, which is deterministic as presented in Section 4.

Read-write performance. Figure 15 shows the throughput of the three approaches under different SEARCH-UPDATE ratios. As the portion of UPDATE grows, the throughput of all three methods decreases because UPDATE requests involve more RTTs. However, FUSEE exhibits the best throughput due to eliminating the computation bottleneck of metadata servers.

Adaptive index cache performance. Figure 16 shows the YCSB-A throughput of FUSEE with different adaptive index cache thresholds. The throughput of FUSEE decreases with the increasing thresholds because more bandwidth is wasted on fetching invalidated KV pairs with a high threshold.

Two-level memory allocation performance. To show the effectiveness of the two-level memory allocation scheme, we compare FUSEE with an MN-centric memory allocation scheme, as shown in Figure 17. The YCSB-A throughput drops 90.9% due to the limited compute power on MNs, while the YCSB-C throughput remains the same since no memory allocation is involved in the read-only setting.

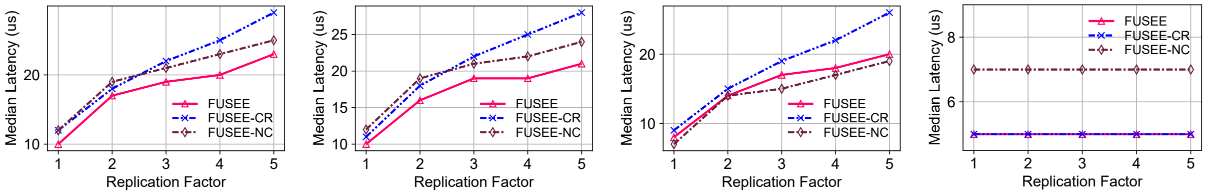
6.4 Fault Tolerance & Elasticity

SNAPSHOT Replication Protocol. Figure 19 shows the median latency of FUSEE, FUSEE-NC, and FUSEE-CR with different replication factors under microbenchmarks. We set both the numbers of index replicas and data replicas to r where r is the replication factor. The latency of FUSEE-CR on INSERT, UPDATE, and DELETE grows linearly as the replication factor because it modifies index replicas sequentially, and the number of RTTs equals the replication factor. Differently, the latency of FUSEE grows slightly with the replication factor because SNAPSHOT has a bounded number of RTTs. For SEARCH requests, FUSEE and FUSEE-CR have comparable latency since they execute SEARCH similarly. Compared with FUSEE-NC, FUSEE has lower latency for UPDATE, DELETE, and SEARCH due to fewer RTTs. The INSERT latency is slightly higher than that of FUSEE-NC because FUSEE spends additional time to maintain the local cache. Figure 18 shows the throughput of FUSEE under different replication factors. For YCSB-A and YCSB-B, the throughput drops as the replication factor grows. The YCSB-D throughput slightly drops from 8.8 Mops to 8.6 Mops due to the read-intensive nature of YCSB-D. The YCSB-C throughput remains the same due to no index modifications.

Search under Crashed MNs. FUSEE allows SEARCH requests to continue when MNs crash under read-intensive workloads. Figure 20 shows the throughput of 9 seconds of execution, where memory node 1 crashes at the 5th second. The overall throughput drops to half of the peak throughput because all data accesses come to one MN. The throughput is then limited by the network bandwidth of a single RNIC.

Recover from Crashed Clients. To evaluate the efficiency of a client recovering from failures, we crash and recover a client after UPDATE 1,000 times. As shown in Table 1, FUSEE takes 177 milliseconds to recover from a client failure. The memory registration and connection re-establishment account for 92% of the total recovery time. The log traversal and KV request recovery only account for 4% of the recovery time, which implies the affordable overhead of log traversal.

Elasticity. FUSEE supports dynamically adding and shrinking clients. We show the elasticity of FUSEE by dynamically adding and removing 16 clients when running the YCSB-C workload. As shown in Figure 21, the throughput increases when the number of clients increases from 16 to 32 and re-



(a) UPDATE median latency. (b) DELETE median latency. (c) INSERT median latency. (d) SEARCH median latency.

Figure 19: Median operation latency of FUSEE, FUSEE-NC and FUSEE-CR under different replication factors.

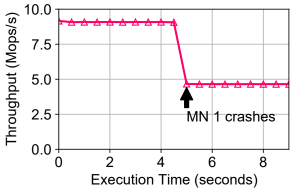


Figure 20: YCSB-C throughput under a crashed memory node.

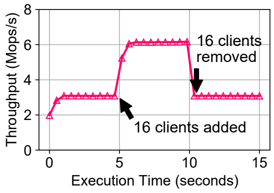


Figure 21: The elasticity of FUSEE.

Table 1: Client recovery time breakdown.

Step	Time (ms)	Percentage
Recover connection & MR	163.1	92.1%
Get Metadata	0.3	0.2%
Traverse Log	3.5	2.0%
Recover KV Requests	3.5	2.0%
Construct Free List	6.6	3.7%
Total	177.0	100%

sumes to the previous level after removing 16 clients.

7 Related Work

Disaggregated Memory. Existing approaches can be classified into software-based, hardware-based, and co-design-based memory disaggregation. Software-based approaches hide the DM abstraction by modifying operating systems [3, 23, 47, 55, 61], virtual machine monitors [41], or runtimes [54, 63]. Hardware-based ones design memory buses [14, 40] to enable efficient remote memory access. Co-design-based approaches co-design software and hardware [8, 25, 38, 65] to gain better application throughput and latency on DM. The design of FUSEE is agnostic to the low-level implementations of all these DM approaches.

Disaggregated Memory Management. MIND [38] and Clio [25] are the two state-of-the-art memory management approaches on DM. But they both rely on special hardware to manage memory spaces. The two-level memory management of FUSEE resembles the hierarchical memory management of The Machine [21, 34]. The difference is that FUSEE focuses on fine-grained KV allocation with commodity RNICs, while The Machine relies on special SoCs and directly manages physical memory devices.

Memory-disaggregated KV stores. Clover [60] and Dinomo [37] are the most related memory-disaggregated KV stores. Compared with Clover [60], FUSEE brings disaggregation to metadata management and gains better resource efficiency and scalability. Finally, Dinomo [37] is a fully-disaggregated KV store that was developed concurrently with our system. Dinomo proposes ownership partitioning to reduce coordination overheads of managing disaggregated metadata. However, it assumes that the disaggregated memory pool is fault-tolerant, and hence its design does not consider MN failures. In contrast, FUSEE addresses the challenges of handling MN failures with the SNAPSHOT replication protocol. There are many related RDMA-based KV stores [18, 28, 30, 31, 45, 48, 51, 57, 60, 66–68]. They are infeasible on DM since they rely on server-side CPUs to execute KV requests. Besides, there are emerging approaches that use SmartNICs to construct KV stores [39, 52]. FUSEE can also benefit from the additional compute power by offloading the memory management to SmartNICs.

Replication. Both traditional [2, 22, 36, 43, 46, 50, 59, 62] and RDMA-based [33, 58, 72] replication protocols are designed to ensure data durability. However, all these approaches are server-centric replication protocols designed for monolithic servers. Differently, SNAPSHOT is a client-centric replication protocol designed for the DM architecture and achieves high scalability with collaborative conflict resolution.

8 Conclusion

This paper proposes FUSEE, a fully memory-disaggregated KV store, that achieves both resource efficiency and high performance by disaggregating metadata management. FUSEE adopts a client-centric replication protocol, a two-level memory management scheme, and an embedded log scheme to attack the challenges of weak MN-side compute power and complex failure situations on DM. Experimental results show that FUSEE outperforms the state-of-the-art approaches by up to 4.5× with less resource consumption.

Acknowledgments

We sincerely thank our shepherd Kimberly Keeton and the anonymous reviewers for their constructive comments and suggestions. This work was supported by the National Natural Science Foundation of China (Nos. 62202511 & 61971145), the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14210920 of the General Research Fund), and Huawei Cloud. Pengfei Zuo is the corresponding author (pfzuo.cs@gmail.com).

References

[1] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association, 2015.

[2] Peter Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering, San Francisco, California, USA, October 13-15, 1976*, pages 562–570. IEEE Computer Society, 1976.

[3] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 14:1–14:16. ACM, 2020.

[4] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 49–60. IEEE Computer Society, 2017.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, August 22-24, 1990*, pages 363–375. ACM, 1990.

[6] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer 1994 Technical Conference, Boston, Massachusetts, USA, June 6-10, 1994, Conference Proceeding*, pages 87–98. USENIX Association, 1994.

[7] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 591–617. USENIX Association, 2020.

[8] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 79–92. ACM, 2021.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 209–223. USENIX Association, 2020.

[10] cgroups. cgroups. https://man7.org/linux/man-pages/man7/cgroups.7.html, 2022.

[11] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 19:1–19:14. ACM, 2019.

[12] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. Giza: Erasure coding objects across global data centers. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 539–551. USENIX Association, 2017.

[13] Zhiguang Chen, Yu-Bo Liu, Yong-Feng Wang, and Yutong Lu. A gpu-accelerated in-memory metadata management scheme for large-scale parallel file systems. *J. Comput. Sci. Technol.*, 36(1):44–55, 2021.

[14] Gen-Z Consortium. Gen-z technology. https://genzconsortium.org/.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[16] Intel Corporation. Driving exascale computing and hpc with intel. https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-driving-exascale-computing.html.

[17] 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 *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 33–49. USENIX Association, 2021.

[18] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70. ACM, 2015.

[19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of cloudlab. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1–14. USENIX Association, 2019.

[20] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[21] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan S. Milojicic. Beyond processor-centric operating systems. In George Candea, editor, *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association, 2015.

[22] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating System Principles, SOSP 1979, Asilomar Conference Grounds, Pacific Grove, California, USA, 10-12, December 1979*, pages 150–162. ACM, 1979.

[23] 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 2017, Boston, MA, USA, March 27-29, 2017*, pages 649–667. USENIX Association, 2017.

[24] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, Carlsbad, CA, 2022. USENIX Association.

[25] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. *CoRR*, abs/2108.03492, 2021.

[26] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[27] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert van Renesse, Sydney Zink, and Kenneth P. Birman. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst.*, 36(2):4:1–4:49, 2019.

[28] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 295–306. ACM, 2014.

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 437–450. USENIX Association, 2016.

[30] 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 2016, Savannah, GA, USA, November 2-4, 2016*, pages 185–201. USENIX Association, 2016.

[31] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 1–16. USENIX Association, 2019.

[32] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In Frank Thomson Leighton and Peter W. Shor, editors, *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 654–663. ACM, 1997.

[33] Antonios Katsarakis, Vasilis Gavrielatos, M. R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 201–217. ACM, 2020.

[34] HP Labs. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/, 2014.

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

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

[37] Se Kwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K. Aguilera, Kimberly Keeton, and Vijay Chidambaram. DINOMO: an elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proc. VLDB Endow.*, 15(13):4023–4037, 2022.

[38] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: in-network memory management for disaggregated data centers. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 488–504. ACM, 2021.

[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, Shanghai, China, October 28-31, 2017*, pages 137–152. ACM, 2017.

[40] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 267–278. ACM, 2009.

[41] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*, pages 189–200. IEEE Computer Society, 2012.

[42] Compute Express Link. Compute express link: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/.

[43] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Digest of Papers: FTCS-27, The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, Seattle, Washington, USA, June 24-27, 1997*, pages 272–281. IEEE Computer Society, 1997.

[44] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 369–384. USENIX Association, 2008.

[45] 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, San Jose, CA, USA, June 26-28, 2013*, pages 103–114. USENIX Association, 2013.

[46] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 358–372. ACM, 2013.

[47] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 16:1–16:12. ACM, 2018.

[48] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos K. Aguilera. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pages 97–108. ACM, 2019.

[49] Brian M. Oki and Barbara Liskov. Viewstamped replication: A general primary copy. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988*, pages 8–17. ACM, 1988.

[50] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014.

[51] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[52] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodík, and Thomas E. Anderson. Floem: A programming system for nic-accelerated network applications. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 663–679. USENIX Association, 2018.

[53] Kai Ren, Qing Zheng, Swapnil Patil, and Garth A. Gibson. Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In Trish Damkroger and Jack J. Dongarra, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 237–248. IEEE Computer Society, 2014.

[54] 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 2020, Virtual Event, November 4-6, 2020*, pages 315–332. USENIX Association, 2020.

[55] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 69–87. USENIX Association, 2018.

[56] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. FUSEE: A fully memory-disaggregated key-value store (extended version). https://github.com/dmemsys/FUSEE/blob/main/documents/fast23_FUSEE_Extended_Version.pdf, 2023.

[57] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: when RPC is faster than server-bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 1–15. ACM, 2017.

[58] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. Tailwind: Fast and atomic rdma-based replication. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 851–863. USENIX Association, 2018.

[59] Jeff Terrace and Michael J. Freedman. Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In *2009 USENIX Annual Technical Conference, San Diego, CA, USA, June 14-19, 2009*. USENIX Association, 2009.

[60] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 33–48. USENIX Association, 2020.

[61] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 306–324. ACM, 2017.

[62] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 91–104. USENIX Association, 2004.

[63] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 261–280. USENIX Association, 2020.

[64] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. *CoRR*, abs/2112.07320, 2021.

[65] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with in-network cache coherence. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 277–292. USENIX Association, 2021.

[66] 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 2020, Virtual Event, November 4-6, 2020*, pages 117–135. USENIX Association, 2020.

[67] 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 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 233–251. USENIX Association, 2018.

[68] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 87–104. ACM, 2015.

[69] 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 Brian N.

Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 307–320. USENIX Association, 2006.

[70] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. Scaling replicated state machines with compartmentalization. *Proc. VLDB Endow.*, 14(11):2203–2215, 2021.

[71] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 191–208. USENIX Association, 2020.

[72] Yiying Zhang, Jian Yang, Amir Saman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 3–18. ACM, 2015.

[73] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 15–29. USENIX Association, 2021.

ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems

Pengfei Li, Yu Hua*, Pengfei Zuo, Zhangyu Chen, Jiajie Sheng
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
**Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

Abstract

Disaggregated memory systems separate monolithic servers into different components, including compute and memory nodes, to enjoy the benefits of high resource utilization, flexible hardware scalability, and efficient data sharing. By exploiting the high-performance RDMA (Remote Direct Memory Access), the compute nodes directly access the remote memory pool without involving remote CPUs. Hence, the ordered key-value (KV) stores (e.g., B-trees and learned indexes) keep all data sorted to provide rang query service via the high-performance network. However, existing ordered KVs fail to work well on the disaggregated memory systems, due to either consuming multiple network roundtrips to search the remote data or heavily relying on the memory nodes equipped with insufficient computing resources to process data modifications. In this paper, we propose a scalable RDMA-oriented KV store with learned indexes, called ROLEX, to coalesce the ordered KV store in the disaggregated systems for efficient data storage and retrieval. ROLEX leverages a retraining-decoupled learned index scheme to dissociate the model retraining from data modification operations via adding a bias and some data-movement constraints to learned models. Based on the operation decoupling, data modifications are directly executed in compute nodes via one-sided RDMA verbs with high scalability. The model retraining is hence removed from the critical path of data modification and asynchronously executed in memory nodes by using dedicated computing resources. Our experimental results on YCSB and real-world workloads demonstrate that ROLEX achieves competitive performance on the static workloads, as well as significantly improving the performance on dynamic workloads by up to 2.2× than state-of-the-art schemes on the disaggregated memory systems. We have released the open-source codes for public use in GitHub.

1 Introduction

Recent *disaggregated memory systems* separate memory, storage, and computing resources into independent pools [16, 34, 42] for high resource utilization, flexible hardware scalability, and efficient data sharing, which become prevalent in many datacenters and clouds [2, 3, 8]. The disaggregated system adopts the RDMA-capable networks for communications due to the salient features, such as high throughput (40-400 Gbps), low latency (a few microseconds), and remote CPU/kernel bypassing [12, 41, 51], which are widely supported by InfiniBand, RoCE, and OmniPath [16, 29, 36, 41, 49].

The disaggregated memory systems become important infrastructures [1, 17, 32, 34, 39, 40, 42] for various applications, including databases [27, 40] and in-memory key-value (KV) stores [12, 39, 44, 53]. Among them, tree-based and learned indexes are two widely used structures for the ordered key-value stores, which provide efficient range query performance via identifying items in a given range [7, 24]. In the disaggregated memory systems, the machines in compute and memory pools are respectively termed as *compute and memory nodes*, which are specialized for computing and storage purposes.

Deploying tree-based structures in the disaggregated memory system becomes inefficient, since the inner nodes consume much memory space and fail to be fully cached, thus resulting in multiple network roundtrips for traversing the entire tree. Various *index caching* schemes [31, 43, 52] propose to alleviate the network penalty via locally caching partial data, which however still suffer from unavoidable capacity misses due to the rapid growth of data.

Unlike them, XStore [44] proposes to cache the learned indexes for remote data accessing, since the learned models consume less memory footprints than tree-based structures by up to several orders of magnitude [14, 24]. By locally holding the whole learned index structure, a one-sided RDMA READ is sufficient for compute nodes to fetch remote data in the context of static (i.e., read-only) workloads. However, the design goal of XStore is not to exploit the strengths of disaggregated memory systems. Instead, XStore relies on the monolithic servers to process dynamic (i.e., read-write and write-intensive) workloads. Inspired by XStore, we adopt the similar idea and construct XStore-D on the diaggregated memory systems, rather than conventional monolithic context, by transferring data modification requests to memory nodes via RPCs. We observe that in fact XStore-D becomes inefficient to handle intensive modification requests, since the computing resources in the memory nodes are insufficient to meet the intensive computation requirements [39,53]. As a result, new models fail to be retrained in time and the stale models expand to a large prediction range to search dynamic workloads. The compute nodes have to consume more network roundtrips on determining the exact positions, since the positions dynamically change for data modifications. To avoid the penalty of large expansion, XStore-D transfers the subsequent requests to memory nodes until new models are retrained, which further increases the computing burden upon memory nodes. It is non-trivial to coalesce ordered KV stores in the disaggregated memory systems due to the following challenges.

1) Limited computing resources on memory nodes. Existing ordered KV stores rely on the monolithic servers to process write-intensive modifications [23,44]. However, the memory nodes in the disaggregated systems contain limited computation capability and fail to meet the requirements of computing-intensive operations, e.g., modifying the large B-tree and frequently retraining models. The CPU access bottleneck on the memory nodes decreases the overall system performance. Moreover, simply adding more CPUs to the memory pool for data processing decreases the scalability of the disaggregated memory systems, since the memory and computing resources fail to be independently scaled out [52].

2) Overloaded bandwidth for data transferring. Offloading data modifications to the compute nodes meets the computing requirements, which however rapidly fills up the entire bandwidth due to transferring massive data. Specifically, the compute nodes consume a large amount of network bandwidth to balance tree-based structures [7,30], e.g., multi-level nodes splitting and merging, as well as fetching a large amount of data to retrain models for the learned indexes [10, 14, 37]. The network bandwidth becomes insufficient to enable high performance for various data requests.

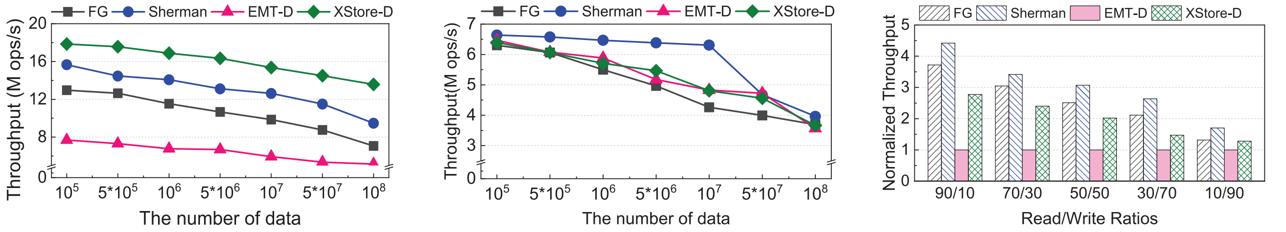
3) Inconsistency issue among different nodes. Guaranteeing data consistency among different nodes during modification is essential to prevent data loss. However, the inconsistent states occur when different compute nodes fail to atomically complete the data and model modification operations, e.g., multiple compute nodes compete for the same space to insert data and the local cache becomes stale when the models are updated. The main reason is that the atomic granularity of an RDMA operation is 8B, which is much smaller than the size of each index operation. The compute nodes require multiple network roundtrips to guarantee data consistency, incurring high overheads for consistency.

To address the aforementioned challenges, we propose a scalable RDMA-oriented key-value store using learned indexes, called ROLEX, for the disaggregated memory systems, which processes data requests on the compute nodes via one-side RDMA operations. The context of "scalable" means that ROLEX efficiently supports dynamic workloads and scales out to multiple disaggregated nodes. Although ROLEX adopts the similar idea with XStore on the static (i.e., read-only) operations, ROLEX is completely different with XStore in terms of the application scope, dynamic (i.e., data modification) operations, and the index structure on memory nodes. Specifically, ROLEX aims to efficiently support both static and dynamic workloads in the disaggregated memory systems. Unlike XStore, ROLEX does not maintain a B-tree on memory nodes to process modifications. Instead, ROLEX directly stores the sorted data in the assigned leaves (i.e., data arrays) on memory nodes. By judiciously decoupling the index operations and moving the retraining phase out of the critical path, the compute nodes efficiently modify the remote data via one-sided RDMA operations. When there are insufficient slots, ROLEX leverages a leaf-atomic shift scheme to atomically allocate a new leaf for accommodating more data. By using the retraining-decoupled index structure, ROLEX asynchronously retrains model in the memory pool when there are sufficient computing resources. The compute nodes identify new models through a shadow redirection scheme and synchronize the retrained models from remote nodes during the next reading. It is worth noting that the memory node generally includes dedicated computing resources provided by FPGA or ARM cores to offload low-computing requirement operations [17] (e.g., infrequent retraining in ROLEX), rather than all index operations.

We implement a prototype of ROLEX[1] and evaluate the performance via widely-used YCSB [47], two real-world, and two synthetic workloads. Our experimental results show that ROLEX achieves competitive performance with XStore-D [44] on static workloads, and outperforms state-of-the-art RDMA-based ordered KV stores by up to 2.2× on dynamic workloads. In summary, we have the following contributions:

- *Scalable ordered KV store for disaggregated memory systems.* We propose ROLEX to directly process data requests on the compute nodes via one-sided RDMA operations, which efficiently explores and exploits the hardware benefits of the disaggregated memory systems, as well as avoiding the computing resources bottleneck in the memory pool.
- *Retraining-decoupled learned indexes for one-sided RDMA execution.* We decouple the insertion and retraining operations for the learned indexes, and enable compute nodes to directly insert data without waiting for the model retraining. Non-retrained models are able to index newly inserted data using the proposed data-movement constraints.
- *Atomic remote space allocation.* When there are insufficient slots, the compute nodes leverage a leaf-atomic shift scheme to atomically allocate data arrays in the memory pool for accommodating new data. In ROLEX, no collisions occur among different machines due to the atomic metadata management.

[1] The source code is available at https://github.com/iotlpf/ROLEX.



(a) Read on static workloads.

(b) Write different numbers of data.

(c) The throughput of various read/write ratios.

Figure 1: The system performance for different schemes. *(a) Read and (b) write throughputs with different numbers of data, using 1 CPU core on memory nodes. (c) Normalized throughput with respect to EMT-D for hybrid read/write workloads.*

2 Background and Motivation

2.1 Disaggregated Memory Systems

The disaggregated memory systems breaks monolithic servers into independent network-attached components, which meets various application requirements via independently scaling out the hardware resources. Different nodes communicate with each other via Remote Direct Memory Access (RDMA) NICs, such as InfiniBand, RoCE, and OmniPath. The significant feature over the traditional network is that RDMA enables the compute nodes to directly access the memory nodes without involving remote CPUs via one-sided verbs, including `RDMA READ`, `WRITE`, and `ATOMIC` operations (e.g., compare-and-swap (`CAS`) and fetch-and-add (`FAA`)). It is worth noting that the granularity of the `ATOMIC` operation is 8B, and multiple `READ` and `WRITE` operations are completed via the doorbell batching [44] to reduce the network latency. Moreover, even though there are no powerful CPUs in the memory pool, each memory node generally includes dedicated computing resources provided by FPGA or ARM cores in NICs that are used for operation offloading [17], which efficiently supports the operation decoupling in ROLEX.

2.2 Network-Attached Ordered KV Store

This paper mainly focuses on the network-attached ordered key-value stores, including tree-based and learned indexes, which keep all data sorted and meet range query requirements.

Tree-based Structures. Tree-based structures [7, 20, 30] (e.g., B$^+$-tree) store data in the leaf nodes and construct multi-level inner nodes to search the leaves. However, the tree-based structures become inefficient to leverage one-sided RDMA for accessing remote data [44], since the local machine fails to cache the whole index structure and has to consume multiple RTTs (i.e., the network roundtrip time) for searching the inner nodes. Recent designs [31, 43, 52] cache top-level nodes on compute nodes to access the remote data. Among them, FG [52] designs a fine-grained B-link tree for the disaggregated systems, which distributes tree nodes across memory nodes and modifies trees with RDMA-based locks. Sherman [43] combines RDMA-friendly hardware and software features to deliver high write performance on the remote B-link tree, which optimizes the locking phase by constructing global locks on the on-chip memory of RDMA NICs. However, tree-based schemes inevitably incur multiple RTTs for retrieving inner nodes when the data overflow the limited local cache.

Learned Indexes. Learned indexes show significant advantages over tree-based structures in terms of searching speed and memory consumption, due to the easy-to-use and small-sized learned models. Specifically, the learned indexes view the process of searching data as a regression model, which record the positions of all data by approximating the cumulative distribution function (CDF) of the sorted keys [10, 14, 15, 24, 37]. The learned models achieve 2-4 orders of magnitude space savings than the inner nodes of the tree-based structures [14], which enables the local machine to cache the whole index structures and avoids the penalty of multiple RTTs to determine the remote data positions.

XStore proposes a hybrid index structure, i.e., maintaining a B-tree to process modifications and locally caching the learned indexes for remote data accessing. XStore [44] delivers high search performance due to only requiring one RTT to access the static workloads. For the dynamic workloads, XStore handles the data modification requests by modifying the B-tree on the memory nodes. At the same time, XStore expands the stale models to large prediction ranges to ensure that the newly inserted data are contained. However, such design becomes inefficient on the disaggregated memory systems, since the memory nodes have limited computing resources and fail to efficiently handle the intensive modification requests. The new models fail to be retrained in time and the stale models cause too low accuracy to search the remote data in one RTT due to the model expansion. As a result, the local cache becomes invalid and the subsequent data requests are transferred to the memory nodes via classic RPCs. The overall performance significantly decreases due to the limited computing resources of memory nodes.

2.3 Performance Analysis

We evaluate and analyze the performance of existing network-attached KV stores in the disaggregated memory system. Among them, FG [52] and Sherman [43] design RDMA-enabled B-link trees, enabling compute nodes to modify B-link trees via one-sided RDMA verbs. Moreover, we also equip the memory nodes with limited computing resources to analyze why RPC-based KV stores are inefficient for the disaggregated memory system, i.e., adopting the similar ideas of EMT-D (i.e., the Masstree [30] based on eRPC [23]) and XStore-D [44] on the computation-constrained memory nodes for evaluations.

Learned indexes outperforms tree-based structures on large-scale static workloads. Figure 1a shows the search throughput on static workloads. As the datasets constantly increase, XStore-D shows higher throughput than tree-based structures, since the compute nodes cache the whole learned index structure, rather than caching partial inner nodes for tree-based structures, avoiding multiple RTTs to determine the data positions. XStore-D obtains remote data within one RTT according to the prediction results of the learned models, while other schemes fail.

Index cache becomes invalid on dynamic workloads. Figures 1b shows the throughput on write-intensive workloads. We observe that XStore-D delivers lower performance than Sherman, since XStore-D sends requests to memory nodes via eRPC and relies on the limited computing resources of memory nodes to process modifications. The local cache of XStore-D is not fully exploited and becomes invalid during the modification phase, while Sherman delivers higher throughput via one-sided RDMA. However, the performance of Sherman decreases when storing a large amount of data, since the increased inner nodes overflow the local cache.

Disaggregated system requires efficient one-sided RDMA operations. Figure 1c shows the throughputs of different schemes with respect to EMT-D when configuring various read/write ratios. FG and Sherman show significant advantages over EMT-D, since all index operations are completed via one-sided RDMA. The performance of XStore-D significantly deceases when configuring large write ratios, due to failing to handle writes via one-sided RDMA operations.

3 ROLEX Design

3.1 Overview

We present *a scalable RDMA-oriented key-value store using learned indexes (ROLEX)* for the disaggregated memory systems. Unlike existing schemes, ROLEX does not maintain a B-tree on the memory nodes to process data requests. Instead, ROLEX constructs the retraining-decoupled learned indexes on the stored data and processes data requests on compute nodes via the one-sided RDMA operations. The challenges are how to efficiently avoid the collisions of various index operations in different compute nodes, as well as enabling all compute nodes to correctly identify the modified data with low consistency overheads. Our main insights are to execute index operations with atomic designs, and asynchronously retrain models by decoupling the insertion and retraining operations with consistency guarantees.

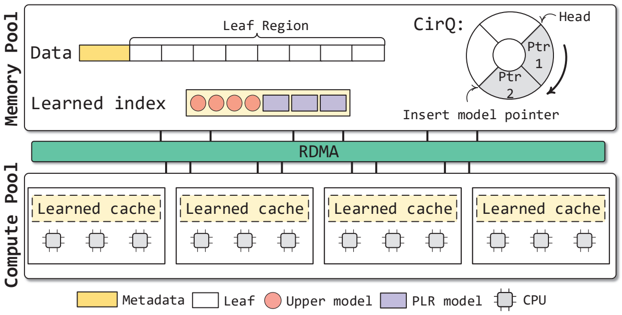


Figure 2: The design overview of ROLEX.

Figure 2 shows the overview of ROLEX. In the memory pool, ROLEX stores all data into fixed-size leaves (i.e., arrays) and constructs a retraining-decoupled learned index based on these data, as shown in Sections 3.2 and 3.3. To process dynamic workloads, the compute nodes directly modify the remote leaves without retraining models, since we decouple the insertion and retraining operations. By adding a bias and some data-movement constraints, the non-retrained models have the ability to correctly identify all data even after inserting new data. To construct sufficient data leaves for the new data with one-sided RDMA, we present a *leaf-atomic shift scheme* in Section 3.4, which also keeps all data sorted for range queries and avoids the collisions among different compute nodes. The stale models need to be retrained for high accuracy when a large amount of data are modified. Although the compute nodes have sufficient computing resources for retaining, obtaining all the pending retraining data from memory nodes consumes much network bandwidth. Instead, we observe that the retraining overheads mainly come from data merging and resorting, while the complexity of the training algorithm is only $O(N)$. Hence, the limited computing resources on memory nodes are sufficient to retrain the models, especially after we have offloaded most index operations to the compute nodes and moved the retraining phase out of the critical path. With the aid of *leaf tables*, ROLEX *asynchronously* retrains models in-place on the memory nodes, as shown in Section 3.5. After retraining, ROLEX updates the models in the memory pool using *the shadow redirection scheme*, while the compute nodes won't synchronize the retrained models until the next reading.

3.2 Retraining-decoupled Learned Indexes

The challenges of coalescing the learned indexes on dynamic workloads come from the high overheads of keeping all data sorted and avoiding data loss from the learned models during insertion. The reason of data loss is that the models record the positions of the trained data after training, while failing to find the new positions after inserting many new data unless retraining. As shown in Figure 3, the red line represents a linear regression model that is trained on the black points (i.e., the trained data). All data are found in the prediction range,

$[pred - \varepsilon, pred + \varepsilon]$ (i.e., the blue block), as long as the data are not moved out of this range, where ε is the predefined maximum model error. When some new data are inserted, point a moves backward to a', which is out of the prediction range. To record the new positions, the models are retrained via step-by-step operations, including resorting data, retraining models, and synchronizing models to all compute nodes. The system is blocked until the retraining and synchronization are completed, thus incurring a long latency and decreasing the overall system performance.

In fact, we observe that the learned indexes don't require frequent retraining as long as the non-retrained models can find all data. This observation offers an opportunity to address the dilemma in coalescing the learned indexes in the disaggregated memory systems, i.e., new data are written to the memory pool without waiting for retraining. To achieve this design goal, we modify the training algorithm and add some constraints to help the non-retrained models always find all data without retraining.

Training Algorithm. Leveraging multiple linear regression models is a common way to learn the data distribution due to the efficiency of training and memory savings [10, 14, 15, 24]. We use an *improved OptimalPLR* algorithm to train the piecewise linear regression (PLR) models, since OptimalPLR algorithm [46] has been proved to have the minimal number of PLR models while incurring small time and space complexity ($O(N)$). The key idea of OptimalPLR is to construct multiple optimal parallelograms with 2ε width on the trained data, where the optimal parallelogram is defined as a parallelogram of 2ε width in the vertical direction such that no trained data are placed outside of the parallelogram, as the blue blocks shown in Figure 3. We thus obtain the linear regression model that intersects the two vertical sides and bisects the parallelogram.

$$\begin{aligned} \varepsilon &>= max|f(X_i) - Y_i| \quad \forall i \in (0, N) \\ P_{range} &= [f(X_i) - \varepsilon - \delta, f(X_i) + \varepsilon + \delta] \end{aligned} \tag{1}$$

To ensure that the trained models find all data even after insertions, we improve the OptimalPLR algorithm by adding a bias (represented as δ) to the prediction calculation, as well as adding some constraints on the data movements. As shown in Equation 1, the optimal parallelogram is determined by guaranteeing that the distances between the predicted ($f(X_i)$) and true (Y_i) positions of all data are not larger than the predefined maximum model error (ε), while the prediction range (P_{range}) is calculated by adding an extra δ. Hence, the area covered by the prediction ranges of all data is larger than the determined optimal parallelogram, i.e., we extend the blue block to the yellow one, as shown in Figure 3. In this case, the models don't require retraining as long as the data move no more than δ positions, since the δ data movements won't exceed the prediction range.

Data-movement constraints. Simply adding a bias to the prediction calculation is insufficient to achieve the design goal of operation decoupling, since the data move more than δ positions when inserting/deleting a large amount of data. To further address these issues, we add some constraints on the data movements.

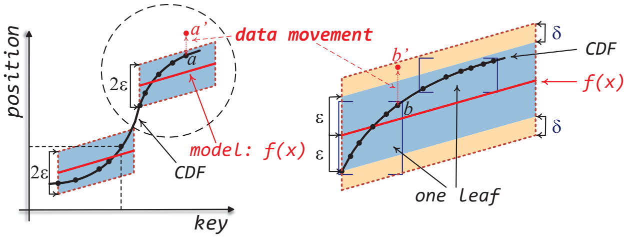


Figure 3: The retraining-decoupled learned indexes.

- *Moving data within fixed-size leaves.* We store the data into fixed-size arrays (termed as *leaves*) in the training phase, and each leaf contains at most δ data. All data are only allowed to be moved within their assigned leaves. In this case, we identify all data via existing trained models since no data move out of P_{range} calculated from Equation 1. Furthermore, we transfer the position prediction to the leaf prediction, i.e., the learned models provide a range of leaves that may contain the queried data via Equation 2. Due to not moving out of the assigned leaves, no data are lost. In the disaggregated memory systems, the leaves in L_{range} are easily obtained via one-sided RDMA verbs.

$$L_{range} = [\frac{f(X_i) - \varepsilon}{\delta}, \frac{f(X_i) + \varepsilon}{\delta}] \quad \forall i \in (0, N) \tag{2}$$

- *Synonym-leaf sharing.* We allocate a new leaf (nl) to accommodate more data when a leaf (l) has insufficient slots, where nl shares the same positions (i.e., the labels used for training) with l. We define nl as a *synonym leaf* of l, which is linked via a pointer. The data of synonym leaves move within each other to facilitate data sorting. Since nl doesn't change the positions recorded by models, the learned indexes still calculate L_{range} via Equation 2. Moreover, we need to search the synonym leaves referred by L_{range}, since the data may locate in the predicted and synonym leaves.

The non-retrained models have the ability to find all data without retraining, since no data move out of the predicted leaves. We hence decouple the insertion and retraining operations for the learned indexes.

3.3 ROLEX Structure

To exploit the hardware benefits of the disaggregated memory systems, ROLEX stores data on the memory nodes while processing requests on the compute nodes, as shown in Figure 2.

Memory pool stores data. Driven by the operation decoupling, we store all data into fixed-size leaves and train a learned index on these data using our improved training algorithm. All leaves are stored in a continuous area (termed as *leaf region*) allocated from an RDMA-registered memory region. The structure of the leaf region is shown in Figure 2,

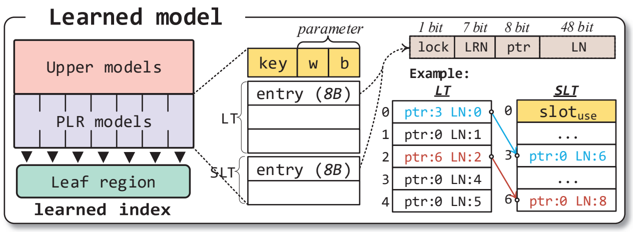


Figure 4: The structure of the learned models.

where the first two 8B data are respectively used to indicate the number of leaves that have been allocated (*alloc_num*) and the total number that the leaf region can allocate. The remaining leaf region stores a large number of leaves, and each leaf contains δ pairs of keys and values[2]. To allocate a new leaf, we read *alloc_num* and write it back with (*alloc_num*+1) via the atomic FAA. We store data into the leaf pointed by the obtained *alloc_num*. The leaves are accessed via adding offsets to the start position of the leaf region. Moreover, the fragmentation and garbage collection can be efficiently mitigated in ROLEX, since ROLEX allocates and reclaims space via fixed-size leaves that are accessed via the atomic-size leaf numbers.

We train multiple PLR models on the stored leaves and each model consists of four parts, including the covered smallest key, the model parameters, a leaf table (LT) and a synonym leaf table (SLT), as shown in Figure 4. LT and SLT store the leaf numbers (i.e., the *alloc_num* when being allocated) to access leaves. It is worth noting that different models independently record the data positions for training, which become easy to be updated since no position dependency exists among models. The obtained PLR models are indexed by training *upper models* on the smallest keys, where the upper models don't contain leaf tables. We repeat this procedure and construct multi-level models like PGM-index [14] due to the small space consumption, which are fully cached in the compute nodes. Moreover, we store the models with pointers, which efficiently support our shadow redirection scheme to update models, as shown in Section 3.5.

Compute pool caches indexes. The memory pool is shared across compute nodes, which supports the system scalability. Specifically, the newly added compute nodes identify the shared memory pool via the RNIC, which obtain the starting addresses of the model and leaf regions. After reading the learned models from the model region, the new compute nodes efficiently access the remote data according to the prediction range of the learned models, where the entry in the prediction range contains the leaf region number and the leaf number, thus indicating the locations of the required data in the memory pool. ROLEX processes various data requests (e.g., search, update, insert, and delete) on compute nodes with one-sided RDMA operations.

3.4 One-sided Index Operations

Simply executing data modification operations on compute nodes incurs two challenges, i.e., long latency of multiple remote operations and inconsistency issues among different machines. For example, on dynamic workloads, conflicts occur when different compute nodes write data at the same address in the memory pool, and inconsistencies occur when one node constructs new leaves while not notifying others. The 8B-atomic RDMA verbs fail to guarantee the data consistency among different machines, since the moved data during insertion are larger than 8B. An intuitive solution is to modify data leaves and LTs with locks, as well as broadcasting other nodes to synchronize their indexes after modifications. However, other nodes could not access or insert data due to the consistence requirement from the locks until the modification completes, which blocks the systems for a long time.

To address these problems, we propose a *leaf-atomic shift scheme* that provides consistence guarantees for concurrently modifying data via compute nodes while requiring few remote RDMA operations. The key insights are to atomically assign the write regions in the shared memory pool for different compute nodes, and enable each compute node to access data via the stale index structure. Specifically, we first show the structures of LT and SLT that are designed for the leaf-atomic shift scheme, and then respectively elaborate how different index operations coalesce with this scheme.

The structures of LT and SLT. We leverage the 8B *alloc_num* in the leaf region to enable the lock-free leaf allocations via FAA, as well as using 8B entries in LT to enable the consistent leaf modifications. The structures of LT and SLT are shown in Figure 4. The first slot in SLT is preserved to indicate how many slots ($slot_{use}$) of SLT have been used, which is modified when constructing new synonym leaves. Other slots of LT and SLT store 8B entries, each of which consists of a lock (1 bit), a leaf-region number (7 bits), a pointer (8 bits) and a leaf number (48 bits). The lock is lightweight and fine-grained due to only locking the current leaf rather than all leaves under the model. We use the leaf-region and leaf numbers to determine the leaves, while the pointer points to an offset of SLT to link the synonym leaf. For example, as shown in Figure 4, the pointer of leaf 0 points to 3, indicating that leaf 0 has a synonym leaf stored in the 3rd position of SLT, while this synonym leaf is stored in the 6th position in the leaf region. The size of LT is determined in the training phase, while the size of SLT is fixed to contain 2^8 slots. In our design, each leaf region registers up to 2^{48} leaves, while a model is able to construct up to (2^8-1) synonym leaves. It is worth noting that the max number of each field can be adjusted by specifying the bits in the entry of LT.

Point query. For a given key, the compute node searches remote data via the following steps: ① Predict L_{range} with the local learned indexes according to Equation 2. ② Translate the leaf positions into physical addresses by looking up LT

[2] Similar to prior RDMA-based schemes [31, 43, 44], ROLEX stores 8B values or 8B pointers for variable-length values.

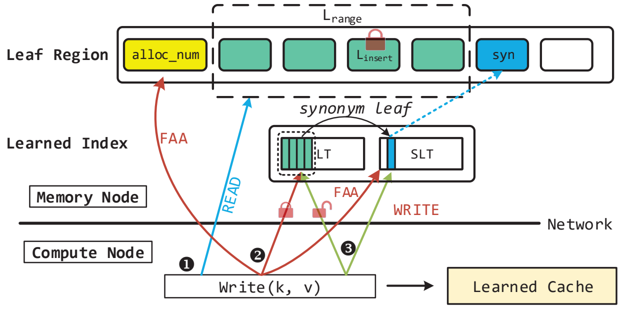


Figure 5: The worst-case insertion of ROLEX.

and SLT. As shown in Figure 4, we lookup the 1st-3rd entries in LT when L_{range} predicts $[1,3]$, and further read the synonym leaf number in the 6th slot of SLT when the 2nd entry points to 6. The physical address (phy_addr) of a remote leaf is calculated via Equation 3, i.e., multiply the leaf number (l_{num}) by the leaf size (l_{size}) and plus the address of the first leaf in the leaf region (LR_{addr}). ③ Read leaves with doorbell batching according to the physical addresses. ④ Search the fetched leaves, and further read the value according to the value pointer. ROLEX leverages the checksum-based schemes like existing KV stores [12, 44, 45] to guarantee the consistency of the read leaves.

The LT and SLT change when constructing new leaves in the memory pool, which is identified by the compute nodes when the first slot (i.e., $slot_{use}$) of SLT changes in the doorbell-batch reading. The compute nodes synchronize remote LT and SLT, and read the new leaves for data consistency.

$$phy\_addr = l_{num} * l_{size} + LR_{addr} \tag{3}$$

Range query. A range query for $[K,N]$ requires N items starting from K. Apart from the leaves in L_{range}, ROLEX reads another (N/δ) adjacent leaves to ensure that at least N items after K are fetched. Like point query, ROLEX reads all required leaves via a doorbell batching.

Insert. ROLEX executes the insertion operation on compute nodes via the following phases:

❶ *Fetching.* The compute node (represented as C_{node}) fetches the remote leaves like point query, without reading synonym leaves in this phase, since the latest synonym leaves will be fetched after acquiring the lock.

❷ *Fine-grained locking.* C_{node} determines the leaf to be inserted (represented as L_{insert}) according to the data order, and locks L_{insert} by changing the lock bit of LT entry to 1 with `CAS`. After locking, C_{node} reads L_{insert} and its synonym leaves to ensure that the data are up to date. The synonym leaves share the same lock with the trained leaf to enable the atomic lock. Even if L_{insert} and its synonym leaves are modified by other compute nodes before being locked by C_{node}, inserting data into these leaves still keeps all data sorted, since the data of L_{insert} are only allowed to move within L_{insert} and its synonym leaves.

❸ *Writing and unlocking.* C_{node} inserts data into the fetched leaves according to the data order and unlock L_{insert} via `CAS`.

When the fetched leaves have insufficient empty slots, C_{node} constructs a new synonym leaf as shown in Figure 5. Within one doorbell batching, C_{node} fetches and increases $alloc\_num$ of the leaf region and $slot_{use}$ of SLT by 1 via `FAA`. Furthermore, C_{node} writes the new synonym leaf in the memory pool according to the physical address calculated by Equation 3, and inserts the $alloc\_num$ of the newly constructed synonym leaf into SLT at position $slot_{use}$. C_{node} also changes the pointer field of L_{insert} to the new leaf and unlocks L_{insert} via `CAS`.

For optimizations, other threads of C_{node} can leverage the acquired lock to modify the same leaves, and the operations of writing leaves and modifying leaf tables are completed in one doorbell batching to improve the performance.

Update. C_{node} fetches the remote leaves like point query. When the given key is matched in one of the fetched leaves, C_{node} locks and re-reads the corresponding leaf to ensure that the data are up to date. The compute node updates the key-value item and unlocks the remote leaf.

Delete. To delete the data K, C_{node} ❶ fetches and ❷ locks the remote leaves like insertion operations, e.g., C_{node} fetches the leaf L_1 and its synonym leaves L_{5-8}. When K is identified in one of the fetched leaves, e.g., L_6, C_{node} removes K in L_6, while other leaves are not modified. When L_6 becomes empty after deleting K, C_{node} removes L_6 by modifying the leaf table, i.e., linking $L5$ to $L7$. ❸ C_{node} writes L_6 to memory nodes and unlocks the leaves. Moreover, the empty trained leaf L_1 is not removed until next retaining to avoid the prediction error, as shown in Section 3.5. Other compute nodes identify the deleted leaf when observing that the data in the synonym leaves are not sorted, which further synchronize the leaf tables and read the remote data.

3.5 Asynchronous Retraining

The retraining overheads come from the data resorting and retraining algorithms [37, 46]. An intuitive solution is to conduct retraining on compute nodes, which however consumes a large amount of available network bandwidth for transferring the pending retraining data. Instead, we observe that all data have been sorted by the leaf tables (i.e., LTs and SLTs) during the runtime, and the OptimalPLR algorithm has a low complexity (i.e., $O(N)$) [46] to train data, where N represents the number of the training data. Hence, ROLEX asynchronously retrains data in-place on the memory nodes to achieve an efficient trade-off between the network consumption and computing resource utilization. After offloading most index operations to the compute nodes, our experimental results show that the limited computing resources (e.g., one CPU core) on memory nodes are enough for retraining, as shown in Section 4.5.

ROLEX maintains a circular queue (CirQ) to identify the pending retraining models, and concurrently retrains models using the shadow redirection scheme without blocking the systems. Specifically, the compute nodes insert the pointer of a model at the end of CirQ when the model consumes 2^7 slots of SLT. The memory nodes periodically check the head of CirQ for retraining, which retrains models in the background and constructs a new LT to merge the old LT and SLT, while the compute nodes concurrently access the old models. Both new and old models access the same data via their own leaf tables. After retraining, the memory nodes replace the models with consistency guarantees.

Consistency guarantee. Figure 6 shows the consistency guarantee when the memory nodes concurrently retrain the leaves L_{1-5}, where L_5 is a synonym leaf of L_3. During retraining, the compute nodes concurrently modify the data, which lead to inconcistency when the positions of the data are not retrained by the new model, e.g., 1) constructing a new synonym leaf L_8 of L_5 and 2) moving data within the synonym leaves. ROLEX ensures the data consistency by redirecting the non-retrained data into a new SLT for the new model.

1) ROLEX identifies the newly constructed leaf (e.g., L_8) by checking the leaf tables of both old and new models, where the entry appearing in old LT or SLT but not appearing in the new LT is identified as a non-retrained leaf. When replacing the old model with the new one after retraining, ROLEX locks the old model and inserts L_8 to the new SLT, as well as changing the model pointer to the new model before unlocking, as shown in Figure 6. Hence, the new model correctly identifies L_8 by accessing the new leaf tables, and the compute nodes correctly identify the new model by checking the model pointer. Similarly, the removed leaves are identified by checking both old and new leaf tables.

2) ROLEX identifies the new positions of the moved data by checking the previous trained leaf. As shown in Figure 6, before the retraining begins, we respectively represent the leftmost and rightmost data in each leaf as X_l and X_r, e.g., X_{3l} represents the leftmost data of L_3. During retraining, the old model inserts the new item 15 in L_3, and inserts the items 18 and 24 into the newly constructed synonym leaf L_9. The challenge is to ensure that the new model correctly identifies the data modified by the old model, including the trained data in the leaves (e.g., the data between X_{3l} and X_{3r}) and the new data between two sorted leaves (e.g., the data between X_{3r} and X_{5l}). According to Equation 2, the new model predicts the data between X_{3l} and X_{3r} in L_3 due to recording these data in L_3 when the retraining begins. The new model correctly identifies these modified data in the synonym leaves by checking the new SLT. However, the inconsistent state occurs for the data between X_{3r} and X_{5l} (e.g., 24), since the new model may predict these data in L_5 but overlook L_3 and L_9. To avoid such error, ROLEX checks the previous leaf (i.e., L_3) to correctly identify the modified data.

ROLEX doesn't need to resort or move any data for retraining, since all data have been sorted by the leaf tables during the runtime. No data are lost during retraining, since all leaves are either retrained by the new model or being inserted into the new SLT.

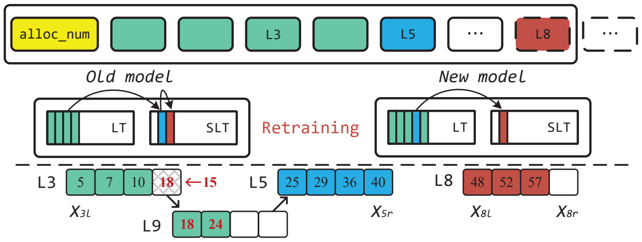


Figure 6: The consistency guarantee of concurrent retraining.

ROLEX inserts the new data in the synonym leaves, which triggers retraining when the synonym leaves consume half of (i.e., 2^7) the slots in the SLT. Before the retraining completes, SLT still contains the space to create 2^7 more synonym leaves to insert new keys. After retraining, the new models include new SLTs to accommodate more data. In our experiments, each leaf contains 16 slots and the model totally inserts 2,048 data before being retrained, while a model covers on average 1,465 trained data. Hence, a retraining is triggered when inserting about 1× new data than the trained data, having a low retraining frequency. The speed of retraining models is much faster than that of filling all synonym leaves. Moreover, ROLEX has a priority queue to identify and train the model with almost full SLT to avoid the scenario where a model has insufficient slots in SLT.

3.6 System Discussions

Scalability. ROLEX distributes large datasets across multiple memory nodes by constructing multiple leaf regions. Specifically, 2^7 leaf regions form a *group* and each region contains at most 2^{48} leaves to store data. A leaf group hence contains 2^{55} leaves and is sufficient to construct a large number of learned models. By training data in the same group, the learned models become efficient to determine the location of a leaf via the leaf-region (7 bits) and leaf numbers (48 bits) of the entry in LT and SLT. Moreover, ROLEX constructs multiple groups to scale across multiple memory nodes and becomes efficient to accommodate a large amount of data.

Durability and fault tolerance. Existing disaggregated memory systems enable the durability and fault tolerance in different ways, such as the persistent memory [39,50], battery-backup system [12], and logging writes [44], while achieving efficient performance. All these solutions are orthogonal to ROLEX for efficient durability and fault tolerance.

Emerging heterogeneous technology. ROLEX benefits from the technology integrating emerging accelerators and specialized hardware into the disaggregated memory nodes [17], due to the sufficient computing resources. Moreover, the powerful network technology [32] incurs low network penalty on remote data accessing. In this case, ROLEX needs a fallback mechanism to avoid the lock contention among many compute nodes, which is our future work.

4 Performance Evaluation

4.1 Experimental Setup

We run all experiments on a cluster with 3 compute nodes and 3 memory nodes, and each server node is equipped with two 26-core Intel(R) Xeon(R) Gold 6320R CPUs @2.10Ghz, 188GB DRAM, and one 100Gb Mellanox ConnectX-5 IB RNIC. The RNIC in each machine is connected with a 100Gbps IB switch. We limit the computing resources utilization (i.e., 1 CPU core in our testbed) for the memory node, which is reasonable due to the fact of the limited computing capability in the typical memory pools [17, 43]. During the initialization, the memory pool registers memory with huge pages to avoid the penalty of the page translation cache misses. The registered memory consists of the model and leaf regions to respectively maintain the learned models and data. Existing RNIC hardware doesn't support remote memory allocation [53], and we hence pre-allocate memory for the leaf region to support our proposed atomic-leaf shift scheme. All compute nodes run with 24 threads by default.

Workloads: We use YCSB [47] with both uniform and Zipfian request distributions to evaluate the performance, which contains 6 default workloads, including (A) update heavy (50% updates), (B) read mostly (95% read), (C) read only, (D) read latest (5% insert), (E) short ranges (95% range request), and (F) read-modify-write (50% modifications). Apart from these workloads, we also evaluate the performance under write-intensive requests with 2 real-world, and 2 synthetic datasets [24]. Among them, *Weblogs* and *DocID* respectively contains 200 and 16 million key-value pairs with different data distributions. The two synthetic datasets contain 100 million items, and respectively meet the normal and lognormal data distributions. We configure all workloads with 8B keys and pointers (i.e., refer to variable-length values) like existing schemes [24, 44] for comprehensive evaluations.

Counterparts for Comparisons: We compare ROLEX with four state-of-the-art distributed KV stores. Specifically, FG [52] and Sherman [43] design RDMA-enabled B-link trees for the disaggregated memory systems. We directly run the source codes of Sherman. Since FG is not open-source, we implement FG from scratch faithfully following the original design principles, as well as caching the top-level nodes on compute nodes for better performance. We also adopt the similar ideas of EMT-D [23] and XStore-D [44] on the disaggregated systems, i.e., using the limited computing resources of memory nodes to show the performance of RPC-based schemes. EMT-D transfers all requests to memory nodes via eRPC (RDMA-based RPC), while XStore-D accesses read-only workloads via compute nodes and relies on memory nodes to process write-intensive requests. We configure our implemented ROLEX with 16 slots in each leaf, as well as setting 16 as the maximum model error to train PLR models for efficient system performance. We further leverage 1 CPU core on the memory node and disable the garbage collection and durability functions for all counterparts to facilitate fair comparisons.

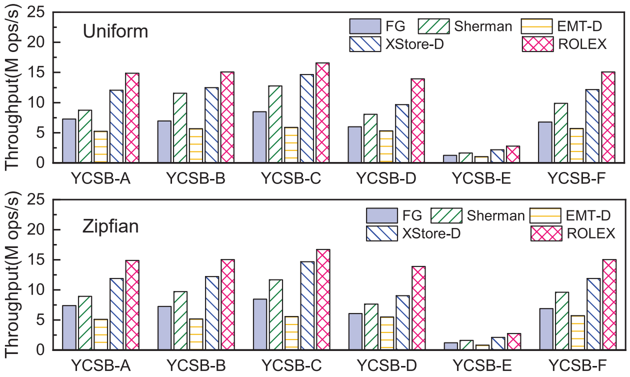


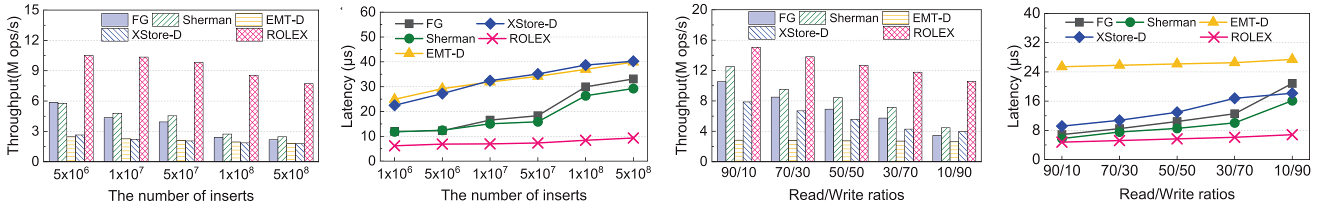
Figure 7: The throughputs on various YCSB workloads.

4.2 Overall Performance in YCSB

Figure 7 shows the throughputs on various YCSB workloads with both Uniform and Zipfian distributions. In general, ROLEX achieves competitive performance with XStore-D on static workloads, while achieving higher throughput on dynamic workloads due to not relying on remote CPUs.

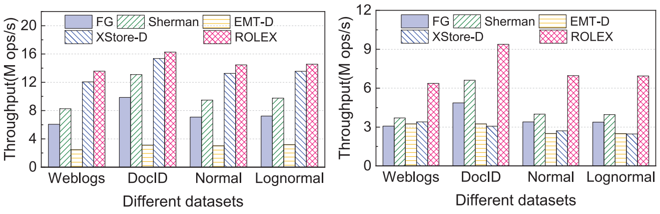
Static workload (YCSB C). On the static workloads, XStore-D and ROLEX efficiently read remote data via one `RDMA READ` according to the prediction results of the learned models, which achieve higher performance than FG and Sherman due to fewer RTTs caused by the local cache. EMT-D achieves the lowest throughput, since the memory nodes have insufficient computing resources to process the data requests. ROLEX achieves higher performance than XStore-D due to the high model accuracy. Specifically, ROLEX leverages the OptimalPLR algorithm [46] to train models according to the data distributions, which guarantees that all model errors are smaller than the predefined threshold. However, XStore-D leverages the recurve model index scheme [24] for training and fails to adaptively train models according to the data distribution. Some model errors are large when failing to train sufficient models, causing a large prediction range and lower performance than ROLEX in the read-only workloads.

Read-write workloads (YCSB A, B, D, F). For data modifications, both XStore-D and EMT-D transfer data requests to the remote side and achieve low throughput, due to the limited CPU cores on memory nodes. The performance of FG and Sherman is limited by the local cache due to the large memory footprint of inner nodes. ROLEX achieves higher performance than other schemes due to exploiting the learned local cache with the efficient one-sided `RDMA WRITE`. Specifically, ROLEX outperforms FG, Sherman, EMT-D, and XStore-D by up to 2.1×, 1.7×, 2.8×, and 1.3× on workload A, since ROLEX directly updates the remote data without involving remote CPUs. For workload D, 5% insertions are mixed with 95% searches, and ROLEX improves the throughput by about



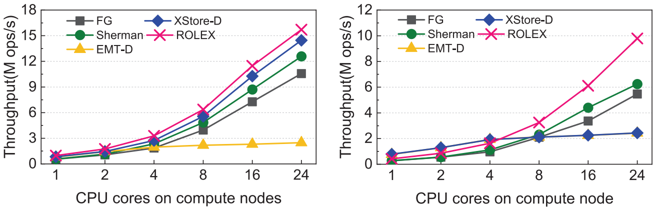
(a) Write-only throughput. (b) Write-only latency. (c) Hybrid read/write throughput. (d) Hybrid read/write latency.

Figure 8: The performance with various read/write scenarios.



(a) Read throughput. (b) Write throughput.

Figure 9: The performance under various data distributions.



(a) Read throughput. (b) Write throughput.

Figure 10: Scalability with various CPUs on compute nodes.

1.5× over other schemes. The reason is that the caches of other schemes become invalid during insertion, while ROLEX leverages the stale cache to write data in synonym leaves. We obtain the similar observations on workloads B and F.

Range-query workload (YCSB E). Workload E contains 95% range query and 5% insert requests. We observe that ROLEX improves the performance by 67% over other schemes, since all data are kept sorted in the synonym leaves during insertion and the range queried data are fetched in a doorbell batching by `RDMA READ`.

4.3 Performance in Various Scenarios

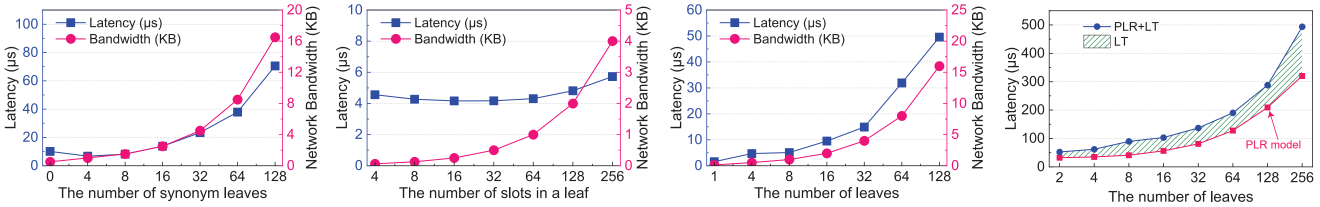
Apart from YCSB, we have the similar observations on other representive workloads, including Weblogs, DocID, Normal, and Lognormal. Figure 8 shows the performance of different schemes in various scenarios.

Throughput with intensive writes. Figure 8a shows the throughput of inserting different numbers of data. As we constantly insert data, ROLEX achieves significant performance improvements over other schemes. Specifically, ROLEX improves the insert throughput by up to 2.1×, 1.8×, 4.5×, and 4.3× over FG, Sherman, EMT-D, and XStore-D. The main reason is that the local cache is fully exploited by ROLEX with one-sided RDMA operations, while the footprints of inner nodes in tree-based schemes overflow the cache and the remote CPUs limit the write performance of RPC-based schemes. Moreover, we evaluate the latencies of the insert operations for different schemes, and the results are shown in Figure 8b. We observe that ROLEX incurs low latency since the stale cache identifies the leaf to be inserted according to the prediction results of the learned models. For the monotonically increasing keys, ROLEX shows low performance when multiple compute nodes contend for the same leaf lock, which is alleviated by sharing the leaf lock among multiple threads of the same compute node.

Performance with hybrid read-writes. Figures 8c and 8d respectively show the throughput and latency under various read/write ratios. The performance of EMT-D doesn't decrease much with the increasing write ratios, since the remote memory nodes suffer from the bottleneck of insufficient computing resources and achieve low performance even under intensive read requests. XStore-D achieves high performance on read-heavy workloads, while significantly decreasing the performance as the write ratio increases, because XStore-D reads data with one-sided RDMA while transferring most data requests to the remote side as the number of write requests increases. ROLEX, FG, and Sherman achieve higher performance than other schemes due to not being limited by the remote CPUs. ROLEX improves the throughput by 2.2× and 1.7× over FG and Sherman, since the improvements mainly come from the efficient learned local cache. FG and Sherman have to spend multiple RTTs on retrieving the remote data when the inner nodes overflow the limited local cache.

The latency of ROLEX is lower than that of RPC-based schemes in the disaggregated memory systems, since the latency of accessing remote data comes from the network roundtrip and the index structure traversal. ROLEX traverses the cached learned indexes via the compute nodes, while RPC-based systems traverse the index structures via the memory nodes. In the disaggregated memory systems, the compute nodes have sufficient computing resources to support high concurrent access, while however the memory nodes have limited computing resources and fail to meet the requirements for processing intensive index requests.

Performance with various data distributions. The data distributions impact the model accuracy of the learned indexes, which decrease the performance when the learned models deliver low accuracy. Figure 9 shows the throughput on various workloads with different data distributions, including



(a) Read with synonym leaves.
(b) Write different sized leaves.
(c) Write multiple leaves.
(d) The latency of training leaves.

Figure 11: In-depth Analysis. *We evaluate the latency and network bandwidth consumption when reading/writing/training different numbers of synonym leaves.*

Weblogs, DocID, Normal, and Lognormal. We observe that ROLEX achieves higher read performance than XStore-D. The main reason is that the improved OptimalPLR algorithm trains independent linear regression models with high accuracy according to the data distributions.

4.4 Scalability Performance

Figure 10 shows the throughput of various schemes with different numbers of cores on the compute nodes. We observe that the performance of EMT-D doesn't increase when configuring more cores on compute nodes, since the bottleneck of EMT-D are the remote CPUs of memory nodes, rather than the compute nodes. The throughputs of other schemes increase with the number of cores on compute nodes, as shown in Figure 10a, because FG, Sherman, XStore-D, and ROLEX don't rely on the remote CPUs to process the read requests. However, the write performance of XStore-D fails to scale out with the number of cores on compute nodes, as shown in Figure 10b, since XStore-D quickly runs out the available computing resources of the memory nodes. The read and write performance of ROLEX increases with the increasing number of cores on compute nodes, since different threads don't block each other.

If the disaggregated memory system is not assumed, in our evaluation, EMT-D and XStore-D achieve higher performance than other designs when configuring the memory nodes with more than 20 CPU cores, since 20 CPU cores in memory nodes meet the requirements of processing various index operations. However, it is worth noting that our paper mainly focuses on the disaggregated memory systems, which generally configure limited computing resources (i.e., much lower than 20 CPU cores) on the memory nodes.

4.5 In-Depth Analysis

We conduct three optimizations in ROLEX, including operation decoupling, one-sided indexing, and asynchronous retraining, which efficiently support the system to obtain high performance. We evaluate the efficiency of different optimizations in Figure 11.

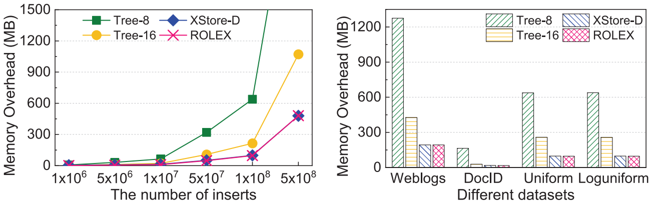
Operation decoupling. An important insight of ROLEX is that we decouple the insertion and retaining operations to enable the compute nodes to directly insert data to the memory pool, which leverages the stale models to identify the new data. As shown in Figures 11(a-d), although retraining incurs long latency, ROLEX achieves low latency to read and write remote data, since the operation decoupling moves the retraining phase out of the critical path and enables the compute nodes to insert data without waiting for the retraining.

One-sided indexing. The compute nodes access remote data via one-sided indexing, which incurs low latency and bandwidth consumption when operating on a small range of data, since one-sided indexing efficiently exploits the benefits of RDMA doorbell batching. We observe that ROLEX achieves high performance when respectively setting ε and δ to [8, 256] and [8, 128], which achieve an efficient tradeoff between the accessing efficiency and the retraining frequency. Specifically, ε and δ respectively represent the maximum prediction error and the leaf size. As shown in Figures 11a and 11c, a large ε provides a large prediction range, which consumes much network bandwidth and latency to identify the requested data. ROLEX achieves high performance when reading/writing 8-256 data, where the number of data is calculated by multiplying the size and the number of the leaves. Moreover, the small δ provides small-size leaves, which frequently triggers retraining since the leaves have insufficient slots to accommodate new data. However, as shown in Figure 11b, too large δ consumes much network bandwidth for modifying remote data, since ROLEX reads/writes data in the granularity of a leaf.

Asynchronous retraining. ROLEX asynchronously retrains the models to construct new models and leaf tables, which increases the model accuracy to read and write few leaves. As shown in Figure 11, the operations upon a small number of leaves significantly reduce the latency and network bandwidth consumption. Figure 11d shows the retraining latency using a single CPU core. We observe that training models and constructing leaf tables on 128 leaves consume about 300μs. Unlike conventional learned indexes [10, 14, 37], ROLEX doesn't need to move or resort any data during retraining, since all data are kept sorted during data modifications.

4.6 Overhead Analysis

Figure 12 shows the memory footprints of the metadata in different schemes, where the metadata refer to the data that



(a) Different numbers of data.

(b) Different datasets.

Figure 12: The memory footprints of the metadata. *Tree-# represents that an inner node contains # keys.*

are required for caching. For example, the metadata consist of the inner nodes for the tree-based schemes, while consisting of trained models and leaf tables for XStore-D and ROLEX. We observe that the memory overheads in tree-based structures rapidly increase with the increasing data, because many levels of inner nodes are constructed for indexing. Moreover, the metadata overheads significantly increase when using small inner nodes due to requiring more levels. Unlike tree-based structures, XStore-D and ROLEX leverage the linear regression models for indexing, and each model only contains 2 parameters and is much smaller than the inner nodes. As shown in Table 1, the memory overhead of ROLEX mainly comes from the LTs, which accounts for 98% of the total memory consumption. These models can be fully cached by the compute nodes, while the LTs can be fetched as needed when the limited cache fails to maintain all LTs.

In general, the compute overhead comes from the training algorithm with O(N) complexity, where N represents the number of trained data. On average, ROLEX spends 0.28μs on training one data to obtain the trained models and store the data in the leaves.

5 Related Work

The disaggregated memory systems. The promising disaggregated memory systems [27, 33, 34, 38, 42, 52] break a monolithic server into independent components to enhance the hardware scalability, which achieves high resources utilization by scaling out different hardware components [16, 49]. Different components communicate with each other via efficient RDMA techniques [4, 5, 19, 36]. Existing academic studies attempt to bring the disaggregated memory systems into practice via hardware designs [27,28]. Recently, Clio [17] proposes a hardware-software co-designed disaggregated memory system to equip each memory node with dedicated computing resources. LegoOS [34] proposes an OS model to manage disaggregated systems. Remote regions [1], LITE [40], and Semeru [42] are used to efficiently manage the remote memory resources. AIFM [32] designs a simple API for applications to use the remote memory. With the widely used NVM [29, 35, 48], Clover [39] remotely manages the persistent memory with low costs. FORD [50] enables the disaggregated memory systems to efficiently support transactions.

Table 1: The metadata analysis for ROLEX.

Number of Data	$5*10^6$	$1*10^7$	$5*10^7$	$1*10^8$	$5*10^8$
Number of Models	5,153	10,283	51,111	101,936	526,236
Size of Models (MB)	0.0798	0.157	0.779	1.555	8.03
Size of LT (MB)	4.768	9.537	47.683	95.367	476.837

Learned indexes for storage systems. The *learned indexes* [24] leverage calculations to predict positions for the given keys. Prior designs focus on various scenarios to enable the learned indexes to be widely used, including dynamically adapting to new data distributions [10, 14, 15], concurrent systems [37], LSM-based [9], and network-attached [44] KV stores. Motivated by the learned indexes, some studies leverage machine learning models to construct learned systems, e.g., DeepDB [18], Tsunami [11], and LISA [26].

Network-attached key-value stores. Due to the salient features of RDMA [4,33,36,49], constructing RDMA-enabled in-memory key-value stores [23,31,44,52] becomes efficient for distributed storage systems. Existing studies rely on two-sided RDMA verbs to process the data requests [6,21,23]. However, such server-centralized designs suffer from the CPU bottleneck when processing intensive requests [22, 44, 45] due to the poor computing capability of memory nodes. Unlike them, one-side RDMA enables compute nodes to directly access the remote data without involving remote CPUs [13, 39, 53]. For the ordered KV stores, Cell [31], FG [52], and Sherman [43] cache top-level nodes to reduce the number of RTTs based on B-link trees [25]. XStore [44] proposes a learned cache to further reduce the network penalty, which incurs one RTT to access the remote data. Unlike them, we design ROLEX for the disaggregated memory systems to efficiently process various requests via one-sided RDMA operations.

6 Conclusion

This paper proposes ROLEX, a scalable RDMA-oriented ordered key-value store using learned indexes for the disaggregated memory systems. ROLEX decouples the insertion and retraining operations, which enables the compute nodes to directly modify the remote data without retraining models. Other compute nodes identify the newly modified data via the stale models with consistency guarantees. ROLEX asynchronously retrains modes to improve the model accuracy. Our evaluation results demonstrate that ROLEX achieves high performance on both static and dynamic workloads in the context of the disaggregated memory systems. We have released the open-source codes for public use in GitHub.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022. We are grateful to our shepherd, Rong Chen, and anonymous reviewers for their comments and suggestions.

References

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (ATC)*, pages 775–787, 2018.

[2] Amazon. Amazon elastic block store. https://aws.amazon.com/ebs/?nc1=h_ls, 2021.

[3] Amazon. Amazon s3. https://aws.amazon.com/s3/, 2021.

[4] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, 2016.

[5] Amanda Carbonari and Ivan Beschasnikh. Tolerating faults in disaggregated datacenters. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets)*, pages 164–170, 2017.

[6] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable rdma rpc on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys)*, pages 19:1–19:14, 2019.

[7] Douglas Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[8] Intel Corporation. Intel rack scale design architecture. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html, 2021.

[9] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. From wisckey to bourbon: A learned index for log-structured merge trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 155–171, 2020.

[10] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. Alex: An updatable adaptive learned index. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD)*, pages 969–984, 2020.

[11] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *Proc. VLDB Endow.*, 14(2):74–86, 2020.

[12] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

[13] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 54–70, 2015.

[14] Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.*, 13(8):1162–1175, 2020.

[15] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, pages 1189–1206, 2019.

[16] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 249–264, 2016.

[17] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[18] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: Learn from data, not from queries! *Proc. VLDB Endow.*, 13(7):992–1005, 2020.

[19] HP. The machine. https://www.hpl.hp.com/research/systems-research/themachine/, 2021.

[20] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST)*, pages 187–200, 2018.

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *ACM SIGCOMM 2014 Conference (SIGCOMM)*, pages 295–306, 2014.

[22] 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)*, pages 185–201, 2016.

[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 1–16, 2019.

[24] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 489–504, 2018.

[25] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[26] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD)*, pages 2119–2133, 2020.

[27] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA)*, pages 267–278, 2009.

[28] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 189–200, 2012.

[29] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (ATC)*, pages 773–785, 2017.

[30] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012 (EuroSys)*, pages 183–196, 2012.

[31] 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 (ATC)*, pages 451–464, 2016.

[32] 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)*, pages 315–332, 2020.

[33] Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, and Tobias Ziegler. Rethinking distributed query execution on high-speed networks. *IEEE Data Eng. Bull.*, 40(1):27–37, 2017.

[34] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed os for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 69–87, 2018.

[35] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, pages 323–337, 2017.

[36] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki-Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 255–270, 2019.

[37] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. Xindex: a scalable learned index for multicore data storage. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 308–320, 2020.

[38] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *15th EuroSys Conference 2020 (EuroSys)*, pages 30:1–30:14, 2020.

[39] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (ATC)*, pages 33–48, 2020.

[40] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 306–324, 2017.

[41] Jérôme Vienne, Jitong Chen, Md. Wasi-ur-Rahman, Nusrat S. Islam, Hari Subramoni, and Dhabaleswar K. Panda. Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing

systems. In *IEEE 20th Annual Symposium on High-Performance Interconnects (HOTI)*, pages 48–55, 2012.

[42] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 261–280, 2020.

[43] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+ tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*, 2022.

[44] 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)*, pages 117–135, 2020.

[45] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 87–104, 2015.

[46] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. Maximum error-bounded piecewise linear representation for online stream approximation. *VLDB J.*, 23(6):915–937, 2014.

[47] Yahoo. Yahoo! cloud serving benchmark (ycsb). `https://github.com/brianfrankcooper/YCSB`, 2019.

[48] 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)*, pages 221–234, 2019.

[49] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transaction can scale. *Proc. VLDB Endow.*, 10(6):685–696, 2017.

[50] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. Ford: Fast one-sided rdma-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, 2022.

[51] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*, pages 523–536, 2015.

[52] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, pages 741–758, 2019.

[53] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided rdma-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (ATC)*, pages 15–29, 2021.

GL-Cache: Group-level learning for efficient and high-performance caching

Juncheng Yang
Carnegie Mellon University

Ziming Mao
Yale University

Yao Yue
Pelikan Foundation

K. V. Rashmi
Carnegie Mellon University

Abstract

Web applications rely heavily on software caches to achieve low-latency, high-throughput services. To adapt to changing workloads, three types of learned caches (learned evictions) have been designed in recent years: object-level learning, learning-from-distribution, and learning-from-simple-experts. However, we argue that the learning granularity in existing approaches is either too fine (object-level), incurring significant computation and storage overheads, or too coarse (workload or expert-level) to capture the differences between objects and leaves a considerable efficiency gap.

In this work, we propose a new approach for learning in caches ("group-level learning"), which clusters similar objects into groups and performs learning and eviction at the group level. Learning at the group level accumulates more signals for learning, leverages more features with adaptive weights, and amortizes overheads over objects, thereby achieving both high efficiency and high throughput.

We designed and implemented GL-Cache on an open-source production cache to demonstrate group-level learning. Evaluations on 118 production block I/O and CDN cache traces show that GL-Cache has a higher hit ratio and higher throughput than state-of-the-art designs. Compared to LRB (object-level learning), GL-Cache improves throughput by 228× and hit ratio by 7% on average across cache sizes. For 10% of the traces (P90), GL-Cache provides a 25% hit ratio increase from LRB. Compared to the best of all learned caches, GL-Cache achieves a 64% higher throughput, a 3% higher hit ratio on average, and a 13% hit ratio increase at the P90.

1 Introduction

Large-scale cache deployments enable the success of today's Internet. Companies have deployed software caches throughout various layers of the data center infrastructure: local and remote storage block I/O caches, in-memory and on-flash key-value caches. Caches are the key to fast data serving and consume a vast amount of resources. For example, Twitter reports that TBs of DRAMs are used for caching [104], and Netflix reports 10s of PBs of storage in use for caching [70].

The main driving force of cache deployments is the cache's ability to serve data with high throughput and low latency. Retrieving data from a cache (e.g., in DRAM) is thousands of times faster than retrieving it from the backend (e.g., in spinning disks). Because caches are often deployed on expensive storage media with limited capacity, the cache sizes are often much smaller than the dataset sizes. Thus, deciding what data to store in the cache is critical. A more *efficient* cache stores *more useful* data and serves more requests without hitting backend storage systems. Cache efficiency is often measured by hit ratio — the fraction of requests served from the cache (termed "hits"). When a cache is full, it uses an eviction algorithm to decide what data to keep and what to evict, and thus, the eviction algorithm is critical to cache efficiency.

Over the years, many eviction algorithms have been proposed to leverage different object features to make better eviction decisions. For example, several LRU variants [41–43,69,76,85] use diverse notions of recency to choose eviction candidates; some algorithms combine frequency and recency to score objects in different ways [4,15,26,28,56,92]; others use a composition of frequency and object size [17,20]. Since different features acquire varying degrees of importance for different workloads, using a specific way to combine one or two object features typically only achieves high efficiency on some workloads (§4.5). Recently, several works have employed machine learning to improve cache evictions. We call these designs "*learned caches*".

We classify learned caches into three categories. First, *"object-level learning"*, such as LRB [87], learns the next access time for each object using dozens of object features and evicts the object with the furthest predicted request time. Second, *"learning-from-distribution"* models request probability distributions to inform eviction decisions. For example, LHD [7] measures object hit density using age and size, and evicts the object with the lowest hit density. Third, *"learning-from-simple-experts"*, such as LeCaR [92] and Cacheus [82], performs evictions by choosing eviction candidates recommended by experts (e.g., LRU and LFU), and updates experts' weights based on their past performance on the workload.

Because object-level learning, such as LRB, leverages more

Table 1: Comparison of different learned caches (numbers describe the example systems).

Learning approach	Example system	Learning granularity	Features for eviction	Storage overhead (bytes per object)	Potential efficiency	Throughput relative to FIFO
Object-level learning	LRB [87]	object	44	189	high	0.001-0.01
Learning-from-simple-experts	Cacheus [82]	expert	2	32	low	0.2-0.25
Learning-from-distribution	LHD [7]	workload	2	24	medium	0.2-0.25
Group-level learning (this paper)	GL-Cache	object group	7	<1	high	0.3-0.8

object features, learns the relative feature importance, and performs fine-grained learning on each cached object, it has the highest potential for achieving high efficiency. However, predicting and ranking objects at each eviction incurs significant computation and storage overheads as we observe LRB suffers from a 775× slow down compared to LRU. Learning-from-distribution has a lower computation and storage overhead because it models request probability using fewer features at a coarser granularity. However, it still has a lower throughput compared to simple heuristics (e.g., LRU) because it has to randomly sample and compare many objects at *each eviction*. Moreover, the existing design (e.g., LHD [7]) does not leverage object features other than age and size, limiting its potential for high efficiency. Lastly, the performance of learning-from-simple-experts, which learns the weights of experts, highly depends on the choice of the experts. Existing systems use simple experts and cannot leverage features not considered by the experts (§4). We show the comparisons of the three types of learned caches in Table 1 and discuss each of these categories more in-depth in §2.2.

To overcome the challenges in the existing approaches to leverage learning in caching, we propose learning at the level of object groups (which we call group-level learning). Group-level learning leverages multiple group-level features to learn object-group utility for evictions. It reduces the computation and storage overheads of learning by hundreds of times through amortization compared to learning at the object level. Furthermore, object groups accumulate more "signals" for learning and can leverage a variety of features for prediction, enabling better eviction decisions.

While group-level learning seems promising, it introduces several challenges: (1) How to group objects and perform evictions efficiently? (2) How to measure the usefulness of object groups (termed "utility") to determine the best eviction candidate? (3) How to learn and predict the object-group utility online?

We present **G**roup-level **L**earned **Cache** (GL-Cache) which leverages group-level learning by overcoming these challenges. GL-Cache clusters similar objects into groups using write time (§3.3) and evicts the least useful groups using a merge-based eviction (§3.6). GL-Cache introduces a group utility function (§3.4) to rank groups, which enables group-based eviction to achieve similar efficiency as object-based eviction (§4.2). GL-Cache uses a hybrid approach for eviction: it performs the heavyweight learning at the group level (thus amortizing the overheads) to identify the best groups to evict. And it leverages lightweight object-level metrics to retain a few highly useful objects from evicted groups. This two-level eviction enables GL-Cache to achieve a superior trade-off between learning overhead and cache efficiency.

We implemented GL-Cache in an open-source production cache and also developed a storage-oblivious implementation for running microbenchmarks. We compare GL-Cache with state-of-the-art designs on 118 production block I/O and CDN cache traces. Compared to object-level learning (LRB), group-level learning allows GL-Cache to achieve a 228× higher throughput on average. Moreover, GL-Cache achieves a slight improvement in hit ratio compared to LRB, with a 7% increase on average and 25% at P90 (10% of the traces) compared to LRB. Compared to the learned cache with the highest hit ratio, GL-Cache increases the hit ratio by 3% on average and 13% at the P90 tail, with a 64% higher throughput. Varying group sizes allow GL-Cache to change learning granularity, leading to a spectrum of algorithms. Along with two other system parameters, this spectrum enables users to navigate the trade-off between efficiency and throughput.

This paper makes the following contributions.

- We classify existing learned caches into three categories based on learning granularity and propose a new approach for learning in caching — group-level learning. Group-level learning amortizes overheads over objects in the group to achieve high throughput. By leveraging multiple group features and accumulating more training signals, group-level learning also achieves a high hit ratio.
- We design and implement GL-Cache, which overcomes the challenges of using group-level learning to achieve high cache efficiency with low-overhead learning. For the first time (to the best of our knowledge), a group-level utility function is defined and used for cache eviction.
- We evaluate GL-Cache using a diverse set of 118 production traces to illustrate and understand the high efficiency and high throughput of group-level learning.

2 Background and motivation

2.1 Software caches in data centers

Applications rely heavily on caching to speed up data access and increase system throughput. The two most important metrics of cache are efficiency measured using *hit ratio* and performance measured using *throughput*. Hit ratio is the fraction of requests fulfilled by the cache without fetching from the backend, and it measures the effectiveness of an eviction algorithm. A cache is more *efficient* if it achieves a higher hit ratio. Throughput measures the volume of requests a cache can handle in a given duration. Higher throughput means serving the workload consumes less CPU resources and reduces expenses.

Over the years, many algorithms have been designed to improve cache hit ratio under different types of workloads [4,7,10,12,13,15,17,21,22,26,28,41–43,45,56,58,59, 68,69,76,79,82,85,87,92,98,103,109,110]. However, most of the algorithms make eviction decisions based on one or two object features, such as recency in LRU variants [43,76,85], and frequency in LFU variants [4,48], or a combination of two features [7, 15, 28, 92]. However, cache workloads are often too complex to be captured by one or two features, and different features may acquire different importance across workloads. Furthermore, the feature importance can be different when the same workload is served at different cache sizes, as we show in §4.5. As a simplified example, assume a workload is composed of Zipf and repeated scans. When the cache size is very small, frequency is more important in selecting popular objects from the Zipf distribution. However, when the cache size is large enough to store both popular objects and repeated scans, recency may become more important in choosing objects to cache. In addition, prior works [10,87] reveal a large hit ratio gap between the state-of-the-art designs and the upper bound (e.g., Belady's algorithm [8] or flow-based offline optimal [11]), illustrating the possibility of improving the cache efficiency further.

2.2 Learning in caching

To make cache eviction algorithms adaptive across workloads, cache size, and over time, recent works have explored the idea of using machine learning in caching [7,10,29,82, 87,93,102]. These approaches can be broadly classified into three classes, which come with their pros and cons, as discussed below and summarized in Table 1.

2.2.1 Object-level learning

Object-level learning performs learning on each object. Multiple works have studied the prediction of object reuse distance [10, 14, 32, 63, 65, 86, 87, 99, 100] and popularity [19,31,71,107]. By predicting reuse distance, a learned cache can mimic Belady's algorithm [8], which evicts the object requested the furthest in the future using an oracle. However, predicting reuse distance is challenging [87] because an object's reuse distance is not only inherent to the object but is also affected by the access patterns of the workload. For example, the reuse distance will increase if a request-burst or scan happens between the two requests to the same object. Moreover, cache workloads often follow Zipf distributions [5,9,18,104]. Thus, most objects only get a limited number of requests. This leads to *limited object-level information for learning*. Meanwhile, it is these less popular objects that often affect cache efficiency [102]. As a result, existing works introduce approximations and proxies for learning reuse distance. For example, LRB [87] introduces Belady Boundary to reduce the range of reuse distance. While learning reuse distance is challenging, with careful feature engineering, large enough data, and a complex model, object-level learning may have the potential to achieve the highest hit ratio among all learned caches. However, object-level learning incurs prohibitively high storage and computation overheads.

Storage overhead. Both training and inference require extra storage. While the storage overhead of training data is often negligible with optimizations such as sampling and offloading to cheaper storage, inference data pose a significantly higher storage overhead. To make predictions on the object level, the cache needs to track features for each object. For example, LRB [87] stores 44 features (189 bytes) per object. Moreover, this large per-object metadata overhead is prohibitively high because it needs to reside in DRAM for frequent updates. Using fewer features is possible, but it leads to worse performance (§4).

Computation overhead. Both training and inference add computation overhead. While training data collection and frequent re-trainings consume CPU cycles, inference is the major source of computation overhead. The prediction in object-level learning uses dynamic features (e.g., object age), and the prediction results cannot be reused over time. Therefore, object-level learning needs to sample objects and perform inference at each write (eviction). For example, LRB samples 32 objects and copies their features to a matrix for inference for *each* eviction. In our measurement, each eviction (including feature copy, inference, and ranking) takes 200 μs on one CPU core, indicating that the cache can evict at most 5,000 objects on a single core per second. As a comparison, a production server achieves over 100,000 requests per second [75].

2.2.2 Learning-from-simple-experts

Several works use reinforcement learning to choose between multiple simple experts (eviction algorithms). For example, LeCaR [93] uses two experts (LRU and LFU). At each eviction, LeCaR chooses one expert to make an eviction decision based on the experts' weights. Similar designs can be found in ACME [2], FRD [80], and Cacheus [82], which use different experts and weight adjustment methods.

By using more than one algorithm for eviction, learning-from-simple-experts can adapt to changing access patterns. The overhead and efficiency of learning-from-simple-experts depend on the experts. Existing systems use simple experts and thus incur lower overhead than object-level learning. However, existing systems suffer from two problems. First, a delay exists between a bad eviction and an update on the expert's weight. The cache only discovers a bad prior eviction when the evicted object is requested again. This challenge, commonly known as "delayed rewards" in reinforcement learning [3,36,47,90], limits the efficiency of caches that use learning-from-simple-experts. Second, the cache efficiency is bounded by the experts selected; an efficient policy requires a good understanding of the workload. Learning-from-simple-experts cannot leverage features that the experts do not consider. If a feature is important to the workload and not considered by any of the experts, then learning-from-simple-experts will not provide a high hit ratio. Some works used more experts [34] to capture more features. However, using more experts incurs higher overheads because it needs more computation and space to evaluate expert performance and update experts' weights.

2.2.3 Learning-from-distribution

The third type of learned cache models the request probability distribution and makes decisions based on the distribution. For example, LHD [7] uses the request probability distribution to calculate *hit density* (hits-per-space-consumed) as a metric for eviction. Specifically, LHD learns the request probability as a function of ages and then modulates it with size to arrive at hit density. LHD is simple yet effective and does not require expensive inference computation to compare objects. However, LHD's hit density is calculated based only on two features: age and size, and it is non-trivial to track probability with more features. Besides, LHD cannot change relative feature importance (how features are composed). Furthermore, because hit density does not change monotonically over time, LHD must sample objects to rank at each eviction, limiting its throughput due to slow random memory access.

Takeaways. We summarize the potential efficiency and overhead of the three types of learned caches in Table 1. We observe that object-level learning has a high potential to achieve high efficiency, but it incurs huge storage and computation overheads. Learning-from-distribution only considers a limited number of features and has lower overhead with lower potential for high efficiency. Although having a lower learning overhead, learning-from-distribution requires random sampling during *each* eviction, which limits its throughput. Learning-from-simple-experts highly depends on the experts used. Existing systems such as LeCaR and Cacheus achieve a higher hit ratio than a single expert but still leave a large hit ratio gap compared to other learned caches (§4.3).

3 GL-Cache: Group-level learned cache

To enable a better trade-off between learning granularity and learning overhead, we propose learning at the level of object groups (which we term "group-level learning"). The key idea behind group-level learning is to learn the usefulness of groups of objects (called "utility"). Based on this idea, we designed **G**roup-level **L**earned **C**ache (GL-Cache), which learns the object-group utility and evicts the least useful object groups. We first give a high-level overview of GL-Cache's design and then go into the details of each component.

3.1 Overview of GL-Cache

Fig. 1 shows an overview of GL-Cache. In GL-Cache, objects are clustered into fixed-size groups when writing to cache (§3.3). The training module in GL-Cache collects training data online and periodically trains a model to learn the utility of object groups (§3.5). The inference module predicts object-group utility and ranks object groups for eviction.

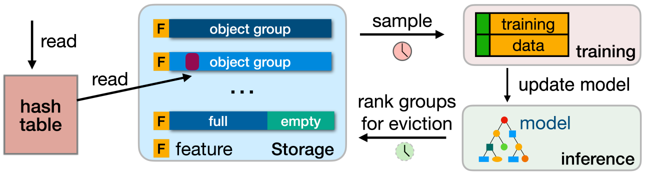


Fig. 1: Overview of GL-Cache. Objects are clustered into groups for learning: feature tracking, model training, and inference are performed on the group level.

Group-level learning requires group-level eviction: when the cache is full, object groups are evicted using a merge-based eviction which merges multiple groups into one, evicts most objects, and retains a small portion of popular objects (§3.6).

3.2 Group-level learning

Group-level learning has several advantages over existing learned caches:

Grouping amortizes overheads. Learning in caching incurs both computation and storage overheads. In group-level learning, these overheads are amortized over multiple objects in the group. In terms of storage, instead of adding huge per-object metadata, the metadata overhead is only added for each group. As a result, each object only incurs a tiny overhead on average (less than one byte in our implementation). The cost of inference computation is also amortized over objects. Compared to object-level learning, which performs one inference per eviction, each inference in group-level learning is used to evict a group of objects.

Grouping accumulates more signal. Many cache workloads follow a Zipf distribution [16, 104], and most of the objects receive very few requests. Because an object group has many objects, it often receives more requests than an individual object. More requests lead to more information on the group level compared to the object level, which makes it easier to learn and predict.

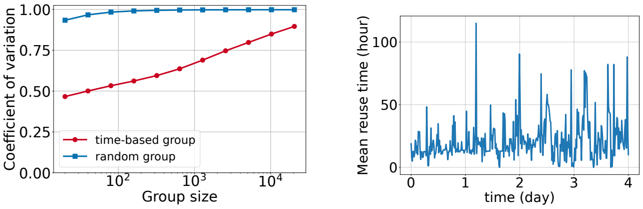
While group-level learning is promising, several challenges need to be addressed to leverage the power of learning:

- How to cluster objects into groups (§3.3)?
- How to compare the usefulness of object groups (§3.4)?
- How to learn the utility of object groups (§3.5)?
- How to perform evictions at group level (§3.6)?

While the ideas of grouping [105] and learning [87] have been studied independently in the context of caching, the combination of the two ideas in group-level learning leads to the unique challenges of *understanding, defining, and learning group utility*. We discuss these challenges and how GL-Cache overcomes them in this section.

3.3 Object groups

Using group-level learning, both learning and eviction are performed at the granularity of an object group, which usually contains tens to thousands of objects. Object grouping happens when an object enters the cache, and an object should not switch groups for two reasons. First, changing groups



(a) **(b)**

Fig. 2: a) Objects grouped using write time have more similar (smaller coefficient of variation) mean reuse time than objects grouped randomly. As group size increases, write-time-based grouping become closer to random grouping. b) Different object groups written at different times exhibit a large variation in mean reuse time.

invalidates the learning pipeline. When an object is added to or removed from a group, the accumulated group information becomes stale and cannot be used for learning. Second, in implementation, changing groups often requires copying data on the storage device. Therefore, the grouping of an object is decided when entering the cache using simple static object features. Depending on workload types, such features include time, tenant id, content type, object size, etc. In this work, we focus on grouping based on write time, which is available in all systems and hence more generalizable.

Similar to observations made in prior works [82, 105], we observe that objects written at a similar time exhibit similar behaviors. Using traces from the evaluation, we measure the mean reuse time variation of objects in (1) write-time-based groups and (2) random groups. Fig. 2a plots the mean coefficient of variation (standard deviation over mean) of 100,000 groups for the two grouping methods at different group sizes. Compared to random groups, write-time-based groups aggregate objects with closer mean reuse time. Besides reuse time, we have similar observations on the frequency and the group utility defined below (not shown due to the space limit).

While objects *within* each write-time-based group have similar reuse, object groups created at different times exhibit dramatically different mean reuse times. Using a group size of 100 objects on the same trace, Fig. 2b shows that some groups exhibit more than 10× higher mean reuse time than others. These high-reuse-time groups are potentially good candidates for eviction. The two observations illustrate the feasibility of group-level learning using write-time-based grouping: objects inside groups are similar. Grouping by write time also allows an efficient implementation using a log-structured cache.

3.4 Utility of object groups

Identifying a good eviction candidate in object-based eviction has been well-studied. When object size is uniform, Belady [8] algorithm evicts the object that is requested the furthest in the future. When object size is not uniform, identifying the optimal candidate is NP-hard [11]. A common approximation is to evict the object that has the largest time till the next request over object size (called "size-aware Belady"). However, no metric exists that applies to object groups, and it is not trivial to adapt object-level metrics to the group level. In this section, we define an *object-group utility* function to measure object-group usefulness. A group with a lower utility is less useful and hence should be preferred for eviction. Because identifying the optimal *object* for eviction (when objects do not have the same size) can be reduced to identifying the optimal *group* for eviction, and the former is NP-hard [11], finding the optimal group for eviction is also NP-hard. Therefore, we define an empirical group utility that satisfies several properties.

3.4.1 Desired properties

(1) Because large objects occupy more space, the utility should consider object sizes. Groups composed of larger objects should have lower utilities.

(2) Similar to Belady, the utility should consider the time till the next access of objects in the group. A group of objects that are requested further in the future should have a lower utility. Importantly, the utility definition should properly handle objects with no future requests.

(3) When the group size is one object, group-level learning becomes object-level learning. In this case, ranking using the defined utility should produce the same result as Belady.

(4) The utility should be easy and accurate to track online. Calculating the ground truth (used for training) requires future information, but the cache cannot wait indefinitely to calculate it. This property requires that within a limited time horizon, the online tracked utility should be close to the utility calculated with complete future information. In other words, objects requested further in the future, including the ones with no future requests, should contribute less to the utility.

3.4.2 Utility definition

We observe that the cost of evicting one object is *always only one miss*. After a cache miss, the evicted object will be inserted into the cache. Meanwhile, the benefit of evicting one object o is proportional to its size s_o and *time till next access* $T_o(t)$ from current time t. Therefore, similar to the cost-benefit analysis in LFS [83] and RAMCloud [77, 78], we define the utility of an object as its cost (one miss) over benefit (freed space multiplied by time till its next request).

$$U_o(t) = \frac{1}{T_o(t) \times s_o} \tag{1}$$

Because GL-Cache evicts object groups, we further define the group utility as the sum of object utilities.

$$U_{group}(t) = \sum_{o \in \text{group}} \frac{1}{T_o(t) \times s_o} \tag{2}$$

The utility of a group measures the penalty of evicting the group or the benefit of keeping the group. Groups with lower utilities are thus better candidates for eviction. We remark that this is one definition of group utility that both satisfies the desired properties and performs well in our experience (§4). With this definition, we compare object-group utility and evict the group with the lowest utility. Since the true utility relies

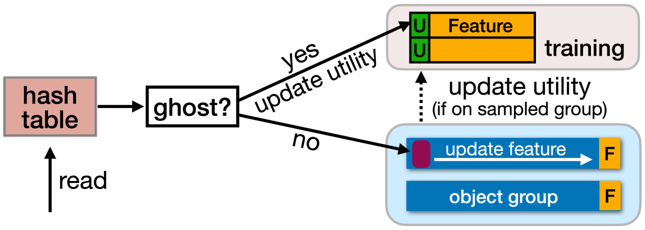


Fig. 3: The read flow in GL-Cache.

on the time till the next request and can only be calculated with future information, we design GL-Cache, which learns a model that can predict a group's utility based on its features.

3.5 Learning object-group utility in GL-Cache

GL-Cache learns a function $\mathcal{F}$ that calculates a group's utility given its features: $\mathcal{F}(X_{group}) = U_{group}$ where X_{group} is the features of an object group.

Object-group features. Features play a crucial role in learning [24, 38]. We consider two types of features in GL-Cache. The first type is *static features*, which includes request rate, write rate, miss ratio in the time window when the group was created (the write time of the first object), and mean object size. The second type is *dynamic features*, which includes age (in seconds), the number of requests, and the number of requested objects. Dynamic features increase over time. Static features do not change after creating a group and capture the workload and cache states (e.g., daily scan, request spike) during group creation time. We focus on these states because access pattern changes are often reflected in these metrics. For example, object groups created from scans are good candidates for evictions, and they often co-appear with increased request rates, write rates, and miss ratios. Compared to many of the existing works [87, 100], which mostly use dynamic features, GL-Cache uses far fewer dynamic features because tracking dynamic features is computationally expensive. We observe that adding more dynamic features only brings marginal hit ratio improvement, which does not justify the added computation overhead.

In total, GL-Cache uses seven features occupying 20 bytes for each group or 28 bytes if mean object size and creation time are not already tracked.

Learning model and objective function. GL-Cache uses gradient boosting machines (GBM) because tree models do not require feature normalization, and they have been shown to work well in previous works [10, 87] as well as many production environments [84, 96]. We formulate the learning task as a regression problem that minimizes the mean square loss (L2) of object-group utilities. We also explored the ranking objective function without observing a significant difference.

Training. GL-Cache trains a model using online collected training data, which consists of features and utilities of object groups. GL-Cache generates new training data by sampling cached object groups, and it copies the features of the sampled groups into a pre-allocated memory region. The utilities of the sampled groups are initialized to zero at the beginning and calculated over time. When an object o from a sampled group is requested, GL-Cache can calculate the $T_o(t)$ (time till next request since sampling) and object utility using Eq. 1 and add the object utility into the group utility. GL-Cache then marks the object to ensure that it only contributes to the group utility *once*. It is possible that some objects may not be requested before training, and the online calculated group utility may be lower than the true utility. However, as mentioned in §3.4, these objects contribute marginally to the group utility due to their large reuse time.

In addition, a sampled group may be evicted before being used for training. Such evictions halt the tracking of group utility. Inspired by prior works [69, 82], GL-Cache keeps ghost entries for objects which have not been factored into group utility. A future request on the ghost entry will update the group utility, bringing it closer to the true utility.

Fig. 3 shows the read flow in GL-Cache. A successful hash table lookup may find two types of entries: a pointer to the object or a ghost entry. If it is a regular object, GL-Cache first updates the group features. Further, if the object is on a sampled group and has not contributed to the group utility, GL-Cache also updates the group utility before returning the data to the user. If it is a ghost entry, GL-Cache updates the corresponding utility and removes the ghost entry from the hash table, then returns a cache miss.

Given the access patterns change over time, the model needs to be retrained regularly. GL-Cache retrains the model every day (i.e., using wall clock time as a reference) because many real-world events that trigger requests repeat on a daily basis, such as cron jobs. In contrast, the other option of retraining every certain number of requests may cause the system to enter metastable failure [40] when an access pattern change increases the system load. Besides, GL-Cache chooses to retrain from scratch each time because tree models do not benefit from continuous training. Moreover, the inference overhead grows with training iterations because a new tree is added to the model in each iteration.

Inference. When GL-Cache needs to perform evictions, it predicts the utilities of all object groups and ranks them. GL-Cache uses the inference/ranking result for multiple evictions, which reduces the frequency of inference and thus the computation overhead. We denote *eviction fraction* $F_{eviction}$ as the fraction of ranked groups to evict using one inference. That is, GL-Cache performs an inference every $F_{eviction} \times N_{group}$ groups where N_{group} is the total number of groups. In our evaluation, $N_{ranked-group}$ is the total number of groups, but we remark that one can also sample some groups for inference if the total number of groups is too large. Also, the groups are evicted over time on demand rather than all at once, and neither training nor inference need to be on the critical path of request serving. In summary, GL-Cache only needs to perform $\frac{1}{F_{eviction}}$ inferences to write a full cache of objects.

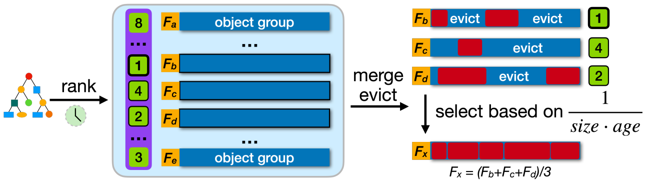


Fig. 4: Object group utility prediction and merge-based group eviction in GL-Cache.

3.6 Evictions of object groups

Learning at the object-group level introduces an interesting challenge to cache eviction: unlike most caches which evict one object each time, GL-Cache evicts a group of objects. Although evicting object groups leads to lower overhead due to batching and amortization, it may evict objects that are still popular. GL-Cache optimizes the group eviction by using a merge-based eviction, similar to Segcache [105]. Upon each eviction, GL-Cache picks the least useful object group and merges it with *the $N_{merge} - 1$ object groups that are closest with respect to write time*. The merge process retains S_{group} objects from the merged groups and evicts all other objects. The retained objects form a new group, and the original N_{merge} groups are evicted. This is the only time that an object changes its group membership in GL-Cache. Unlike group selection, which uses ranking, object selection uses a simple metric based on object age and size: $\frac{1}{size \cdot age}$ where *age* is the time since the last access. We choose to use this metric because recency and size are the two most common metrics used in other eviction algorithms (§2). GL-Cache performs the heavyweight online learning at the group level to identify the best groups to evict. It leverages lightweight object-level metrics to retain a few highly useful objects. This two-level eviction approach enables GL-Cache to achieve a superior tradeoff between learning overhead and cache efficiency.

In summary, each eviction evicts N_{merge} groups of objects and retains one group of objects, as illustrated in Fig. 4. The features (except mean object size) of the merge-produced group take the mean values of the N_{merge} merged groups. Note that only the first object group is picked based on the group utility; the next $N_{merge} - 1$ object groups are chosen as ones with write time close to the first group. This ensures that objects in the new group after a merge-based eviction are still close in write time and similar. In contrast, objects from the N_{merge} least useful groups may not be similar. Clustering similar objects into groups is critical for effective group-level learning. In our experience, merging the N_{merge} least useful groups shows lower efficiency with up to 20% decrease in hit ratio. Compared to evicting one object each time, group-based eviction evicts more objects than needed at each eviction, which may reduce the efficiency upper bound group-level learning can achieve. However, we show in §4.2 that evicting object groups can achieve hit ratios very close to Belady, indicating that group eviction will not be the bottleneck for cache efficiency.

Table 2: Parameters used in the design.

Para	Meaning
S_{group}	Size of an object group (in number of objects or bytes)
N_{merge}	Number of object groups to merge each eviction
$F_{eviction}$	Each inference evicts $F_{eviction}$ fraction of ranked groups

Table 3: Three sets of 128 traces were used in the evaluation.

Dataset	# traces	# requests (millions)	Source
CloudPhysics [94]	103	2115	VM disk I/O
MSR [73]	14	410	Disk I/O
Wikimedia [87]	1	2804	CDN requests

3.7 A spectrum of GL-Cache

GL-Cache has three parameters in its design (Table 2): the size of each object group S_{group}, the number of object groups to merge at each eviction N_{merge}, and how many groups are evicted using one inference which is determined by $F_{eviction}$. Varying these parameters leads to a spectrum of algorithms for optimizing hit ratio and throughput. A larger S_{group} reduces learning granularity; a larger N_{merge} retains fewer objects; and a larger $F_{eviction}$ reduces the ranking frequency. Each of these changes reduces the computation overhead with a potential hit ratio drop. Therefore, GL-Cache allows the users to navigate the trade-off between cache efficiency and throughput. For scenarios that are more sensitive to overheads, such as local cache deployments, GL-Cache can provide higher throughput with a slightly lower hit ratio, and vice versa. In §4.6, we show that these parameters generalize well across workloads.

4 Evaluation

In this section, we evaluate GL-Cache to answer the following questions.

- Will group-based eviction limit the efficiency upper bound when compared to object-based eviction (§4.2)?
- Can GL-Cache improve hit ratio and efficiency over other learned caches (§4.3)?
- Can GL-Cache meet production-level throughput requirements and how much overhead does GL-Cache add (§4.4)?
- How does GL-Cache improve efficiency without compromising throughput (§4.5)?

4.1 Experiment methodology

Prototype system. GL-Cache groups objects using write time and can be efficiently implemented using a log-structured cache. Hence, we implement GL-Cache on top of Segcache [105], an open-source production in-memory cache that uses segment-structured (log-structured) storage. We map an object group in GL-Cache to a "segment" in Segcache and replace FIFO with the learned model. We use the XGBoost [1] library to implement our GBM models and use the default values for all parameters. GL-Cache has three parameters (Table 2). In our evaluation, GL-Cache uses 1 MB group size, merges five groups at each eviction, and evicts 5% of ranked groups after each inference. We compare GL-Cache with Segcache [105], a segment-structured cache used by Twitter; Cachelib [9], Meta's production cache library, which uses slab storage and a throughput-optimized LRU for eviction; TinyLFU [28], implemented within Cachelib by Meta engineers. We have also implemented LHD [7] on top of Pelikan's slab storage [81].

Micro-implementation. In addition to the prototype system, we build a storage-oblivious implementation of GL-Cache in C on top of libCacheSim [101] to compare different eviction algorithms. Our implementation mimics Memcached's design but has neither a networking stack nor object value storage, and we call it micro-implementation. Compared to the prototype, the micro-implementation only performs eviction-related metadata operations and does not consider storage layout or system overheads such as fragmentation. We use two sets of parameters (Table 2) to demonstrate the spectrum of GL-Cache. The first demonstrates a better *efficiency* and uses $S_{group} = 60$ objects, $N_{merge} = 2$ groups, $F_{eviction} = 0.02$. We call this system GL-Cache-E. The second demonstrates a higher *throughput* using $S_{group} = 200$ objects, $N_{merge} = 5$ groups and $F_{eviction} = 0.1$, and we call it *GL-Cache-T*. We remark that the parameters are not tuned per workload. Thus GL-Cache may provide better performance (hit ratio or throughput) with workload-specific fine-tuning.

Besides GL-Cache, we implement Cacheus [82] in C following the authors' open-source Python implementation. For LHD [7] and LRB [87], our micro-implementation used code open-sourced by the authors. We use default parameters except for changing the LRB optimization target from byte miss ratio to object miss ratio (implemented by LRB's author). Besides state-of-the-art designs, we have also implemented FIFO, LRU, and size-aware Belady [11].

GL-Cache trains the first model after running one day of workload (using timestamps from the traces). Before a model is trained, it uses FIFO to perform evictions, GL-Cache then trains the model once a day from scratch, which has little overhead as discussed in §3.5.

Workloads. We use a wide variety of traces representing a diverse set of workloads from three dataset sources (Table 3). The CloudPhysics [94] dataset includes 103 block I/O traces with different CPU/DRAM configurations and access patterns. Each trace records the I/O requests from a VM for around one week. Because 86% of the VMs had DRAM sizes between 1 GB and 16 GB with a median of 3880 MB, we performed evaluations at 1 GB, 4GB, and 16 GB cache sizes. We present only 1 GB and 16 GB for space reasons. We have also evaluated GL-Cache using 14 block I/O traces (we ignore the traces which contain fewer than 5 million requests) from Microsoft Research Cambridge (MSR) [73]. Because the working set sizes of MSR traces exhibit a very wide range, we set cache sizes for each trace at 0.01%, 0.1%, and 1% of each trace's footprint (size of all objects). Besides block I/O request traces, we have also evaluated GL-Cache with the Wikimedia CDN trace used in previous works such as LRB [87] and LFO [10]. All the workload traces have at least three fields: the timestamp, id, and size of the requests.

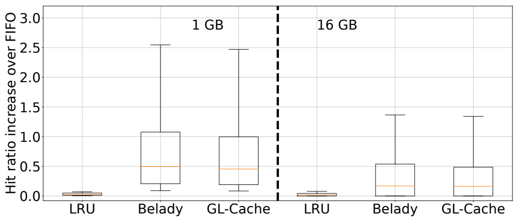


Fig. 5: With oracle assistance, group eviction can achieve a similar hit ratio improvement as object eviction.

We ran micro-implementation experiments on the Cloudlab [25] Utah site using m510 nodes with Intel Xeon D-1548 CPU, 64GB ECC DDR4 DRAM. And we ran prototype experiments on the Cloudlab Clemson site using c6420 nodes with Intel Xeon Gold 6142 CPU and 384 GB of DRAM.

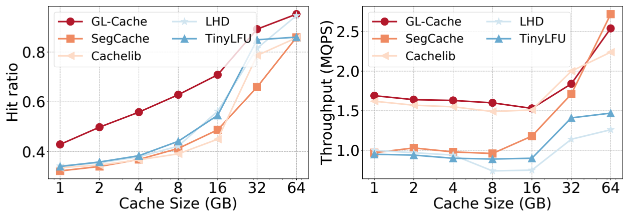
Metrics. We replayed traces by reading and writing to a local cache in a closed loop and measured hit ratio and throughput. Because all traces are week-long traces, we started measurements after finishing the first three days' requests to make sure the cache is properly warmed up under all the configurations considered. We present evaluations using a one-day warmup time in §4.6, which shows that the observations remain the same as with a three-day warmup.

We report aggregated results from 103 CloudPhysics traces and 14 MSR traces using box plots for the micro-implementation results. Due to the diversity of the workloads, both hit ratio and throughput have wide ranges. Hence, for ease of visual presentation, we report results compared to FIFO using the following two metrics: *hit ratio increase over FIFO* defined as $\frac{HR_{alg} - HR_{FIFO}}{HR_{FIFO}}$ where HR stands for hit ratio; throughput relative to FIFO defined as $\frac{R_{alg}}{R_{FIFO}}$ where R is the throughput. The box plots have the following format: the orange line inside the box is the median, the box shows 25 and 75 percentiles, and the whiskers show 10 and 90 percentiles. Because several other factors in the prototype systems (e.g., storage layout) affect efficiency and throughput, for ease of understanding, we focus our evaluation on the micro-implementation results. We present raw hit ratio and throughput numbers using the prototype systems for one representative trace in §4.3 and §4.4.

4.2 Group-based eviction

Group-level learning evicts most objects in the selected groups. The bulk eviction may limit the efficiency of group-level learning. To understand the limitation of group eviction, we compare oracle-assisted group eviction with oracle-assisted object eviction (size-aware Belady [11]). The oracle-assisted group eviction uses the same design as GL-Cache except using future request time to calculate group utility and retain objects. Size-aware Belady evicts the object that has the largest $(T_{next} - T_{now}) \times s_o$ where T_{next} is the time of the next request, and s_o is the object size.

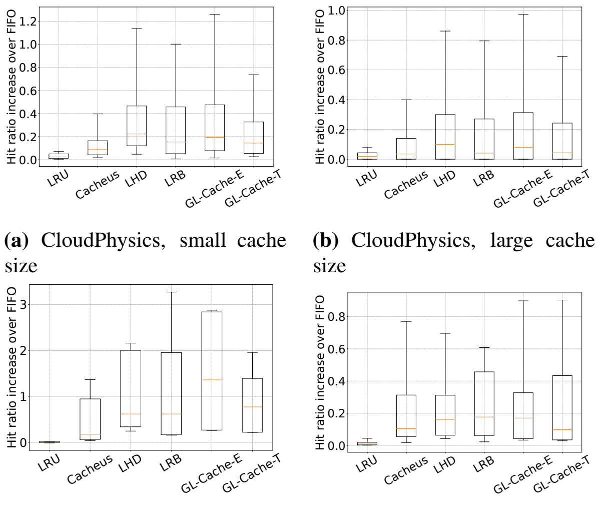
We compare these two approaches using CloudPhysics



(a) Hit ratio

(b) Throughput

Fig. 6: Prototype evaluation of a CloudPhysics trace.



(a) CloudPhysics, small cache size

(b) CloudPhysics, large cache size

(c) MSR, small cache size

(d) MSR, large cache size

Fig. 7: Hit ratio increase over FIFO. GL-Cache runs under two modes, GL-Cache-E is the efficient mode, GL-Cache-T is the throughput mode.

traces. Fig. 5 shows that group-based eviction can achieve a hit ratio similar to object-based eviction at both small and large cache sizes. The similar hit ratios suggest that *group eviction will not become the bottleneck for achieving high efficiency*. While the algorithms in this comparison use oracle information, in the following sections, we show how GL-Cache can use learning to replace the oracle and achieve high cache efficiency.

4.3 Cache efficiency

We compare the efficiency of GL-Cache with state-of-the-art designs in both the prototype and the micro-implementation. Fig. 6a shows hit ratios for the prototype running one CloudPhysics trace at different sizes. Compared to other systems, GL-Cache consistently achieves the best efficiency, providing a significant hit ratio increase (up to 40%) over the best of all baselines. Compared to Segcache, which uses the same storage layout with FIFO-based group eviction, group-level learning increases the hit ratio by 60% at 8 GB. Cachelib uses a throughput-optimized LRU and has the lowest hit ratio among all the baselines. LHD and TinyLFU use two object features to make eviction decisions: LHD models hit density based on age and size; TinyLFU uses frequency to filter out unpopular objects and uses recency to evict objects. Leveraging more than one feature to choose eviction candidates allows LHD and TinyLFU to achieve higher hit ratios. However, not using more features puts an upper bound on their potential. In comparison, GL-Cache evicts groups based on seven features covering recency, frequency, cache, and workload states at group creation time (miss ratio, write rate, request rate). Considering multiple features in conjunction with learned importance allows GL-Cache to make better eviction decisions and achieves a higher hit ratio. Evaluations on the other traces show similar results.

To compare with more algorithms and on more traces, we show hit ratio results from the micro-implementation on CloudPhysics and MSR traces in Fig. 7. Because of the wide range of hit ratios across traces, we show the relative hit ratio increase compared to FIFO instead of the raw hit ratios. We observe that both LRU and Cacheus improve FIFO's hit ratio, but only by a single-digit percentage for the median workload on both datasets. Meanwhile, LRB, LHD, and GL-Cache increase FIFO's hit ratio more prominently.

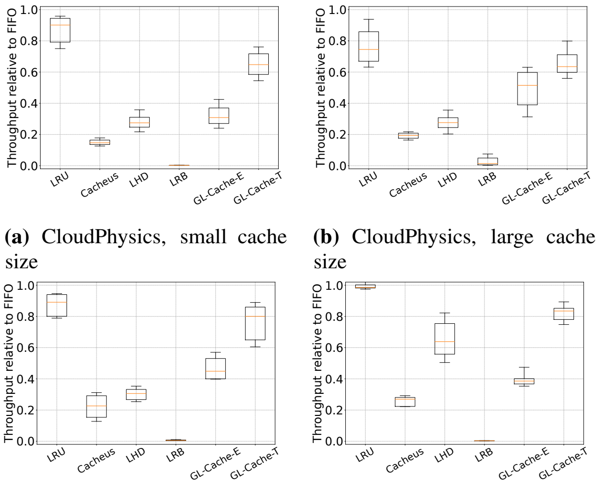
Among LRB, LHD, and GL-Cache-E, LRB has the smallest observed hit ratio improvement. We conjecture that learning at the object level receives limited information on each object since cache workloads often follow Zipf distributions, and thus is more challenging to learn compared to learning at the group level. Compared to LHD, we observe that GL-Cache-E shows similar efficiency on CloudPhysics traces. However, on MSR traces, GL-Cache-E is more efficient than LHD with a 60% hit ratio increase for a median workload at the small size. This observation suggests that leveraging more features to make eviction decisions can be very useful for some workloads at certain cache configurations.

Compared to GL-Cache-E, GL-Cache-T trades hit ratio for higher throughput (§4.4). However, we observe that GL-Cache-T's efficiency is still on-par with LRB. Overall, we observe that GL-Cache improves the hit ratio by up to 37.8% compared to LHD and 87% compared to LRB (not shown in the figure). While LRB uses more features/information than other eviction algorithms, it does not always provide the highest hit ratio. More information leads to higher efficiency only when the information is useful and well-utilized. We conjecture that perhaps not all the features in LRB are useful, and the model may not be making the best use of the features.

When comparing prototype and micro-implementation results, we observe that the hit ratio difference also depends on the storage design. GL-Cache uses log-structured storage, and the difference between prototype and micro-implementation is smaller (<10%); LHD uses slab storage, and sometimes the prototype can have a significantly lower hit ratio (>20%) compared to the micro-implementation. This large difference comes from fragmentation and slab calcification problems [39, 105]. However, we did not find a way to efficiently implement LHD on top of log-structured storage because it requires the storage to have the capability of evicting (removing) any cached object, while log-structured storage can only

Table 4: Comparing LRB and GL-Cache-E on the Wikimedia trace used in LRB paper [87]. We use miss ratio because it is more commonly used in web caches.

Algorithm	Miss ratio			Throughput (MQPS)		
Size (GB)	20	200	2000	20	200	2000
FIFO	0.39	0.16	0.025	7.62	7.91	9.68
LRB	0.24	0.048	0.016	0.01	0.04	0.07
GL-Cache-T	0.24	0.065	0.017	4.97	6.53	4.89
GL-Cache-E	**0.20**	**0.041**	**0.013**	2.55	3.91	4.20



(a) CloudPhysics, small cache size

(b) CloudPhysics, large cache size

(c) MSR, small cache size

(d) MSR, large cache size

Fig. 8: Throughput relative to FIFO.

efficiently support sequential write and removal.

Besides block I/O cache traces, we have also evaluated GL-Cache using the Wikimedia CDN trace from LRB evaluations. Table 4 shows that learning helps LRB to achieve miss ratios 35% to 70% lower than FIFO. Compared to LRB, GL-Cache-E further reduces the miss ratio by up to 16%. In summary, the evaluations on three datasets totaled 118 traces illustrating the high efficiency and generality of group-level learning.

4.4 Throughput and overheads

Not only does GL-Cache achieve a high hit ratio, but GL-Cache also achieves high throughput. Fig. 6b shows the throughput of GL-Cache in the prototype. We observe that compared to production systems (Cachelib, Segcache), GL-Cache achieves a similar throughput, indicating that GL-Cache meets the throughput requirement of a production system. Moreover, compared with eviction algorithms such as LHD and TinyLFU, GL-Cache is 2-3× faster.

Besides the prototype evaluation, Fig. 8 compares the throughput of GL-Cache with several state-of-the-art algorithms evaluated on *all* CloudPhysics and MSR traces. While LRU achieves throughput close to FIFO, all advanced eviction algorithms exhibit a significant slowdown compared to FIFO. However, among all learned caches, GL-Cache is significantly faster than others. Compared to LRB, GL-Cache-E has a 228× higher throughput, and GL-Cache-T has a 586× higher throughput on average at the small cache size. Compared to the *fastest* of all learned caches, GL-Cache-E is on average 64% faster, and GL-Cache-T is on average 3× faster at the small cache size. Similarly, on the Wikimedia trace (Table 4), GL-Cache-E is tens to hundreds of times faster than LRB and achieves almost half of FIFO's throughput.

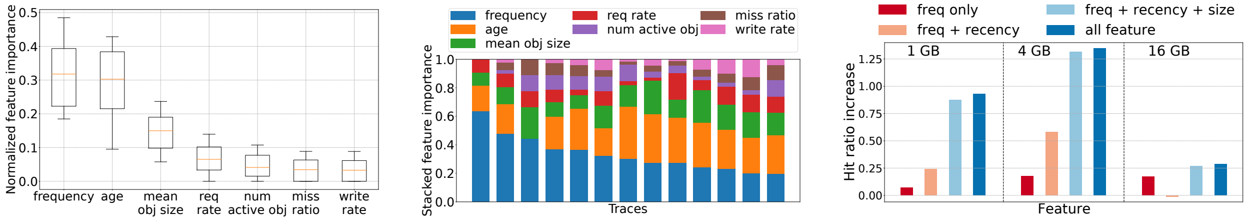
GL-Cache achieves high throughput because it needs very few metadata updates on cache hits and misses. On a cache hit, GL-Cache only needs to update the last access time and group utility if it is on a sampled group (§3.5). On a cache miss, GL-Cache does not need to update any metadata most of the time; occasionally, it performs a group eviction and evicts 100s to 1000s of objects. In contrast, other systems must update multiple metadata entries on both cache hits and cache misses. For example, TinyLFU needs to maintain the frequency counting sketch and the LRU chain; LHD needs to sample 32 objects, thus having 32 random DRAM accesses for each eviction. Segcache is simpler than GL-Cache in per-request operations. However, the lower hit ratio of Segcache leads to its reduced throughput because of more evictions.

The second reason for GL-Cache's high throughput is that the overheads of training and inference are amortized. Because GL-Cache uses fewer features to learn simpler high-level patterns instead of per-object access patterns, it uses a simple model and is only retrained once a day. In our measurement, each training consumes 10 - 50 ms of one CPU core (not amortized by the number of training samples). In addition, each inference consumes 0.4 - 3 ms of one CPU core and is triggered every time 5% of ranked groups are evicted. Because each inference evicts many groups and each eviction evicts many objects, the inference computation is amortized. The amortization is the key reason for GL-Cache's high throughput compared to other learned caches. Moreover, although training and inference are not on the critical path of request serving, our throughput evaluation measures run time including both training and inference.

While throughput evaluations show the low computation overhead of GL-Cache, machine learning in caching also introduces storage overhead. First, GL-Cache uses DRAM to store 8000 training samples. The training data storage is pre-allocated and small (256 KB) compared to the cache size (GBs). For deployments with very limited memory, the training data can also be stored on the storage device. Second, each object group in GL-Cache uses 28 bytes of features — each object thus adds less than one byte. Besides the group-level features, GL-Cache tracks each object's last access time using 4 bytes. In total, GL-Cache uses 5 bytes of object metadata for eviction. As a comparison, LRU requires two pointers with 16 bytes of metadata per object, and LRB uses 192 bytes of features per object.

4.5 Understanding GL-Cache's efficiency

So far we have demonstrated that GL-Cache has a higher miss ratio and throughput than existing systems. While amortized overhead explains the high throughput, this section ex-



(a) Feature importance across traces

(b) Feature breakdown across example traces.

(c) Feature utility case study.

Fig. 9: Feature case study.

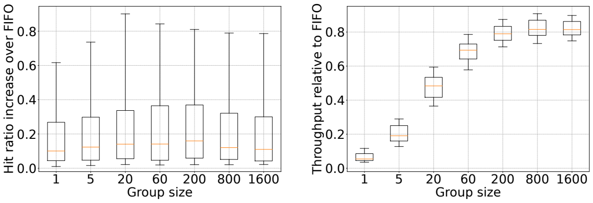
plores how learning helps GL-Cache achieve high efficiency.

Most eviction algorithms use one or two object features to decide which object to evict. For example, LRU evicts the object with the largest access age (recency), LeCaR and Hyperbolic [15] use recency and frequency to make eviction decisions, LHD relies on access age and object size to choose eviction candidates. In contrast, object-level learned cache such as LRB uses 44 features covering different measurements of recency and frequency, as well as object size, to compare objects. Similarly, GL-Cache uses seven features to compare object groups. To better understand GL-Cache's efficiency, we examine how GL-Cache uses these features.

We obtained the feature importance score directly from XGBoost. The importance score is calculated using the number of times a feature is used to split the data across all trees and may not represent the ground truth. Fig. 9a shows the normalized feature importance scores of different features across traces obtained from the models trained for each trace. We observe that across traces, frequency and age have relatively high scores with medians of arround 0.3. This aligns well with existing literature on eviction algorithms, which mostly use recency and frequency to make eviction decisions. The next important feature is the mean object size, which is essential for algorithms that consider variable-size objects. Besides these features, the workload and cache states (request rate, miss ratio, write rate) at the group creation time have similar scores with a median of around 0.05. When summed up, they have a similar importance as the object size.

While we observe that the most commonly used features (recency, frequency, size) are critical, we also observe that no feature is dominant across all traces. Fig. 9b shows the feature importance score for 12 randomly selected traces. For some traces, frequency is more important, with an importance score of 0.6. For others, recency or size is more important. GL-Cache weighing features differently across traces suggests that GL-Cache can effectively adapt the feature importance to each workload. For comparison, the algorithms leveraging more than one feature often combine the features in a way that cannot adapt to workloads. For example, Hyperbolic scores an object using $\frac{frequency}{age}$, leaving the *relative* importance of frequency and age unchanged across workloads.

Fig. 9c uses one trace to illustrate the importance of GL-Cache *adaptively* using multiple features. It shows how gradually including more features improves the hit ratio. We observe that the combination of frequency, recency, and size at small sizes (1 GB and 4 GB) leads to a large hit ratio increase (e.g., 80% at 1 GB). Meanwhile, frequency alone is insufficient and can only increase the hit ratio by 10% at 1 GB. Using all features increases the hit ratio modestly on this trace compared to only using frequency, age, and size. Moreover, Fig. 9c shows that feature importance could change with cache sizes. Object size is more important at 1 GB cache size, while frequency becomes more important than other features at 16 GB. This could be because small objects contribute more hits per consumed byte than large objects, so caching small objects is better when the cache size is small. Meanwhile, when most small objects are cached at a larger cache size, choosing between large objects depends on request frequency. This observation suggests that in GL-Cache, the choice and use of features adapt not only to the workloads but also to different configurations such as cache sizes.



(a) Hit ratio

(b) Throughput

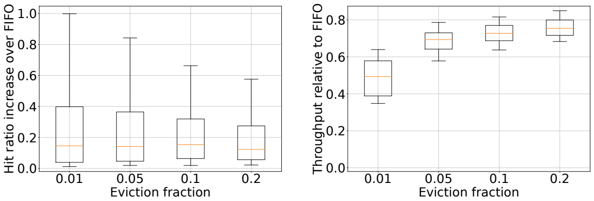
Fig. 10: Impact of group size.

In summary, learning at the group level can leverage multiple features to adapt to both workload and cache sizes, enabling higher cache efficiency.

4.6 Sensitivity analysis

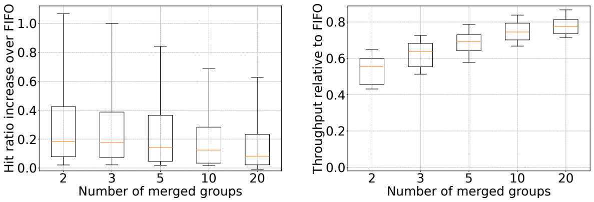
We have discussed the three parameters used by GL-Cache in §3.7, and we have shown the two modes of GL-Cache: one achieves higher efficiency (GL-Cache-E), and the other achieves higher throughput (GL-Cache-T). This section shows in detail how these parameters affect hit ratio and throughput. In addition, we show that the warmup time does not significantly change the hit ratios.

Group size. A smaller group indicates a finer granularity for learning and evictions. Varying group size affects both throughput and efficiency. First, reducing group size increases storage and computation overhead due to finer learning granularity. As a result, throughput increases with group size, as shown in Fig. 10. Second, the hit ratio increases when



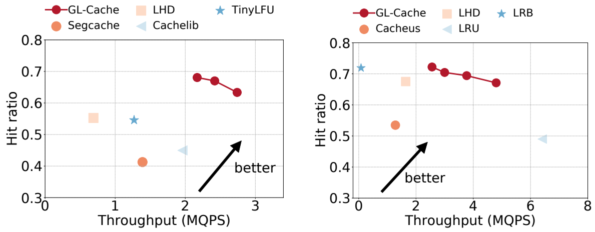
(a) Hit ratio **(b)** Throughput

Fig. 11: Impact of eviction fraction $F_{eviction}$.



(a) Hit ratio **(b)** Throughput

Fig. 12: Impact of the number of groups to merge at each eviction.



(a) Prototype evaluation **(b)** Micro-implementation

Fig. 13: A spectrum of GL-Caches allow users to tradeoff between hit ratio and throughput.

the group size increases from 1 (object-level learning) to 20, then decreases as the group size further increases from 60 to 1600. A smaller group indicates that each eviction evicts fewer objects, enabling a higher hit ratio. However, when the group size is too small, each group gets too few requests for group feature learning to be effective, thus decreasing the hit ratio. The non-monotonic hit ratio change (hit ratio first increases then decreases) also explains why object-level learning achieves a lower hit ratio than GL-Cache.

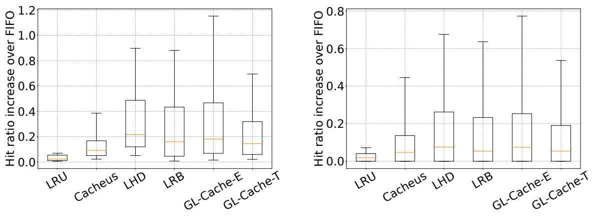
Eviction Fraction. GL-Cache evicts $F_{eviction}$ fraction of ranked groups between each inference to reduce computation overhead and better tolerate inaccurate predictions. The more groups (larger $F_{eviction}$) evicted per inference, the fewer inferences, thus higher throughput. However, a larger $F_{eviction}$ means more (useful) groups are evicted after each inference, resulting in a lower hit ratio. Fig. 11 shows that increasing $F_{eviction}$ reduces hit ratio and increases throughput.

Number of groups to merge. The last tunable parameter in GL-Cache is the number of groups to merge at each eviction. Because GL-Cache evicts the majority of the objects on the N_{merge} groups and retains one group worth of objects, merging more groups means that GL-Cache retains fewer objects from each group. Retaining fewer objects reduces the computation needed at each eviction, but it also reduces efficiency. Fig. 12 shows that increasing the number of merged groups increases throughput and reduces the hit ratio.

Besides the above three parameters, the learning component also introduces several parameters such as training data size and retraining frequency. GL-Cache retrains the model once a day because many events (such as cron jobs and diurnal patterns) happen on a daily basis. Wall clock time sometimes is more important than virtual time (reference count), and has also been recognized by researchers from Google when they use neural networks to predict the lifetime of a memory allocation [64]. The retraining interval affects both efficiency and performance. Note that more frequent retraining does not always lead to a higher hit ratio because shorter retraining intervals reduce the accuracy of the group utilities used for training as they are accumulated over time. We observe that the best retraining interval depends on the workload — some workloads show higher hit ratios with half-day retraining, and some others benefit from two-day retraining. While fine-tuning retraining intervals can improve the hit ratio by up to 10%, one-day retraining achieves a good performance across workloads as shown. Besides training frequency, another parameter in training is the number of training samples. Because GL-Cache learns high-level access patterns, which we conjecture is easier to learn than per-object behavior, GL-Cache does not require a large amount of training data. While we cannot prove that 8000 training samples are sufficient for all workloads under all scenarios, we find that it is sufficient for the diverse traces in our evaluation.

The sensitivity analysis shows that GL-Cache is relatively robust to parameter changes. The parameters of GL-Cache-E and GL-Cache-T were chosen based on evaluations of 10 random traces. Our results show that these two sets of parameters work well across the diverse traces in the evaluation. However, like in any other system, a general set of parameters provides reasonable performance but does not guarantee the best performance. Per-workload fine-tuning can potentially provide larger benefits. GL-Cache provides the opportunity for users to explore the trade-off between efficiency and throughput. Fig. 13 shows the throughput and hit ratio of GL-Cache compared to baselines (we do not plot multiple close-by points of GL-Cache for clarity). In both prototype and micro-implementation evaluations, GL-Cache achieves higher throughput than systems with a similar hit ratio or a higher hit ratio than systems with a similar throughput. Deployments with less computation power can use GL-Cache in a high-throughput mode with a slightly lower hit ratio. And deployments that are less sensitive to computation may use GL-Cache to achieve a higher hit ratio.

Our evaluation so far used a warmup time of three days to make sure the cache is warmed up for any trace under any size. We have also evaluated with a one-day warmup time and presented the results in Fig. 14. We observe that although the absolute values exhibit some differences, the overall trends on hit ratio increase are similar when compared to using a three-day warmup time (Fig. 7). In addition to the hit ratio



(a) Small cache size **(b)** Large cache size

Fig. 14: Using one-day warmup, evaluated on CloudPhysics traces.

results, throughput results using a one-day warmup are also similar to that of a three-day warmup. Similarly, evaluations on the MSR and Wikimedia traces also exhibit little difference between using one-day and three-day warmup times.

5 Related work

The study of cache designs has a long history with the majority of works focusing on improving cache efficiency. With increasing complexity in cache management, many recent works have also improved the throughput and scalability.

Better eviction algorithms. Most works improving cache efficiency focus on cache eviction algorithms, especially how to define and use recency, frequency, and size to make better eviction decisions. For example, ARC [69] uses two LRU lists to balance between recency and frequency; CAR [6], LIRS [43, 44, 57, 108], Clock-pro [42], 2Q [85], SLRU [41], LRU-K [76] use a different metric to measure recency; variants of LFU [4, 48], LRFU [56], tinyLFU [26–28] and hyperbolic [15] use a combination of frequency and recency to make evictions; various greedy-dual algorithms [17,20,45,59] use two metrics (e.g., frequency and size) to choose eviction candidates. In addition, several learned caches have been designed in the past few years, as discussed in detail in §2.2. Compared to existing learned caches, GL-Cache employs group-level learning, which amortizes overheads and accumulates stronger learning signals to make better eviction decisions. Moreover, existing learning approaches to caching cannot be directly applied to group-level learning due to challenges such as comparing object groups' usefulness.

Improve cache throughput. Most algorithms that improve efficiency trade throughput for higher efficiency. With increasing complexity in cache systems, throughput and scalability become critical. MICA [62] uses a holistic design with a lossy hash table and partitioned log-structured DRAM storage to achieve high throughput and scalability; Segcache [105] uses an approximate-TTL-indexed segment-chain with batched eviction to achieve high throughput and scalability; MemC3 [30] uses a cuckoo hash table and Clock eviction to improve scalability; Cachelib [9] reduces LRU promotion frequency to improve scalability. These systems often use weaker eviction algorithms such as FIFO, Clock, or weak LRU. Compared to these works, GL-Cache improves efficiency without sacrificing throughput. Specifically, GL-Cache and Segcache share some design aspects such as object grouping. However, Segcache primarily innovates on the design of storage layout for key-value caches, and it uses FIFO for eviction. Instead, GL-Cache focuses on using learning for evictions, which is the key to GL-Cache's efficiency gains.

Use of machine learning to improve system efficiency. Machine learning has seen increasing use to improve system efficiency. For example, Google uses machine learning to improve the efficiency of data center operations [33]. Microsoft uses machine learning to improve database query optimizer [46]. Prior works have designed learned components to replace various parts of a system, such as index [23, 50, 54, 55, 74, 97] and query optimizer [66, 67] in databases, straggler mitigation in inference systems [52, 53], and FTL for SSD [89]. Moreover, many other works look into automatic database tuning using machine learning [60, 91]. In caching, in addition to the three categories of learned cache evictions that we have discussed in §2, recent works have also looked into using sub-sampling to reduce learned cache's time horizon [95], using machine learning to predict memory access [37], designing cache admission [35, 51], designing cache prefetching [61, 61, 88, 102] predicting hot records in LSM-Tree storage [106], using deep recurrent neural network models for content caching [72], using Markov cache model for size-aware cache admission policy [13]. Compared to these works, GL-Cache is the first system to perform learning on a group of entities and navigates efficiency-throughput trade-off using coarse-grained learning granularity.

6 Conclusion

We propose a new approach for using machine learning to improve cache efficiency: group-level learning. Group-level learning predicts and evicts the least useful object groups. Group-level learning leverages multiple object-group features to adapt to workload and cache size, accumulates stronger signals for learning, and amortizes learning overheads over objects. As a result, it makes better eviction decisions with a tiny overhead. We build GL-Cache in a production cache to demonstrate group-level learning and evaluate it on 118 production block I/O and CDN traces. GL-Cache achieves a significantly higher throughput as compared to all other learned caches while retaining a higher hit ratio. Thus, GL-Cache paves the way for the adoption of learned caches in production systems.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. In addition, our shepherd Carl Waldspurger has given very valuable feedback that improved the writing of this paper. This work was supported in part by a Facebook Ph.D. fellowship, and in part by NSF grants CNS 1901410 and CNS 1956271. The processing and evaluations were performed on Cloudlab [25] and ChameleonCloud [49].

Availability

We open-source our implementation at `https://github.com/Thesys-lab/fast23-glcache`.

References

[1] Xgboost. `https://github.com/dmlc/xgboost`. Accessed: 2022-09-06.

[2] Ismail Ari, Ahmed Amer, Robert B Gramacy, Ethan L Miller, Scott A Brandt, and Darrell DE Long. Acme: Adaptive caching using multiple experts. In *WDAS*, volume 2, pages 143–158, 2002.

[3] Jose A Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. Rudder: Return decomposition for delayed rewards. *Advances in Neural Information Processing Systems*, 32, 2019.

[4] Martin Arlitt, Rich Friedrich, and Tai Jin. Performance evaluation of web proxy cache replacement policies. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 193–206. Springer, 1998.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[6] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. USENIX Association.

[7] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd : Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[8] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[9] 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.

[10] 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.

[11] Daniel S Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical bounds on optimal caching with variable object sizes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2:1–38, 2018.

[12] Daniel S. Berger, Sebastian Henningsen, Florin Ciucu, and Jens B. Schmitt. Maximizing cache hit ratios by variance reduction. *SIGMETRICS Perform. Eval. Rev.*, 43:57–59, sep 2015.

[13] 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.

[14] Adit Bhardwaj and Vaishnav Janardhan. Pecc: Prediction-error correcting cache. In *Workshop on ML for Systems at NeurIPS*, volume 32, 2018.

[15] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, 2017.

[16] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 1, pages 126–134. IEEE, 1999.

[17] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS'97, page 18, USA, 1997. USENIX Association.

[18] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[19] Zheng Chang, Lei Lei, Zhenyu Zhou, Shiwen Mao, and Tapani Ristaniemi. Learn to cache: Machine learning for network edge caching in the big data era. *IEEE Wireless Communications*, 25(3):28–35, 2018.

[20] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[21] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[22] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[23] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020.

[24] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

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

[26] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018.

[27] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, page 94–106. ACM, Nov 2018.

[28] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[29] 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.

[30] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[31] Qilin Fan, Jian Li, Xiuhua Li, Qiang He, Shu Fu, and Sen Wang. Pa-cache: Learning-based popularity-aware content caching in edge networks. *arXiv preprint arXiv:2002.08805*, 2020.

[32] Vladyslav Fedchenko, Giovanni Neglia, and Bruno Ribeiro. Feedforward neural networks for caching: N enough or too much? *SIGMETRICS Perform. Eval. Rev.*, 46:139–142, jan 2019.

[33] Jim Gao. Machine learning applications for data center optimization. `https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42542.pdf`, 2014. Accessed: 2022-04-06.

[34] Robert B. Gramacy, Manfred K. Warmuth, Scott A. Brandt, and Ismail Ari. Adaptive caching by refetching. In *Proceedings of the 15th International Conference on Neural Information Processing Systems*, NIPS'02, page 1489–1496, Cambridge, MA, USA, 2002. MIT Press.

[35] Yu Guan, Xinggong Zhang, and Zongming Guo. Caca: Learning-based content-aware cache admission for video content in edge caching. In *Proceedings of the 27th ACM International Conference on Multimedia*, pages 456–464, 2019.

[36] Beining Han, Zhizhou Ren, Zuofan Wu, Yuan Zhou, and Jian Peng. Off-policy reinforcement learning with delayed rewards, 2021.

[37] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning memory access patterns. In *International Conference on Machine Learning*, pages 1919–1928. PMLR, 2018.

[38] Jeff Heaton. An empirical analysis of feature engineering for predictive modeling. In *SoutheastCon 2016*, pages 1–6. IEEE, 2016.

[39] 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 *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, 2015.

[40] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX*

Symposium on Operating Systems Design and Implementation (OSDI 22), pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[41] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 167–181, 2013.

[42] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2005.

[43] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.

[44] Song Jiang and Xiaodong Zhang. Making lru friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance. *IEEE Transactions on Computers*, 54(8):939–952, 2005.

[45] Shudong Jin and A. Bestavros. Popularity-aware greedy dual-size web proxy caching algorithms. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 254–261, 2000.

[46] Alekh Jindal, Shi Qiao, Rathijit Sen, and Hiren Patel. Microlearner: A fine-grained learning optimizer for big data workloads at microsoft. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2423–2434. IEEE, 2021.

[47] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[48] George Karakostas and D Serpanos. Practical lfu implementation for web caching. *Technical Report TR-622–00*, 2000.

[49] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[50] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Sosd: A benchmark for learned indexes. *arXiv preprint arXiv:1911.13014*, 2019.

[51] 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.

[52] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity Models: Erasure-Coded Resilience for Prediction Serving Systems. *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[53] Jack Kosaian, KV Rashmi, and Shivaram Venkataraman. Learning-based coded computation. *IEEE Journal on Selected Areas in Information Theory (JSAIT special issue on deep learning: mathematical foundations and applications to information science)*, 1(1):227–236, 2020.

[54] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[55] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[56] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361, 2001.

[57] Cong Li. Dlirs: Improving low inter-reference recency set cache replacement policy with dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, pages 59–64, 2018.

[58] Cong Li. CLOCK-pro+: improving CLOCK-pro cache replacement with utility-driven adaptation. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 1–7, Haifa Israel, May 2019. ACM.

[59] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[60] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019.

[61] Zhenmin Li, Zhifeng Chen, Sudarshan M Srinivasan, Yuanyuan Zhou, et al. C-miner: Mining block correlations in storage systems. In *FAST*, volume 4, pages 173–186, 2004.

[62] 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.

[63] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *Proceedings of the 37th International Conference on Machine Learning*, page 6237–6247. PMLR, Nov 2020.

[64] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for c++ server workloads. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[65] Ayush Mangal, Jitesh Jain, Keerat Kaur Guliani, and Omkar Bhalerao. Deap cache: Deep eviction admission and prefetching for cache. *arXiv preprint arXiv:2009.09206*, 2020.

[66] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Learning to steer query optimizers. *arXiv preprint arXiv:2004.03814*, 2020.

[67] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*, 2019.

[68] 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.

[69] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, volume 3, pages 115–130, 2003.

[70] Sailesh Mukil, Prudhviraj Karumanchi, Tharanga Gamaethige, and Shashi Madappa. Cache warming: Leveraging ebs for moving petabytes of data. `https://netflixtechblog.medium.com/cache-warming-leveraging-ebs-for-moving-petabytes-of-data-adcf7a4a78c3`. Accessed: 2022-09-16.

[71] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, NetAI'18, page 48–53, New York, NY, USA, 2018. Association for Computing Machinery.

[72] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, pages 48–53, 2018.

[73] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Trans. Storage*, 4(3), November 2008.

[74] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 985–1000, 2020.

[75] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[76] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[77] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[78] 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.

[79] Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. predis: Penalty and locality aware memory allocation in redis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 193–205, 2019.

[80] Sejin Park and Chanik Park. Frd: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[81] Pelikan. `https://github.com/twitter/pelikan`. Accessed: 2022-09-06.

[82] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with cacheus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354, 2021.

[83] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[84] Amazon SageMaker. Xgboost algorithm. `https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html`. Accessed: 2022-09-06.

[85] D Shasha and T Johnson. 2q: A low overhead high performance buffer management replacement algoritm. In *Proc. 20th Int. Conf. Very Large Databases*, pages 439–450, 1994.

[86] Wanxin Shi, Qing Li, Chao Wang, Gengbiao Shen, Weichao Li, Yu Wu, and Yong Jiang. Leap: learning-based smart edge with caching and prefetching for adaptive video streaming. In *Proceedings of the International Symposium on Quality of Service*, pages 1–10, 2019.

[87] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[88] Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza. Context-aware prefetching at the storage server. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.

[89] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. Leaftl: A learning-based flash translation layer for solid-state drives. *arXiv preprint arXiv:2301.00072*, 2022.

[90] Richard S Sutton. Introduction: The challenge of reinforcement learning. In *Reinforcement Learning*, pages 1–3. Springer, 1992.

[91] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017.

[92] 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 *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.

[93] 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 *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[94] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, February 2015. USENIX Association.

[95] Haonan Wang, Hao He, Mohammad Alizadeh, and Hongzi Mao. Learning caching policies with subsampling. In *NeurIPS Machine Learning for Systems Workshop*, 2019.

[96] Joseph Wang, Anne Holler, Mingshi Wang, and Michael Mui. Productionizing distributed xgboost to train deep tree models with large data sets at uber. `https://eng.uber.com/productionizing-distributed-xgboost/`. Accessed: 2022-04-06.

[97] 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.

[98] 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 *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307–323. USENIX Association, February 2021.

[99] Nan Wu and Pengcheng Li. Phoebe: Reuse-aware online caching with reinforcement learning for emerging storage models. *arXiv preprint arXiv:2011.07160*, 2020.

[100] 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.

[101] Juncheng Yang. libcachesim. `https://github.com/1a1a11a/libCacheSim`. Accessed: 2022-12-16.

[102] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 66–79, 2017.

[103] 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 *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1159–1177, Renton, WA, April 2022. USENIX Association.

[104] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[105] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[106] 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.

[107] Yuchao Zhang, Pengmiao Li, Zhili Zhang, Bo Bai, Gong Zhang, Wendong Wang, Bo Lian, and Ke Xu. Autosight: Distributed edge caching in short video network. *IEEE Network*, 34:194–199, May 2020.

[108] Chen Zhong, Xingsheng Zhao, and Song Jiang. Lirs2: an improved lirs replacement algorithm. In *SYSTOR'21: Proceedings of the 14th ACM International Conference on Systems and Storage*, 2021.

[109] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on parallel and distributed systems*, 15(6):505–519, 2004.

[110] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.

SHADE: Enable Fundamental Cacheability for Distributed Deep Learning Training

Redwan Ibne Seraj Khan
Virginia Tech

Ahmad Hossein Yazdani
Virginia Tech

Yuqi Fu
University of Virginia

Arnab K. Paul
BITS Pilani, Goa

Bo Ji
Virginia Tech

Xun Jian
Virginia Tech

Yue Cheng
University of Virginia

Ali R. Butt
Virginia Tech

Abstract

Deep learning training (DLT) applications exhibit unique I/O workload behaviors that pose new challenges for storage system design. DLT is I/O intensive since data samples need to be fetched continuously from a remote storage. Accelerators such as GPUs have been extensively used to support these applications. As accelerators become more powerful and more data-hungry, the I/O performance lags behind. This creates a crucial performance bottleneck, especially in distributed DLT. At the same time, the exponentially growing dataset sizes make it impossible to store these datasets entirely in memory. While today's DLT frameworks typically use a random sampling policy that treat all samples uniformly equally, recent findings indicate that not all samples are equally important and different data samples contribute differently towards improving the accuracy of a model. This observation creates an opportunity for DLT I/O optimizations by exploiting the data locality enabled by importance sampling.

To this end, we design and implement SHADE, a new DLT-aware caching system that detects fine-grained importance variations at per-sample level and leverages the variance to make informed caching decisions for a distributed DLT job. SHADE adopts a novel, rank-based approach, which captures the relative importance of data samples across different mini-batches. SHADE then dynamically updates the importance scores of all samples during training. With these techniques, SHADE manages to significantly improve the cache hit ratio of the DLT job, and thus, improves the job's training performance. Evaluation with representative computer vision (CV) models shows that SHADE, with a small cache, improves the cache hit ratio by up to 4.5× compared to the LRU caching policy.

1 Introduction

Deep learning (DL) approaches are increasingly being employed to solve crucial complex problems. The use of DL has become common in disparate domains such as health sciences [28, 29, 43, 64], environmental sciences [41, 47], bio-technical systems [48], high-energy scientific experiments [16], finance [25,33,35,39], smart cities [12,19], industrial production [13,79], autonomous vehicles, and IoT systems [45,58,72]. Moreover, DL has given rise to a huge market that is expected to reach 12.12 billion dollars by 2025 [4]. To meet the demands of unprecedented scale and performance, DL researchers and practitioners are developing distributed DL, which employs distributed computing and storage resources to support DL. While promising, the approach poses numerous challenges in handling massive workloads while keeping the usage cost in check.

DLT is extremely compute-intensive and data-intensive [24], and the resource demands vary at different phases of the process [22,42]. A key challenge is efficiently matching the DL application needs with available system resources. A common practice is to scale up/out a DL training job using multiple compute accelerators such as GPUs, FPGAs, or custom ASICs; that is, by using data parallelism [71] with each accelerator, e.g., GPU, holding a replica of the model and processing a subset of the training data in parallel.

A large body of research has focused on optimizing the efficiency of computing [53, 57], scheduling [37, 82], and data communication [74, 77, 81] for DL jobs. This is because data-parallel DL training is both compute-intensive—typically requiring multiple GPUs to train in parallel—and communication-intensive [17,59,75]—newly calculated model gradients are transferred or broadcast to all the involved GPUs for iterative model updates. However, as state-of-the-art research [14,56] demonstrates, the efficiency of data storage and retrieval can also significantly impact the end-to-end performance of DL training.

To better understand the impact of data storage configuration on distributed DL training efficiency, we perform an experiment to study the performance difference when distributed DL jobs are run using a local or a remote storage medium. Figure 1 shows that remote storage mediums can significantly impact the training time ($\sim 2.5\times$) compared to faster storage mediums, i.e., RAM, even though all the rest

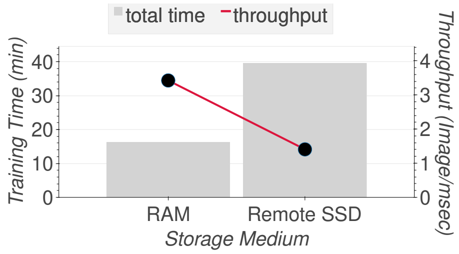


Figure 1: Training throughput and time comparison of a single job using 2 nodes and 8 GPUs with ResNet-18. Remote storage comprises of SSDs on a BeeGFS server.

of the training configurations were kept the same. This result is in line with recent studies [20, 70], which show that I/O can take up from 85-90% of the total training time. As high-performing accelerators can consume the training data samples faster, efficient I/O can significantly reduce the training time.

However, it remains challenging to improve the I/O efficiency for distributed DLT as the I/O workloads of a DLT job exhibit unique patterns: (1) full-object, sequential, read-only accesses at per-object level; (2) dominant, small, random I/Os spread across the whole training sample dataset [88]; and (3) highly concurrent I/Os [70]. Today's high-performance and distributed storage systems, such as parallel file systems (PFS) [21, 40], network file systems (NFS) [63], and cloud object stores [2, 5] are inefficient at supporting dsitributed DL applications. This is especially true given the excessive metadata overhead for small-I/O-intensive accesses [85].

For efficient I/O, faster storage mediums like RAM are needed, but compared to increasingly large training datasets that can range from terabytes [36, 67] to petabytes [10], these resources are often too small, even on large supercomputers like Piz Daint (64GB RAM/node) [3] and Fugaku (32GB RAM/node) [8]. Moreover, because of the high cost of GPUs, DL jobs are mainly run by renting GPU Spot VMs [1, 9, 15, 49, 56, 78] that are 6-8× cheaper than dedicated VMs. As these VMs are preemptive, meaning these can be terminated at any time depending on available resources, DL training has to be resumed from a checkpoint on a different VM leading to the loss of local SSD state. As a result, instead of local SSDs, large datasets are put in persistent cloud storage, and training is conducted on VMs that access the cloud storage remotely.

Worse, conventional wisdom holds that the I/O workload of a DL training job is not *cache-friendly* due to the aforementioned I/O randomness and lack of data locality [85]. This property renders existing caching policies (such as LRU and LFU) ineffective, as there is no recency or frequency pattern to exploit. Recent work such as Mercury [84], CoorDL [65], Quiver [56], and Hoard [69] try to solve this I/O problem by employing caching techniques. Unfortunately, none of them provides fundamental solutions that enable the ability to cache (i.e., *cacheability*) a DLT job's working set. The main reason is that these works consider that each sample will only be accessed once in every epoch (one iteration over the dataset). However, as has been shown by prior work [50], some samples are more important than others in DL training. Hence, if we can design effective mechanisms and policies to exploit this importance variance, we can fundamentally improve the cacheability for DL training.

In this paper, we show that we can deliver better cacheability by designing a new dataset sampling algorithm inspired by importance-sampling [61] and an effective caching policy atop that. DL models are trained on a dataset in batches (multiple equal partitions of the entire dataset). Our sampling algorithm combines the intra-batch importance of individual data samples with inter-batch importance to detect the most important samples for placing in the in-memory pooled cache. We develop a novel technique of *rank-based importance* that ranks the training samples within a batch based on their contributions to increasing the overall accuracy of the model. Rank-based importance further helps increase the probability of identifying (predicting) the most important samples in later epochs. Using this technique, we further design a *priority-based sampling strategy* that ensures multiple accesses to the important samples within an epoch to train more on hard-to-learn samples to increase the accuracy improvement rate. As a result, our caching solution keeps the most important samples in the cache and avoids random evictions, which in turn improves the cache hit ratio and training throughput.

Specifically, this paper makes the following contributions.

- We introduce a novel, rank-based importance calculation approach to precisely identify the relative importance of data samples for DLT jobs.
- We design a priority-based sampling policy to exploit the data locality of samples.
- We present the design and implementation of SHADE, a new DLT-aware caching system that incorporates rank-based importance scores and the priority-based sampling policy to improve the I/O efficiency for DLT jobs.
- We incorporate and evaluate SHADE in the widely-used DLT framework PyTorch and compare SHADE against a series of baseline and advanced caching and sampling methods. Our results show that SHADE: improves the read hit ratio by up to 4.5× given the same cache size, increases the training throughput by up to 2.7×, and reaches accuracy convergence by up to 3.3× faster compared to a baseline LRU caching policy.

SHADE is open source and publicly available at:
https://github.com/R-I-S-Khan/SHADE.

2 Background

2.1 Distributed Deep Learning Training

There are mainly three types of distributed DL training techniques: *data-parallel training* [71], *model-parallel train-*

ing [30], and *pipeline parallelism* [44] that combines data-parallel and model-parallel training. While this paper focuses mainly on data-parallel training based on Stochastic Gradient Descent (SGD), our approach is applicable to other training methods as well.

A Deep Neural Network (DNN) model consists of multiple layers of computation units whose output is the input for subsequent units. DNN model training consists of a forward propagation method, which sequentially moves information related to the input data through all model layers and generates a prediction. For example, in an image recognition application, image pixels information is moved through the layers for predicting image contents. To generate the prediction, DL defines a cost/loss function with respect to the forward propagation output and ground truth labels. The DL process aims to minimize the cost function through a process of increasing or decreasing the weights of the outputs of the intermediary layers of the model so that it can improve its prediction. This step is known as backward propagation, which adjusts the parameters of the DL model starting from the outermost layer back up to the input layer through a technique known as gradient descent optimization. Gradient descent adjusts the parameters in the opposite direction of the gradient. SGD is a stochastic approximation of gradient descent optimization; instead of calculating the gradient from the entire data set, SGD randomly selects a subset of training data samples from the entire dataset to reduce computation cost.

In a typical data-parallel, SGD-based training, the whole training dataset is partitioned and processed in parallel by multiple GPU devices. Each GPU has a replica of the same DNN model, which is iteratively synchronized with other GPUs using centralized communication techniques, e.g., parameter server [59]) or decentralized communication techniques, e.g., all-reduce [17].

2.2 I/O Characteristics of Data-Parallel DL Training

DL training applications feature unique characteristics that differentiate them from conventional data-intensive applications such as big data analytics [31, 83] and web applications [6, 7]. A DL training job typically runs multiple epochs, with each epoch consuming the entire training dataset once in a random permutation order. Each epoch is further divided into multiple batches. At the beginning of processing a batch, each GPU process loads a randomly-sampled bulk (i.e., a mini-batch) of training data whose size is configurable. These behaviors lead to highly-concurrent, read-only, repetitive, and totally random I/O accesses. Therefore, a common belief is that such I/O patterns are not cache-friendly to traditional caching policies that exploit recency and/or frequency-based data locality, such as the widely used LRU, LFU, and ARC [62].

2.3 DL Training with Importance Sampling

Traditionally, SGD-based DL training is oblivious to the "importance" of training samples and simply applies random sampling or shuffling to generate a random permutation order at the end of each training epoch, thereby treating all training samples equally. Recently, researchers found that in SGD-based DL training, a specific set of training samples tend to generate little-to-no impact on the model quality and, therefore, can be ignored [50, 61]. This process of finding the set of training samples that are more important than others, i.e., contribute the greatest towards the loss function, is known as importance sampling. That is, a few samples would lead to a higher loss between hidden layer output and target label in backward propagation after a few epochs.

Therefore, by prioritizing training using samples with relatively higher importance, i.e., the ability to contribute towards building model accuracy, a DL training job can achieve improvement in both training time and test errors [46, 50].

In SGD, gradient $g(x)$ is estimated by sampling from a uniform distribution p where x is a data sample from a minibatch. Importance sampling estimates $g(x)$ using a new data distribution q (such that $q(x) > 0$ whenever $p(x) > 0$) to speed up the process. That is,

$$\mathbb{E}_{p(x)}[g(x)] = \mathbb{E}_{q(x)}[\frac{p(x)}{q(x)}g(x)] \tag{1}$$

It has been proved [11] that the variance of gradient is minimized when Eq. 2 is maintained, i.e., to ensure gradient variance reduction, optimal data distribution $q^*(x)$ should be proportional to sample's gradient norm $|g(x)|$.

$$q^*(x) \propto p(x)|g(x)| \tag{2}$$

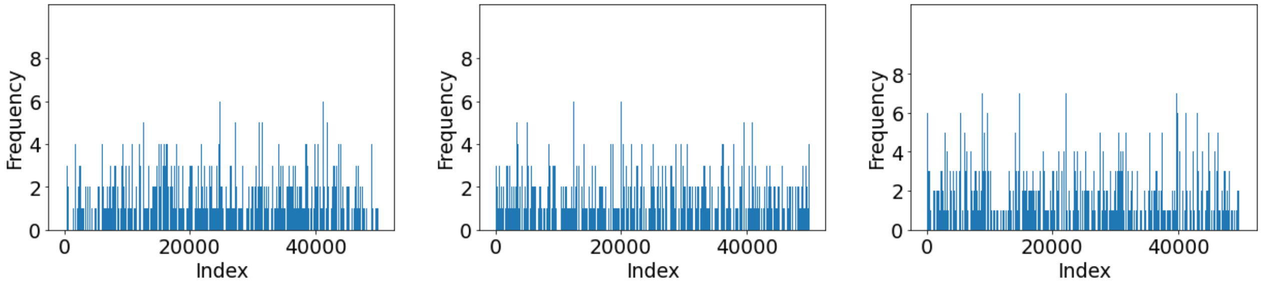
In practice, the feed-forward loss is often used to measure the importance of each data sample as an alternative of gradient.

3 Motivation

3.1 Exploiting Importance Sampling

As discussed earlier, the shuffling-based sampling method passes over the entire training dataset in each epoch, making DL training not cache-friendly and failing to make efficient use of faster storage mediums such as main memory or SSDs. However, as observed in Figure 1, there exists ample opportunity to make efficient use of faster storage devices to enhance the performance of DL training. In this regard, importance sampling treats training samples differently and introduces inherent data locality that can be exploited by a caching system to make better use of faster storage mediums.

To better understand the implication of importance sampling on DL training and dataset caching, we analyze importance sampling based training over benchmarking datasets.

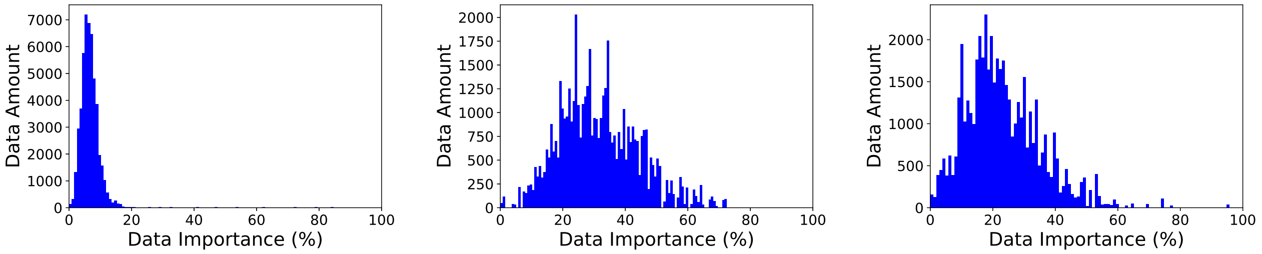


(a) Sample access pattern in epoch 1.

(b) Sample access pattern in epoch 20.

(c) Sample access pattern in epoch 89.

Figure 2: Frequency of samples accessed across different epochs in default single process importance sampling (CIFAR-10).



(a) Epoch 1.

(b) Epoch 20.

(c) Epoch 89.

Figure 3: Distribution of data importance as the number of epochs increases in single process default importance sampling on the CIFAR-10 dataset. Data importance is the ability of a sample to contribute towards improving the accuracy of the model.

Sample Access Pattern. We first analyze the sample access pattern of importance sampling. We use the CIFAR-10 dataset [54] to train a ResNet-18 DNN model using a loss-based importance sampling algorithm [50] for a single GPU. As shown in Figure 2, 26.5%, 26.6%, and 26.2% of the samples are accessed more than once in epochs 1, 20, and 89, respectively. More importantly, 9.6% of the samples are accessed 3 times or more in epoch 89, indicating a good data locality within a training epoch.

Data Importance Distribution. We further analyze the importance scores of training samples. Unlike standard random sampling, which treats each sample equally, not all samples contribute equally to model training. As shown in Figure 3(a)-(c), in epoch 1, the importance scores of all samples are clustered towards the least-important end of the spectrum; whereas during epochs 20 and 89, more samples become more important. In particular, in epoch 20, around 49.91% of samples have a normalized importance score greater than 30%. This observation further implies that the importance information could be exploited by a priority caching policy to optimize the I/O efficiency of DL training.

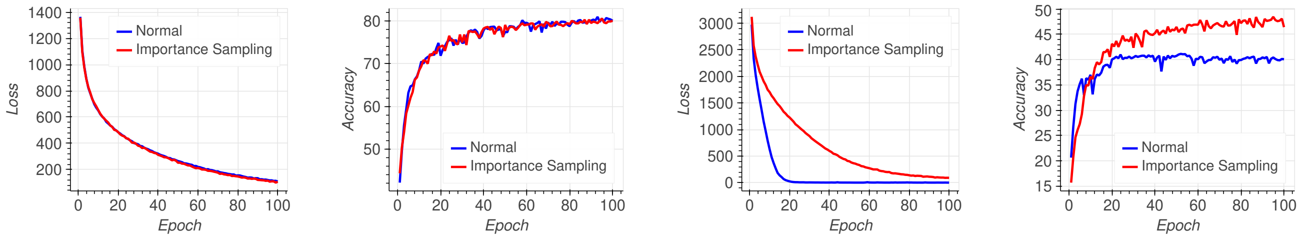
Impact of Importance Sampling on Training. Next, we analyze the impact of importance sampling on training quality. In this test, we use the CIFAR-10 and CIFAR-100 datasets and train a ResNet-18 model using standard random sampling and a loss-based importance sampling method. As shown in Figure 4(a)-(b), we verify that importance sampling incurs negligible impact on the model accuracy for the CIFAR-10 dataset. The CIFAR-100 dataset is much harder to predict than CIFAR-10 due to the larger number of classes present in the dataset. Figure 4(c) shows that importance sampling does not have a drastic loss degradation implying that it has a good learning rate, which further contributes to its improved accuracy. Figure 4(d) shows that importance sampling can achieve better accuracy in under 20 epochs than the accuracy achieved by normal baseline random sampling in 100 epochs. This is because random sampling just shuffles the dataset indices, which does not contribute much towards quickly learning fine-grained details of the dataset.

4 SHADE Design

Our study in §2 sheds light on the potential to enable fundamental data locality for DL training workloads and motivates a new caching system co-designed with the DL training framework. This section presents the challenges and design principles of SHADE, followed by the design detail.

4.1 Challenges

Our goal is to achieve a caching system that can exploit importance sampling to improve the cache efficiency for DLT's I/O workload. One may think that a priority-based caching policy that always prioritizes the most important samples could effectively improve the read hit ratio. However, as shown in Figure 5, a naive priority-based caching policy achieves the same low read hit ratio as standard LRU and LFU. Ideally,



(a) Loss vs. Epoch (CIFAR-10). (b) Accuracy vs. Epoch (CIFAR-10). (c) Loss vs. Epoch (CIFAR-100). (d) Accuracy vs. Epoch (CIFAR-100).

Figure 4: Comparison of loss and accuracy convergence of ResNet-18 model using single process default importance sampling against baseline training on the CIFAR-10 and CIFAR-100 datasets.

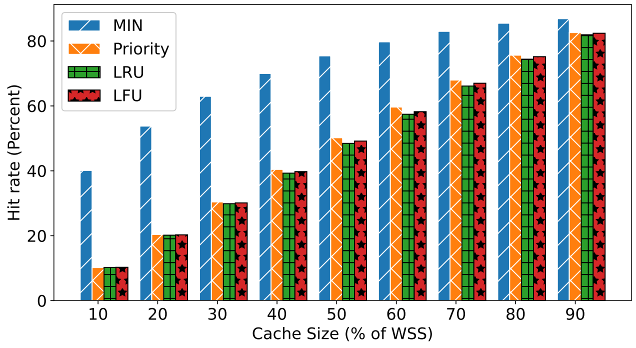


Figure 5: Comparison of different caching policies during ResNet-18 model training over the CIFAR-10 dataset. Working Set Size (WSS) denotes the percentage of cached dataset.

Belady's MIN cache replacement policy [18] achieves an optimal read hit ratio assuming perfect future knowledge: Belady's MIN replaces the item that will be accessed furthest in the future (Figure 5). In the context of DLT, an ideal priority caching policy would accurately capture the priorities (i.e., importance scores) of training samples, resembling the optimal behavior of the offline MIN. To make it even better, the policy could take advantage of the importance information to prefetch important samples into the cache. This way, the new policy can potentially outperform Belady's MIN when incorporating prefetching [87].

The key insight of this paper is that DLT treats different training samples differently and that the priorities of I/O accesses are inherently predictable, therefore exposing interesting exploitation opportunities to fundamentally improve the I/O efficiency. However, it also poses non-trivial challenges to effectively translate the potential exploitation opportunities to the I/O efficiency improvement.

First, default importance sampling (importance sampling considered in prior works) assigns per-minibatch scores, which are too coarse-grained and inaccurate. That is, all samples of a single minibatch are, by default, assigned the same importance scores. This creates ambiguity, which leads to inaccurate estimation of the per-sample importance and, thus, loss of cache efficiency. Ideally, we would want an importance score that precisely tells us the relative importance that each sample carries within a minibatch.

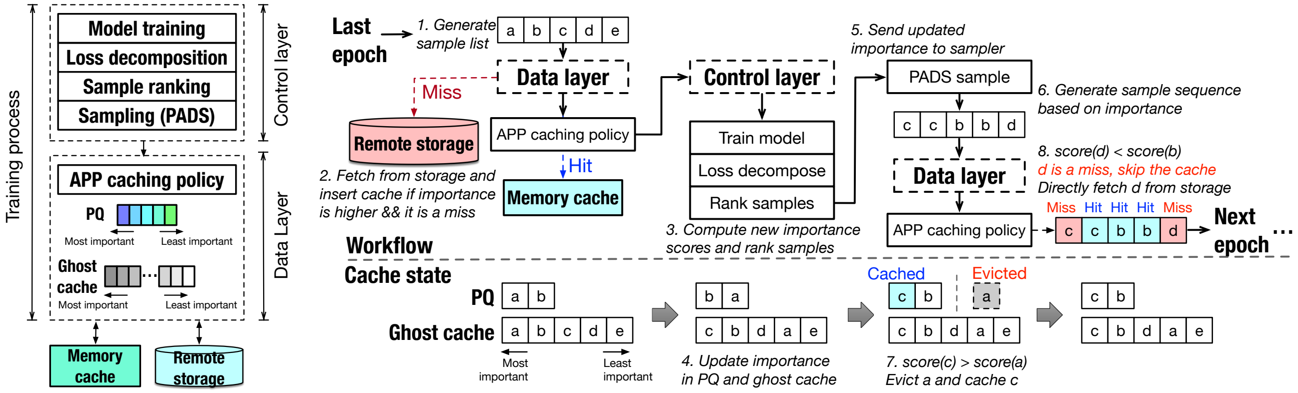
Second, even if important samples are identified properly, aggressively feeding the DL model with repetitive samples might make training model biased. Thus, it is necessary to ensure that the accuracy is not compromised while trying to increase the hit rate of samples.

Third, importance scores are constantly changing and may get stale quickly. The same sample in a later minibatch may contribute differently toward the model than it did in an earlier minibatch. Thus, capturing the most up-to-date importance score information is imperative to make informed caching decisions.

In the next section, we discuss how we use four novel techniques to address each of these challenges.

4.2 SHADE Overview

SHADE consists of two main components—the control layer and the data layer. The control layer provides the data layer with the list of samples needed for training. For the first iteration, the data layer fetches samples from a remote storage and populates the cache with the samples that are to be accessed first. During training, the control layer finds the importance (loss decomposition + ranking) associated with the samples and the priority queue (PQ) and ghost cache tracking the importance of samples in the data layer are updated. Based on the newer importance, a *sampler* in the control layer prepares a samples list with associated repetitions information. When the data layer receives the list of samples, it checks whether it is beneficial to cache a newer item instead of evicting a cached prior sample. Let's suppose the sample being accessed has higher importance than the min_sample (sample having lowest importance in the current cache). In this case, the min_sample is evicted, and the current sample is cached using our new Adaptive Priority-aware Prediction (APP) cache policy. This process is repeated throughout the entire DL training. As SHADE keeps the most important samples in the distributed cache and repeatedly uses these hard-to-learn samples for training, it can ensure improved rate of accuracy and a good cache hit ratio. Figure 6 shows the architecture of SHADE along with the components and interactions therein.



(a) SHADE's logical architecture.

(b) SHADE's workflow.

Figure 6: SHADE architecture overview. (b) In illustration of how SHADE's components interact in a single epoch.

4.2.1 Control Layer

The SHADE control layer performs two main functions. (1) It calculates the importance scores associated with data samples, and (2) it samples the data for different training processes. The importance scores are then transferred to the data layer in real time for making prefetching and caching decisions.

The SHADE control layer features three techniques to find accurate, fine-grained importance scores for each data sample. The first technique finds out the importance of samples in per-sample granularity (i.e., fine-grained). The second technique uses fine-grained importance and ranks the samples to make them suitable for priority-based caching. Finally, the third technique uses rank-based importance to build a list of important samples with repetitions to be used for training that will increase the read hit ratio and maintain a good learning curve.

Loss Decomposition. In default importance sampling, the forward training loss is calculated for minibatches, and this forward loss is then assigned as the importance score for all the samples in the minibatch [61]. As a result, the default importance sampling method calculates the ability of a minibatch to contribute towards improving the overall accuracy of the model instead of the data samples themselves. However, as expected, not all the samples of a minibatch contribute equally to the accuracy improvement of the model. Therefore, we need the sample-level loss information, i.e., the loss of individual samples of a minibatch, to calculate the importance score of each data sample.

To address the first challenge concerning the coarse-grained importance scores at the minibatch granularity, SHADE uses both the sample-level and minibatch-level cross entropy loss information to decompose the coarse-grained importance scores into per-sample scores. The cross entropy for each sample denotes the uncertainty with which the model could predict the class label for a sample. Measuring the uncertainty helps SHADE to detect the importance of a sample.

Assume a minibatch has S data samples, and T represents the number of class labels. This constructs an output layer for the DNN model with a matrix that has a dimension of $S \times T$. Each row of the matrix encodes the raw likelihood or logits of a sample for each of the T-class labels. To capture the contribution of each data sample, SHADE decomposes the loss function and calculates the loss corresponding to each sample in the minibatch. SHADE decomposes the loss function using two steps. In the first step, SHADE calculates the categorical-cross entropy for each sample in the minibatch. The categorical-cross entropy for each sample, E_{sample}, is defined in Eq. 3:

$$E_{sample} = -\sum_{i=1}^{T} T_i \log S_i, \tag{3}$$

where T_i represents the hot-encoded truth label for a given sample under class i, and S_i denotes the softmax probability for a sample in a minibatch for the class i.

S_i is calculated using Eq. 4:

$$S_i = \frac{e^{r_i}}{\sum_{j=1}^{T} e^{r_j}}, \tag{4}$$

where r_i denotes the raw likelihood of a sample for class i, and the denominator is a normalization term. SHADE uses a softmax normalization over a standard normalization method for two reasons. (i) This method can effectively identify small and large variations in raw logit values and thus assign the importance scores accordingly. (ii) The raw logit values can be negative, so taking exponents ensures that we always end up with a positive value. SHADE uses the per-sample-based entropy loss for finding and feeding the model with the most important samples.

The second step of the loss decomposition involves calculating the minibatch importance necessary for adjusting the model weights. As softmax is continuously differentiable, it is possible to calculate the derivative of the cost function with respect to every weight of a DNN model. SHADE uses all the per-sample-based entropy losses found from the first stage (Eq. 3) for calculating a mean entropy loss according to Eq. 5:

$$E_{batch} = \frac{\sum_{k=1}^{S} E_{sample_k}}{S} \tag{5}$$

A higher entropy for a sample means that the model generates multiple predictions for a single sample out of the several T different possibilities, i.e., the model faces more difficulties in generating a single accurate prediction for that sample. Correspondingly, a lower entropy for a single sample signifies that the model can generate a single prediction for it with high enough accuracy. Thus, a lower entropy value for a sample means that the sample is not highly important in increasing the accuracy of the model in later epochs, and a higher entropy value signifies the opposite. The reason is straightforward: samples that the model has already learned cannot help much in increasing the accuracy of the model in later epochs, and only by learning the harder samples can the DLT job improve the accuracy. Ideally, an entropy value of zero means that the difference between the predicted and ground-truth label is an absolute zero and that the sample is accurately learned. In practice, however, the entropy cannot reach zero as there are no useful models that have 100% accuracy.

Our goal is to prioritize samples that have higher entropy during model training so that the model can learn these hard-to-learn samples better. Consequently, the loss decomposition method enables SHADE to capture hard-to-learn samples from a minibatch without extra transformation of the raw data.

Rank-based Importance Score. Even though the per-sample entropy score provides a simple tool for quantifying the importance of different samples, it does not tell how much different samples contribute to the accuracy of the model when sampled together in a single minibatch. The relative rankings allow SHADE to prioritize the most important set of samples from each minibatch and, thus, the entire training dataset. To identify the *relative* contribution of a sample in a minibatch, we derive a log-based ranking method shown in Eq. 6:

$$rank_i = \log(\sum_{k=1,k\neq i}^{B} I(l_i > l_k) + b_0) \tag{6}$$

The rank-based importance score for the i^{th} sample in a minibatch with B samples is denoted by $rank_i$. l_i and l_k denote the entropy loss for the i^{th} and k^{th} sample, respectively. b_0 is a bias term used for fixing the range of ranks in the log scale. I is an identity function that returns 1 when the condition $l_i > l_k$ is true, and 0 otherwise. For each k item in a batch B, this condition helps to place each sample in the proper rank in a minibatch. A sample having a higher loss gets a higher rank.

Consider the following example. Assume two minibatches, $B1$ and $B2$, contain samples <4, 5, 6> and <7, 8, 9>, respectively. Assume the samples in $B1$ have entropy scores of <0.3, 0.5, 0.4> and samples in $B2$ have <0.6, 1.2, 0.8>, respectively. These entropy values are raw values, which will be problematic when comparing the sample importance across minibatches. For example, a priority-queue-based cache would rank sample 5 from $B1$ in the lower half globally when sorting all samples from both the two minibatches, even though sample 5 is the most important one in $B1$. Sorting these samples by the entropy scores would give us a relative rank with respect to each of the two minibatches $B1$ and $B2$, meaning sample 5 and 8 are the most important in $B1$ and $B2$, respectively.

To get the accurate changes in importance score, i.e., whether the importance score is increasing, staying unchanged, or decreasing, SHADE uses the log scale. In this case, the relative importance scores remain in the same range and can be used for priority differentiation in a priority queue data structure. Relative scores are desirable for three reasons.

First, our method guarantees that the per-sample-based importance scores from different minibatches and epochs are in the same range in order to precisely differentiate their priorities in the cache. Different samples from different minibatches would share the same rank in our method if they contributed the same proportion in improving the accuracy of the model when grouped in their corresponding minibatches. Whereas in the default importance sampling method, all samples from the same minibatch are assigned the same importance, resulting in at most one minibatch that will have the highest importance score, r_{max}. This is erroneous as all samples in the same minibatch do not contribute equally to improving the accuracy. When the number of minibatches is more than the minibatch size, SHADE is guaranteed to capture more important samples than the default method, and this, in turn, helps in improving the read hit ratio of the cache. Specifically, if the DLT job is configured to train on N samples with a minibatch size of B, where N is significantly larger than B (which is the common case in DLT), then SHADE can effectively identify (N/B) important samples in one epoch. In contrast, the default method can only capture B important samples.

Second, although models are constantly being updated during the training, training against harder-to-learn samples may help mitigate the high volatility of the accuracy rate, therefore leading to smooth model training.

Third, the relative ranks make it easy to predict the data importance online: the top x% important data in a minibatch is guaranteed to be within the set of the top x% of the whole dataset according to our defined importance score. Based on this property, SHADE effectively offers an implicit prefetching mechanism, which will be described in Algorithm 1.

Priority-based Adaptive Data Sampling (*PADS*). At the end of an epoch, when SHADE has calculated the (rank-based) importance of samples, the SHADE sampler sends the data layer the list of sample indices that should be used for training.

However, instead of naive random shuffling, the SHADE sampler first prioritizes the samples that contributed the most to the accuracy of the model by constructing a multinomial probability distribution of the data samples, as there are many possible outcomes/selections of the dataset. Based on the generated distribution, the sampler builds a list of important samples for shuffling and sharding across different DLT processes. SHADE seamlessly combines prefetching and caching:

based on the updated importance scores and the list of samples provided by the sampler, SHADE data layer prefetches the most important samples to the distributed in-memory cache. The sampler provides the data layer with a list of repetitive samples, which helps the data layer automatically prefetch important samples in real-time. The design strikes a balance between the cache efficiency (read hit ratio) and model accuracy. SHADE keeps track of the loss convergence of the model in real-time to decide the number of repetitive sample accesses in order to boost the hit rate without sacrificing the accuracy of the model. To do so, on noticing a steep decline in the loss convergence curve or a stagnant accuracy curve, SHADE intentionally shuffles the important samples to avoid training against a small subset of the most important samples. This way, the system is able to mitigate aggressive importance sampling, which minimizes training biases.

SHADE's *PADS* policy plays a crucial role in increasing the hit rate of a limited-sized cache and, in certain cases, can even outperform offline MIN. Consider the following example. Assume we have ten samples <1, 2, 3, 4, 5, 6, 7, 8, 9, 10> during training. The samples have no repetitions, as random sampling does not consider the importance of the samples. Assume each training process trains on five samples, and the cache holds two samples. Assume the sampler provides samples [1, 3, 5, 6, 8] randomly for training. Then, offline Belady's MIN will put any of the two samples that will be used by the training process. Assume it puts [1, 3] in the cache. In this case, the hit rate will be only 40% <hits 1, 3>.

In the case for SHADE, *PADS* would create a samples list with repetitions based on importance. Assume the samples are [4, 7, 3, 5, 3, 5, 2, 2, 1, 10] and samples 3, 5, and 2 are the most important. Then *PADS* would provide the training process with the samples <3, 3, 5, 5, 2> from the list of samples so that <3, 5> can be cached. The hit rate is now 80% <hits 3 3 5 5>. Even if we consider repetitions in the case of Belady's MIN, it is bounded by the access pattern of the sample list provided by the random sampler. For example, if the sampler provides <4, 5, 5, 1, 10> to the training process, on knowing the future, Belady's MIN can, at most, cache 5 (as it is needed twice) along with another sample. Assume it caches <5, 10>, so the hit rate will be only 60% <hits 5, 5, 10>.

SHADE's sampling method is fully decentralized and does not require a centralized server to coordinate. Decentralized sampling means that each training process derives the importance scores of the samples independently based on its own local model training.

4.2.2 Data Layer

The SHADE data layer provides mechanisms and policies for cache eviction and prefetching.

A challenge regarding sample caching is that the importance scores are constantly changing, even within an epoch, as one data sample can be accessed in multiple minibatches. In order to address this challenge, we design a new cache policy called *Adaptive Priority-Aware Prediction (APP)*, a dynamic policy that updates the importance score of a data sample as soon as the importance score changes.

SHADE's data layer consists of two components: *an indexer* and *a pooled in-memory cache* that spans multiple key-value storage (KVS) servers. The index uses two heap-based priority queues (PQ and ghost cache) for tracking the samples along with their associated rank and access frequency for each training process.

The data layer introduces index numbering for each individual data sample. Index numbering enables the control layer to assign importance score at the sample granularity. Once the control layer calculates and assigns importance scores for the data samples, the indexer inserts the data sample index numbers, but not the actual sample data, into the PQ (priority queue for the current state of the cache). During data loading, the cache will use the importance scores provided by the indexer to make informed prefetching and eviction decisions. The data layer also performs serialization and deserialization when inserting and fetching image samples to and from the cache. The APP caching policy is shown in Algorithm 1.

The PQ and the ghost cache are sorted by the importance scores. PQ keeps track of the metadata state of currently cached samples in the cache, while the ghost cache tracks all the metadata state of the samples that have ever been cached (including those that have been evicted). The ghost cache entries do not store the actual data samples, but rather store a mapping between the data sample ID (the sample index) and the metadata tuple record of <ir, af>, where ir is the importance score and af is the access frequency.

During training, a cached sample might lose its importance if it is well-learned; that is, it might lose its priority in the cache and gets evicted. At the same time, another sample that has been evicted previously might turn out to be important and therefore gets inserted into the cache. The ghost cache helps decide whether a previously-evicted sample could be brought back into the cache.

When the cache is full, and the data sample to be processed is a miss, SHADE checks ghost cache for the previous importance score of the data sample (Algorithm 1 line 12): if this data sample had been previously evicted out of the cache, it should be included in the ghost cache. If the most recent importance score of this data sample is greater than or equal to that of the cached sample that has the smallest importance score, the currently-cached least-important sample (min_sample) is evicted from the data cache as well as from PQ. After the eviction, the data sample that is to be processed (and was previously evicted) is inserted in the cache (line 18).

In summary, by comparing the current importance scores of data samples already in the cache and that of the most recent importance score of the current data sample being processed, SHADE's data layer predicts and maximizes the likelihood of a sample being reused in the cache in the future.

Algorithm 1: Adaptive Priority-aware Prediction (APP) Cache.

```
Input and Initialization:
PQ: Priority queue for currently-cached samples, ghost_cache: Priority queue for all previously-trained samples
for epoch in total_epochs do
    for s in sample_dataset do
        v = score(s) # Calculate importance score based on Eq 3 and Eq 6
        ghost_cache.set(s,v) # Insert/update in ghost_cache
        if cache_hit then
            cache.get(s)
        else if cache_miss and cache_not_full then
            cache.insert(s) and PQ.set(s,v)
        else
            if cache_miss and ghost_cache.exist(s) then
                x = ghost_cache.get(s).score # Get the most recent score of data sample
                min_sample,min_score = PQ.min() # Find the sample with the minimal score in the cache
                # Check if the sample to be processed is more important than the least important one stored in the cache
                if $x \geq min\_score$ then
                    cache.evict(PQ.pop(min_sample)) # Evict the least important sample from cache
                    cache.insert(s) and PQ.set(s,v) # Insert this sample into the cache and PQ
                else
                    read_from_storage(s) # The data is less important than any samples currently cached, skip caching
            else
                read_from_storage(s) # Evicting a known sample for an unknown one may not be beneficial, skip caching
```

5 Implementation

SHADE is implemented in PyTorch 1.7. PyTorch has three main components: `Dataset`, `Sampler`, and `DataLoader`. Dataset class provides the image dataset access points and exposes a *__get_item__* method that fetches a sample along with its target label for a given index. `Sampler` provides subsets of samples of the dataset to the training processes in random permutations. `DataLoader` uses the information provided by the sampler to load the samples in minibatches with the help of worker processes. In SHADE, we implement a new class by inheriting the PyTorch `Dataset` class. The `ShadeDataset` class has functionalities to combine the samples and their corresponding class labels so that `Dataloader` can fetch data samples easily from the remote storage.

We extend the `DistributedSampler` class to prepare the `ShadeSampler` class that has the main logic of the SHADE's *PADS* policy. It has APIs for communicating with the training processes to receive the calculated per-sample entropy value for each minibatch. At the end of each epoch, `ShadeSampler` forwards the important samples to the training processes.

We introduce the logic for the data layer by overriding the *__get_item__* and *__len__* method in the `ShadeDataset` class. The *__len__* method returns the total length of the `ShadeDataset`, and the *__get_item__* method exposes the indices associated with the data samples that enables the client layer to find importance in per-sample granularity. In addition, the *__get_item__* method is connected to the in-memory pooled cache to make decisions on caching and eviction based on the heap-based PQs of the data layer using the *APP* cache policy. For the in-memory pooled cache, we use Redis [7].

We have implemented an analysis framework atop the setup to facilitate experimentation and statistics collection. The framework takes as input the configurations of the experiment, which include paths of the training dataset, master's address:port, number of training nodes, number of GPU devices, number of epochs to run the test, the batch size, and the DNN model to be trained. The framework then sets up the environment accordingly. It collects GPU-related statistics using nvidia-smi [66] and I/O-related statistics using sar [73]. SHADE's implementation is system agnostic – DL practitioners do not need to write new code to use SHADE in their systems. SHADE does not use any extra system-level resources compared to a normal DL training with local/global caching.

6 Evaluation

6.1 Experimental Setup

Our study covers distributed training with multiple GPUs and a remote storage deployed on Chameleon Cloud [51]. Several recent works [26, 27, 52] have used Chameleon Cloud for conducting high-performance experiments, making it a representative testing platform. Our method is compared against baseline distributed training in PyTorch—one of the most popular frameworks for deep learning [68]. Although caching policies are not publicly available in PyTorch, we have built an LRU caching policy on top of PyTorch to ensure a thorough evaluation of our proposed storage caching policy. In addition, we have evaluated SHADE against importance sampling with six different caching policies to perform a robust ablation study. For our analysis, an HDD-based NFS Server [63] is used as remote storage. For training, we have used eight NVIDIA P100 GPUs (PCIe with 16 GB memory) spread across four nodes. All the GPU nodes and storage nodes are connected via a 10 Gbps interconnect.

Our experiments primarily use the ImageNet-1K dataset [32], which contains ∼1.2 million images with a total size of 138 GB spanning 1,000 object classes. We also use CIFAR-10 [54]. We conduct our study on four representative computer vision (CV) distributed DL models, namely, Alexnet [55], ResNet-18, ResNet-50 [38], and VggNet [76]. The setup we use to evaluate our system is representative of CV distributed DL training, which has been used to evaluate prior research works [20, 56, 69, 84]. In the following, the reported percentages for cache size indicate the portion of the dataset that had been cached.

6.2 Cache Hit Ratio

In this set of experiments, we evaluate the performance of our APP caching policy against several other policies. This will help explain the extent to which SHADE is able to make better utilization of the limited cache space. The baseline uses PyTorch's default random sampling and LRU caching policy for eviction. We also implement and evaluate the offline Belady's MIN policy. In addition, we evaluate six SHADE policy variants based on importance sampling:

1. Priority-based LFU policy (sh_pqlfu), which evicts the samples with the least importance score based on a hybrid priority that combines the batch-forward loss (i.e., coarse-grained score) and sample access frequency. If the forward loss is the same, then eviction decision is made based on the access frequency of samples.
2. Priority-based policy (sh_pq), which uses the batch-based, goarse-grained forward loss as the importance score.
3. LRU (sh_lru), which uses the coarse-grained forward loss but evicts samples based on the recency of the items and not the importance scores.
4. LFU (sh_lfu), which uses the coarse-grained forward loss but evicts the least-frequently-used sample.
5. Random (sh_rand), which performs random evictions.
6. APP (sh_app), which makes eviction decisions using our APP policy but does not use loss decomposition, rank-based importance score, and *PADS* sampling.

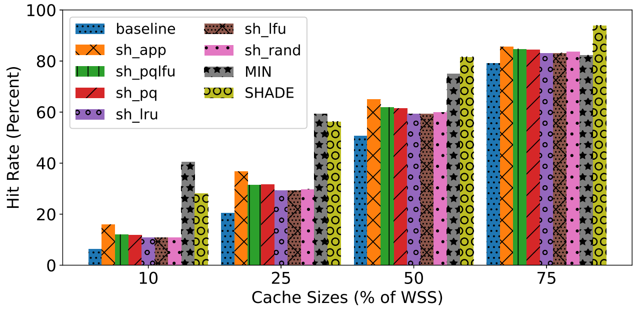


Figure 7: Comparison of the read hit ratio of various caching policies and cache sizes. The sh_ prefix denotes a baseline version of SHADE that uses the coarse-grained importance. SHADE denotes our contribution, SHADE, with all techniques enabled. WSS denotes working set size.

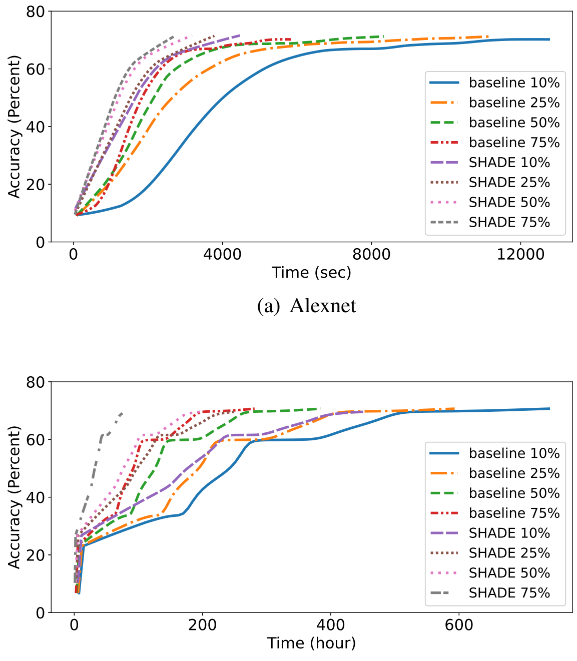
Figure 7 shows the the average read hit ratios when training the three DL models (Alexnet, ResNet-50, VggNet) over the CIFAR-10 dataset under the studied policies with different cache sizes.

We observe that the margin by which SHADE performs better than policy 1–6 increases as the cache size becomes smaller. When only 10% of the WSS is cached, SHADE, with all techniques incorporated, shows a 4.5× higher read hit rate than the baseline LRU. SHADE without the importance derivation techniques (sh_app) can achieve 2.67× and SHADE without the APP cache policy (all the SHADE sh_ baselines except sh_app) can achieve around 1.94× higher hit rates than the baseline. The reason for the improved hit rate is that our techniques are able to predict which samples the training processes would need in the future for better accuracy, and hence it approaches the hit rates of the optimal MIN and even outperforms MIN in some cases (WSS 50% and 75%). This is because MIN's knowledge about samples' future access pattern relies on the sampler. However, SHADE manipulates the sampler to create the desired future sample access pattern, which will benefit the DLT job the most in terms of both training duration and accuracy. This sample access pattern comprising multiple hard-to-learn samples enables precise I/O prediction and maximizes the likelihood of a sample being reused in the cache in the future. By ensuring a higher hit rate with limited available cache space, SHADE holds effectively more data in the limited cache space, therefore achieving higher DLT efficiency.

6.3 Accuracy vs. Time

In our next tests, we evaluate how model accuracy changes over time for SHADE when compared to the baseline at different WSS. This shows how quickly SHADE is able to train a model and increase the accuracy using a very small cache even when the baseline has the advantage of using more cache space than SHADE.

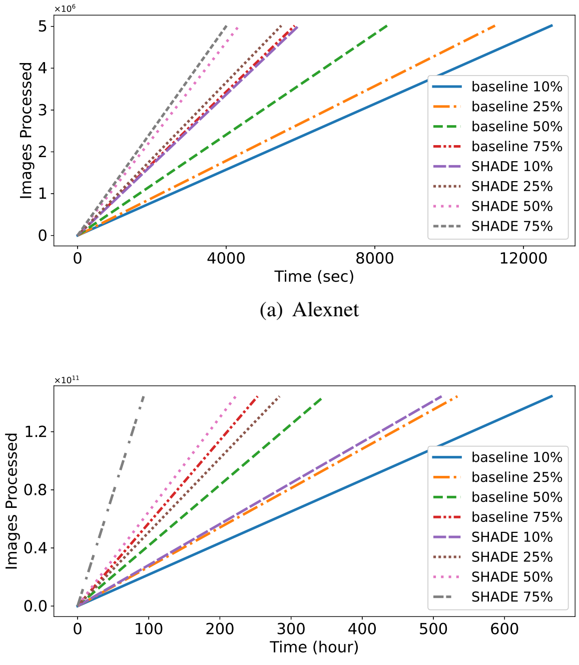
Figure 8 shows that SHADE has a better accuracy improvement rate compared to baseline policy. For example, SHADE can achieve up to 3× faster accuracy convergence compared



(a) Alexnet

(b) ResNet-50

Figure 8: Accuracy improvement rate of SHADE against baseline LRU when different portions of the entire dataset is cached (denoted by the percentages).



(a) Alexnet

(b) ResNet-50

Figure 9: Throughput of SHADE against baseline LRU when different portions of the entire dataset is cached (denoted by the percentages).

to baseline storing 10% of the dataset in the cache when being trained on the Alexnet model. Fine-grained relative importance of samples helps SHADE detect the most important, i.e., hard-to-learn samples, train more on them and thus improve the accuracy quickly to reach convergence. Again when being trained on the ResNet-50 model, SHADE continuously maintains a better accuracy improvement rate compared to baseline at similar WSS. SHADE can reach accuracy convergence 3.3× faster compared to baseline at 75% WSS. The accuracy improvement rate of the baseline with a larger cache is not always better in Figure 8(a) because the baseline uses random sampling. Random sampling places equal emphasis on all of the samples and hence cannot improve the accuracy quickly by training more on the hard-to-learn samples. The improvement in accuracy vs. time curve for the baseline comes only from caching more data. Hence, our results show that even if a larger portion of the dataset is cached, naively caching data items without proper techniques to exploit data locality can not guarantee improved performance.

6.4 Throughput

In our next experiment, we evaluate the throughput of SHADE, which help demonstrate the superiority of SHADE in processing data while storing a limited portion of the dataset. Figure 9(a) shows that SHADE while caching just 10% of the dataset has around 2.3× better throughput compared to baseline policy caching 10% of the dataset. The baseline matches the throughput performance of SHADE only when the baseline is caching 7.5× more data compared to SHADE. In the experiment with ResNet-50, shown in Figure 9(b), we observe that SHADE at 75% WSS has 2.7× higher throughput compared to baseline at similar WSS. Even at lower WSS, SHADE can achieve higher throughput compared to baseline at higher WSS. For example, SHADE at 50% WSS has a 1.14× higher throughput than baseline at 75% WSS. The improvement in the ability to process more images is due to the ability of SHADE to exploit data locality with APP cache policy. Although baseline at 75% WSS has a slightly higher throughput compared to SHADE at 25% WSS, it is unable to get a better accuracy improvement rate seen in Figure 8(b). This is because SHADE can exploit data locality and has techniques to train on important samples which ensures a better accuracy improvement rate.

6.5 Minibatch Load Time

In our next test, we evaluate the performance gain observed in minibatch load time. Consistency in minibatch load time is important so that all the training processes can remain coordinated. It also shows the effectiveness that a caching policy has in exploiting data locality. Figure 10 shows the average minibatch load time of the GPUs during the course of training with the vertical lines showing the standard deviation of the load time within a single epoch.

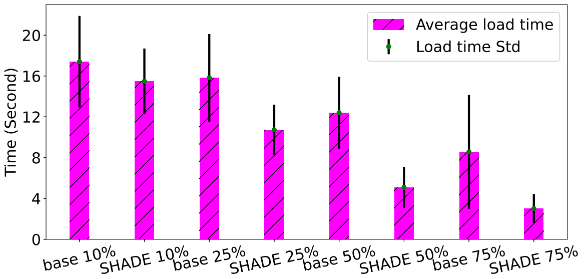


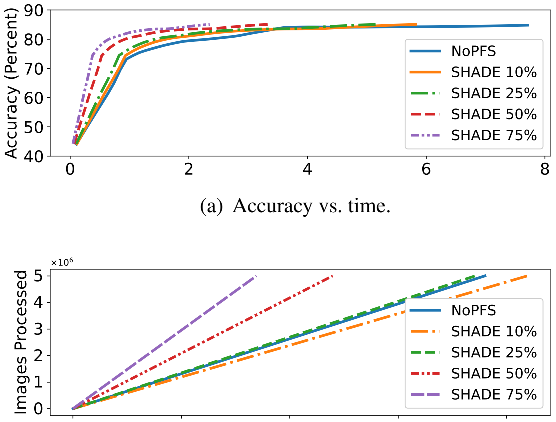
Figure 10: GPUs' minibatch load time when training ResNet-50. Percentages denote the amount of cached dataset.

As we can see in the figure, SHADE can achieve a lower mean load time compared to baseline at similar and higher WSS. The baseline at 50% and 75% WSS has 2.5× and 1.7× higher minibatch load time compared to SHADE at 50% WSS. Moreover, SHADE can maintain a small standard deviation in minibatch load time. Ideally, we would expect the standard deviation in minibatch load times to be low if larger portions of the dataset get cached because of using a higher portion of the fast RAM storage. However, it is not the case for baseline, even if it caches larger portions of the dataset in the cache. Average minibatch load time is highly variant for baseline caching 50% and 75% of the dataset. The baseline at 75% WSS has a 3.9× higher standard deviation in minibatch load time compared to SHADE at 75%.

6.6 End-to-End System Comparison

In our last set of experiments, we compare the performance of SHADE against NoPFS [34], a state-of-the-art storage system for improving the I/Os of DLT workloads. NoPFS exploits the seeds that generate the random access pattern when performing SGD-based DLT to predict when and where a training sample will be accessed. Similar to our baseline, NoPFS uses random sampling of indices. The difference lies in that NoPFS does not consider importance and uses *Clairvoyance* (i.e., seeds that generate random access patterns) to approximate "future distances" of Belady's MIN [18]. SHADE considers fine-grained importance of samples and uses *PADS* policy to prioritize samples for training.

For fair comparison, we keep the experimental setup and training parameters the same for both SHADE and NoPFS while training on the CIFAR-10 dataset. Figure 11 shows that NoPFS incurs a 4.5× and 2.4× increase in training time to reach accuracy convergence compared to SHADE at 75% and 50% WSS, respectively. At the same time, SHADE has 2.2× and 1.6× better throughput compared to NoPFS when working at 75% and 50% WSS, respectively. SHADE can still attain accuracy convergence faster even at 10% WSS. SHADE performs better than NoPFS as it adopts a prefetching system that aims to approximate Belady's MIN; it is bounded by the sample access pattern provided by the sampling policy and hence prioritizes all samples equally. As a result, it takes more time to reach accuracy convergence compared to SHADE, which trains more on hard-to-learn samples to increase the accuracy improvement rate faster.



(a) Accuracy vs. time.

(b) Throughput.

Figure 11: Comparison of SHADE and NoPFS [34]. Percentage denotes the percentage of cached dataset.

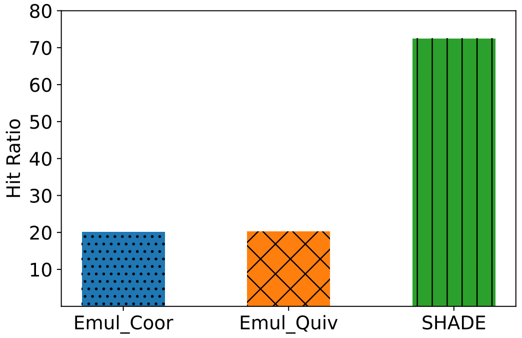


Figure 12: Comparison of the read hit ratio of different caching policies at 20% WSS of CIFAR-10.

We further compare the cache hit ratios of SHADE with state-of-the-art DLT caching policies. we configure a small cache space of 20% of the WSS over the CIFAR-10 dataset using the ResNet-18 model against emulated caching policies including CoorDL [65] and Quiver [56]. To understand the impact of these techniques in importance-aware training, we use a loss-based importance sampling technique [50] inspired by Mercury [84]. For emulating CoorDL and Quiver, we create our own implementations of the core techniques of CoorDL and Quiver, which we name as `Emul_Coor` and `Emul_Quiv`, respectively. Both `Emul_Coor` and `Emul_Quiv` use a KVS as a cache similar to SHADE. `Emul_Coor` ensures that no items are ever evicted from the cache once these are inserted in the cache. In the case of `Emul_Quiv`, we implement the substitutability technique, which replaces a missed sample with a sample already in the cache to avoid memory thrashing.

Figure 12 shows that both `Emul_Coor` and `Emul_Quiv` can only extend their utilization up to the size of the cache

($\sim 20\%$) because these caching policies are not importance-aware and therefore cannot exploit the data locality of the important samples perfectly in importance-aware training. Both of these policies populate the cache using random samples and hence are unable to get a good hit rate by exploiting the repetitions among samples that occur throughout training. On the other hand, as SHADE can manipulate the sampling process (*PADS* policy) and keep repeated samples in the cache, it can achieve a higher hit ratio (72.5%) and thus outperforms both `Emul_Quiv` and `Emul_Coor` by 3.6×.

7 Related Work

Several recent works have explored the use of importance sampling for optimizing the system efficiency of DL workloads [23, 84]. iCACHE [23] is an importance-sampling-informed DLT cache. Although this approach uses a form of fine-grained importance similar to SHADE, it does not have a rank-based relative score scheme and SHADE's *PADS* sampling approach, due to which it may suffer from a lower cache hit ratio than SHADE. Moreover, in case of a cache miss, iCACHE uses substitutability, which may impact the training accuracy convergence.

Mercury [84] improves DL training efficiency by exploiting the important samples. Mercury is not an I/O cache, and therefore, unlike SHADE, it does not handle data replacement and eviction.

CoorDL [65] analyzes the data retrieval process in PyTorch and proposes a MinIO cache, which populates the cache with a random set of data items from the first epoch, and keeps these items in the cache during the training with no item being ever evicted. However, as shown in Figure 12, simply caching random samples does not provide the expected performance gain.

A body of work is focused on optimizing the I/O components of DL applications. NoPFS [20] adopts a prefetching approach that uses hardware level configurations to take caching decisions based on a sample access pattern obtained from trying to approximate Belady's MIN. However, in common online training like hyperparameter tuning experiments [60] with different random seeds, such sample access patterns change constantly and hence are not readily available. We address this constant change in sample access pattern through our dynamic cache management policy without depending on hardware configurations for boosting our performance.

Hoard [69], Quiver [56], and FanStore [86] explore the idea of adding a global caching layer to the GPU cluster for improving the training performance of DL workloads. DeepIO [88] proposes an entropy-aware mechanism for determining next minibatches but it does not offer any cache eviction policies and suffers from lack of dataset randomization. DIESEL [80] is a comprehensive storage solution that supports key-value-based metadata service, task-level caching, and chunk-based shuffling. However, these works do not focus on how to enable fundamental data locality for DLT jobs. SHADE, on the other hand, exploits importance sampling to enable data locality for DLT jobs.

8 Conclusion

The I/O pipeline is a major bottleneck in distributed DLT when data is read from a remote storage. To address this bottleneck, ad hoc solutions such as using faster local storage devices (e.g., SSDs) had been employed. However, those ad hoc solutions cannot fundamentally address the I/O efficiency of DLT workloads. Although caching is possible for DLT, naively caching redundant samples does not provide any benefits. SHADE realizes a DLT-aware caching policy, which takes advantage of the fine-grained importance scores of data samples in order to enable a high level of data locality, and therefore, fundamental cacheability for DLT jobs. Our evaluation demonstrates that SHADE improves the read hit ratio of a small memory cache (of only 10% of the WSS of the dataset) by up to 4.5× compared to traditional, non-DLT-aware caching policies, thus significantly improving the DLT performance.

9 Acknowledgments

We thank our shepherd, Raju Rangaswami, and our anonymous reviewers for their detailed feedback and valuable suggestions. This work is sponsored in part by the NSF under the grants: CSR-2106634, CCF-1919113/1919075, CNS-2045680, OAC-2004751, and OAC-2106446. Results presented in this paper were obtained using the Chameleon testbed supported by the NSF.

References

[1] Amazon EC2 Spot Instances. Run Fault Tolerant workloads for up to 90% off. https://aws.amazon.com/ec2/spot/.

[2] Amazon S3. https://aws.amazon.com/pm/serv-s3/.

[3] CSCS. 2021. Piz Daint. https://www.cscs.ch/computers/piz-daint/.

[4] Global Conversational AI Market Report 2021. https://tinyurl.com/Global-AI-Market-Report-2021.

[5] Google Cloud Storage. https://cloud.google.com/storage.

[6] Memcached. https://memcached.org/.

[7] Redis. https://redis.io/.

[8] RIKEN Center for Computational Science. 2021. About Fugaku. https://www.r-ccs.riken.jp/en/fugaku/about/.

[9] Use Spot VMs with Batch. https://learn.microsoft.com/en-us/azure/batch/batch-spot-vms.

[10] Franklin Abodo, Robert Rittmuller, Brian Sumner, and Andrew Berthaume. Detecting work zones in shrp 2 nds videos using deep learning based computer vision. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 679–686. IEEE, 2018.

[11] Guillaume Alain, Alex Lamb, Chinnadhurai Sankar, Aaron Courville, and Yoshua Bengio. Variance reduction in sgd by distributed importance sampling. *arXiv preprint arXiv:1511.06481*, 2015.

[12] Muhammad Aqib, Rashid Mehmood, Aiiad Albeshri, and Ahmed Alzahrani. Disaster management in smart cities by forecasting traffic plan using deep learning and gpus. In *International Conference on Smart Cities, Infrastructure, Technologies and Applications*, pages 139–154. Springer, 2017.

[13] Jorge F Arinez, Qing Chang, Robert X Gao, Chengying Xu, and Jianjing Zhang. Artificial intelligence in advanced manufacturing: Current status and future outlook. *Journal of Manufacturing Science and Engineering*, 142(11), 2020.

[14] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W Lee. Flashneuron: Ssd-enabled large-batch training of very deep neural networks. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, pages 387–401, 2021.

[15] Stephen Balaban. Deep learning and face recognition: the state of the art. *Biometric and surveillance technology for human and activity identification XII*, 9457:68–75, 2015.

[16] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):1–9, 2014.

[17] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. Preemptive all-reduce scheduling for expediting distributed dnn training. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 626–635. IEEE, 2020.

[18] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[19] Sweta Bhattacharya, Siva Rama Krishnan Somayaji, Thippa Reddy Gadekallu, Mamoun Alazab, and Praveen Kumar Reddy Maddikunta. A review on deep learning for future smart cities. *Internet Technology Letters*, page e187, 2020.

[20] Roman Böhringer, Nikoli Dryden, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning i/o. *arXiv preprint arXiv:2101.08734*, 2021.

[21] Peter Braam. The lustre storage architecture. *arXiv preprint arXiv:1903.01955*, 2019.

[22] CL Philip Chen and Chun-Yang Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information sciences*, 275:314–347, 2014.

[23] Weijian Chen, Shuibing He, Yaowen Xu, Xuechen Zhang, Siling Yang, Shuang Hu, Sun Xian-He, and Gang Chen. icache: An importance-sampling-informed cache for accelerating i/o-bound dnn model training. In *2023 IEEE International Symposium on High-Performance Computer Architecture*, 2023.

[24] Xue-Wen Chen and Xiaotong Lin. Big data deep learning: challenges and perspectives. *IEEE access*, 2:514–525, 2014.

[25] Eunsuk Chong, Chulwoo Han, and Frank C Park. Deep learning networks for stock market analysis and prediction: Methodology, data representations, and case studies. *Expert Systems with Applications*, 83:187–205, 2017.

[26] Joon Yee Chuah. Machine learning gpu power measurement on chameleon cloud. In *Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 181–181, 2017.

[27] Joaquin Chung, Zhengchun Liu, Rajkumar Kettimuthu, and Ian Foster. Elastic data transfer infrastructure (dti) on the chameleon cloud. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–2. IEEE, 2019.

[28] Angel Cruz-Roa, Hannah Gilmore, Ajay Basavanhally, Michael Feldman, Shridar Ganesan, Natalie NC Shih, John Tomaszewski, Fabio A González, and Anant Madabhushi. Accurate and reproducible invasive breast cancer detection in whole-slide images: A deep learning approach for quantifying tumor extent. *Scientific reports*, 7:46450, 2017.

[29] Padideh Danaee, Reza Ghaeini, and David A Hendrix. A deep learning approach for cancer detection and relevant gene identification. In *PACIFIC SYMPOSIUM ON BIOCOMPUTING 2017*, pages 219–229. World Scientific, 2017.

[30] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.

[32] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[33] Xiao Ding, Yue Zhang, Ting Liu, and Junwen Duan. Deep learning for event-driven stock prediction. In *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[34] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning i/o. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[35] J Fombellida, I Martín-Rubio, S Torres-Alegre, and D Andina. Tackling business intelligence with bioinspired deep learning. *Neural Computing and Applications*, pages 1–8, 2018.

[36] Google. YouTube-8M Segments Dataset. https://research.google.com/youtube8m/.

[37] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[39] James B Heaton, Nick G Polson, and Jan Hendrik Witte. Deep learning for finance: deep portfolios. *Applied Stochastic Models in Business and Industry*, 33(1):3–12, 2017.

[40] Jan Heichler. An introduction to beegfs, 2014.

[41] M Shamim Hossain and Ghulam Muhammad. Environment classification for urban big data using deep learning. *IEEE Communications Magazine*, 56(11):44–50, 2018.

[42] Jeremy Howard and Sylvain Gugger. Fastai: a layered api for deep learning. *Information*, 11(2):108, 2020.

[43] Zilong Hu, Jinshan Tang, Ziming Wang, Kai Zhang, Ling Zhang, and Qingling Sun. Deep learning for image-based cancer detection and diagnosis- a survey. *Pattern Recognition*, 83:134–149, 2018.

[44] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[45] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

[46] Tyler B Johnson and Carlos Guestrin. Training deep models faster with robust, approximate importance sampling. *Advances in Neural Information Processing Systems*, 31:7265–7275, 2018.

[47] Brigitte Juanals and Jean-Luc Minel. Categorizing air quality information flow on twitter using deep learning tools. In *International Conference on Computational Collective Intelligence*, pages 109–118. Springer, 2018.

[48] Vanessa Isabell Jurtz, Alexander Rosenberg Johansen, Morten Nielsen, Jose Juan Almagro Armenteros, Henrik Nielsen, Casper Kaae Sønderby, Ole Winther, and Søren Kaae Sønderby. An introduction to deep learning on biological sequence data: examples and solutions. *Bioinformatics*, 33(22):3685–3690, 2017.

[49] Jaret M Karnuta, Michael P Murphy, Bryan C Luu, Michael J Ryan, Heather S Haeberle, Nicholas M Brown, Richard Iorio, Antonia F Chen, and Prem N Ramkumar. Artificial intelligence for automated implant identification in total hip arthroplasty: A multicenter external validation study exceeding two million plain radiographs. *The Journal of Arthroplasty*, 2022.

[50] Angelos Katharopoulos and Francois Fleuret. Not all samples are created equal: Deep learning with importance sampling. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2525–2534. PMLR, 10–15 Jul 2018.

[51] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[52] Kate Keahey, Pierre Riteau, Dan Stanzione, Tim Cockerill, Joe Mambretti, Paul Rad, and Paul Ruth. Chameleon: a scalable production testbed for computer science research. In *Contemporary High Performance Computing*, pages 123–148. CRC Press, 2019.

[53] Mohammad Khan, Didrik Nielsen, Voot Tangkaratt, Wu Lin, Yarin Gal, and Akash Srivastava. Fast and scalable Bayesian deep learning by weight-perturbation in Adam. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2611–2620. PMLR, 10–15 Jul 2018.

[54] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[55] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[56] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 283–296, 2020.

[57] Quoc V Le, Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, and Andrew Y Ng. On optimization methods for deep learning. In *ICML*, 2011.

[58] He Li, Kaoru Ota, and Mianxiong Dong. Learning iot in edge: Deep learning for the internet of things with edge computing. *IEEE network*, 32(1):96–101, 2018.

[59] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[60] Lizhi Liao, Heng Li, Weiyi Shang, and Lei Ma. An empirical study of the impact of hyperparameter tuning and model optimization on the performance properties of deep neural networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–40, 2022.

[61] Ilya Loshchilov and Frank Hutter. Online batch selection for faster training of neural networks. *arXiv preprint arXiv:1511.06343*, 2015.

[62] Nimrod Megiddo and Dharmendra S Modha. {ARC}: A {Self-Tuning}, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, 2003.

[63] Sun Microsystems. Rfc1094: Nfs: Network file system protocol specification, 1989.

[64] Riccardo Miotto, Fei Wang, Shuang Wang, Xiaoqian Jiang, and Joel T Dudley. Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics*, 19(6):1236–1246, 2018.

[65] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *arXiv preprint arXiv:2007.06775*, 2020.

[66] NVIDIA Corporation. nvidia-smi - NVIDIA System Management Interface program. `http://manpages.ubuntu.com/manpages/precise/man1/alt-nvidia-current-smi.1.html`.

[67] Yosuke Oyama, Naoya Maruyama, Nikoli Dryden, Erin McCarthy, Peter Harrington, Jan Balewski, Satoshi Matsuoka, Peter Nugent, and Brian Van Essen. The case for strong scaling in deep learning: Training large 3d cnns with hybrid parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1641–1652, 2020.

[68] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[69] Christian Pinto, Yiannis Gkoufas, Andrea Reale, Seetharami Seelam, and Steven Eliuk. Hoard: A distributed data caching system to accelerate deep learning training on the cloud. *arXiv preprint arXiv:1812.00669*, 2018.

[70] Sarunya Pumma, Min Si, Wu-Chun Feng, and Pavan Balaji. Scalable deep learning via i/o analysis and optimization. *ACM Transactions on Parallel Computing (TOPC)*, 6(2):1–34, 2019.

[71] Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880, 2009.

[72] Viktor Rausch, Andreas Hansen, Eugen Solowjow, Chang Liu, Edwin Kreuzer, and J Karl Hedrick. Learning a deep neural net policy for end-to-end control of autonomous vehicles. In *2017 American Control Conference (ACC)*, pages 4914–4919. IEEE, 2017.

[73] Sebastien Godard. sar(1) - Linux Man Page, 2021. `https://linux.die.net/man/1/sar`.

[74] Shaohuai Shi, Qiang Wang, Xiaowen Chu, and Bo Li. A dag model of synchronous stochastic gradient descent in distributed deep learning. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 425–432. IEEE, 2018.

[75] Shaohuai Shi, Qiang Wang, Xiaowen Chu, Bo Li, Yang Qin, Ruihao Liu, and Xinxiao Zhao. Communication-efficient distributed deep learning with merged gradient sparsification on gpus. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 406–415. IEEE, 2020.

[76] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[77] Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. Communication-efficient distributed deep learning: A comprehensive survey. *arXiv preprint arXiv:2003.06307*, 2020.

[78] Arnold Tunick. On benchmarking multiple gpu computing resources for faster training of deep neural networks. Technical report, CCDC Army Research Laboratory Adelphi United States, 2020.

[79] Jinjiang Wang, Yulin Ma, Laibin Zhang, Robert X Gao, and Dazhong Wu. Deep learning for smart manufacturing: Methods and applications. *Journal of Manufacturing Systems*, 48:144–156, 2018.

[80] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[81] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *arXiv preprint arXiv:1705.07878*, 2017.

[82] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[83] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[84] Xiao Zeng, Ming Yan, and Mi Zhang. Mercury: Efficient on-device distributed dnn training via stochastic importance sampling. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 29–41, 2021.

[85] Zhao Zhang, Lei Huang, Uri Manor, Linjing Fang, Gabriele Merlo, Craig Michoski, John Cazes, and Niall Gaffney. Fanstore: Enabling efficient and scalable i/o for distributed deep learning, 2018.

[86] Zhao Zhang, Lei Huang, J Gregory Pauloski, and Ian T Foster. Efficient i/o for neural network training with compressed data. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 409–418. IEEE, 2020.

[87] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying cache policies for photo stores at scale: A tencent case study. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 284–294, New York, NY, USA, 2018. Association for Computing Machinery.

[88] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Entropy-aware i/o pipelining for large-scale deep learning on hpc systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 145–156. IEEE, 2018.

Intelligent Resource Scheduling for Co-located Latency-critical Services: A Multi-Model Collaborative Learning Approach

Lei Liu[1]*, Xinglei Dou[2], Yuetao Chen[2]
[1]Beihang University; [2]ICT, CAS; Sys-Inventor Lab

Abstract

Latency-critical services have been widely deployed in cloud environments. For cost-efficiency, multiple services are usually co-located on a server. Thus, run-time resource scheduling becomes the pivot for QoS control in these complicated co-location cases. However, the scheduling exploration space enlarges rapidly with the increasing server resources, making schedulers unable to provide ideal solutions quickly and efficiently. More importantly, we observe that there are "resource cliffs" in the scheduling exploration space. They affect the exploration efficiency and always lead to severe QoS fluctuations in previous schedulers. To address these problems, we propose a novel ML-based intelligent scheduler – OSML. It learns the correlation between architectural hints (e.g., IPC, cache misses, memory footprint, etc.), scheduling solutions and the QoS demands. OSML employs multiple ML models to work collaboratively to predict QoS variations, shepherd the scheduling, and recover from QoS violations in complicated co-location cases. OSML can intelligently avoid resource cliffs during scheduling and reach an optimal solution much faster than previous approaches for co-located applications. Experimental results show that OSML supports higher loads and meets QoS targets with lower scheduling overheads and shorter convergence time than previous studies.

1 Introduction

Cloud applications are shifting from monolithic architectures to loosely-coupled designs, consisting of many latency-critical (LC) services (e.g., some microservices, interactive services) with strict QoS requirements [18,19,52,53]. Many cloud providers, including Amazon, Alibaba, Facebook, Google, and LinkedIn, employ this loosely-coupled design to improve productivity, scalability, functionality, and reliability of their cloud systems [2,3,18,52].

QoS-driven resource scheduling faces more challenges in this era. The cost-efficiency policy drives providers to co-locate as many applications as possible on a server. However, these co-located services exhibit diverse behaviors across the storage hierarchy, including multiple interactive resources such as CPU cores, cache, memory/IO bandwidth, and main memory banks. These behaviors also can be drastically different from demand to demand and change within seconds. Moreover, with the increasing number of cores, more threads contend for the shared LLC (last-level cache) and memory bandwidth. Notably, these shared resources interact with each other [32,33,45,80]. All these issues make resource scheduling for co-located LC services more complicated and time-consuming. Moreover, in reality, end-users keep increasing demands for quick responses from the cloud [15,47,53]. According to Amazon's estimation, even if the end-users experience a 1-second delay, they tend to abort the transactions, translating to $1.6 billion loss annually [1]. Briefly, unprecedented challenges are posed for resource scheduling mechanisms [8,10,31,47,52].

Some previous studies [17,31,33,45,61] design clustering approaches to allocate LLC or LLC together with main memory bandwidth to cores for scheduling single-threaded applications. Yet, they are not suitable in cloud environments, as the cloud services often have many concurrent threads and strict QoS constraints (i.e., latency-critical). Alternatively, the existing representative studies either use heuristic algorithms – increasing/decreasing one resource at a time and observing the performance variations [10] or use learning-based algorithms (e.g., Bayesian optimization [46]) in a relatively straightforward manner. The studies in [10,46] show that scheduling five co-located interactive services to meet certain QoS constraints can take more than 20 seconds on average. Existing schedulers still have room for improvement in scheduling convergence time, intelligence, and how to schedule complicated interactive resources (e.g., parallel computing units and complex memory/storage hierarchies) in a timely fashion. Moreover, the existing schedulers cannot easily avoid "resource cliffs", i.e., decreasing a resource only slightly during scheduling leads to a significant QoS slowdown. To the best of our knowledge, our community has been expecting new directions on developing resource-scheduling mechanisms to handle co-located LC services [16,30,31,45].

To this end, we design OSML, a novel machine learning (ML) based resource scheduler for LC services on large-scale servers. Using ML models significantly improves scheduling exploration efficiency for multiple co-located cloud services and can handle the complicated resource sharing, under/over-provision cases timely. ML has achieved tremendous success in improving speech recognition [54], benefiting image recognition [25], and helping the machine to beat the human champion at Go [13,24,51]. In OSML, we make progress in leveraging ML for resource scheduling, and we make the following contributions.

(1) Investigation in RCliff for Multiple Resources during Scheduling. In the context of cloud environment, we study resource cliff (RCliff, i.e., reducing a resource only slightly leads to a significant QoS slowdown) for computing and cache resources. More importantly, we show that RCliffs commonly

*Corresponding author (PI): lei.liu@zoho.com; liulei2010@buaa.edu.cn.

Table 1: Latency-critical (LC) services, including micro-/interactive services [18,19,52,68,46]. The max load - max RPS - is with the 99th percentile tail latency QoS target [10,18,46,52].

LC service	Domain	RPS (Requests Per Second)
Img-dnn [62]	Image recognition	2000,3000,4000,5000,6000 (Max)
Masstree [62]	Key-value store	3000,3400,3800,4200,4600
Memcached [65]	Key-value store	256k,512k,768k,1024k,1280k
MongoDB [64]	Persistent database	1000,3000,5000,7000,9000
Moses [62]	RT translation	2200,2400,2600,2800,3000
Nginx [66]	Web server	60k,120k,180k,240k,300k
Specjbb [62]	Java middleware	7000,9000,11000,13000,15000
Sphinx [62]	Speech recognition	1,4,8,12,16
Xapian [62]	Online search	3600,4400,5200,6000,6800
Login [68]	Login	300,600,900,1200,1500
Ads [68,52]	Online renting ads	10,100,1000

Table 2: Our platform specification vs. a server in 2010~14 [80].

Conf. / Servers	Our Platform	Server (2010s)
CPU Model	Intel Xeon E5-2697 v4	Intel i7-860
Logical Processor Cores	36 Cores (18 phy. cores)	8 Cores (4 phy. cores)
Processor Speed	2.3GHz	2.8GHz
Main Memory / Channel / BW	256GB, 2400MHz DDR4 / 4 Channels / 76.8GB/s	8GB, 1600MHz DDR3 / 2 Channels / 25.6GB/s
Private L1 & L2 Cache Size	32KB and 256KB	32KB and 256KB
Shared L3 Cache Size	45MB - 20 ways	8MB - 16 ways
Disk	1TB,7200 RPM,HD	500GB,5400 RPM,HD
GPU	NVIDIA GP104 [GTX 1080], 8GB Memory	N/A

exist in many widely used LC services and challenge existing schedulers (Sec.3.3). Furthermore, we show ML can be an ideal approach that benefits scheduling (Sec.4.4).

(2) Collaborative ML Models for Intelligent Scheduling. OSML is an ML-based scheduler that intelligently schedules multiple interactive resources to meet co-located services' QoS targets. OSML learns the correlation between architectural hints (e.g., IPC, cache misses, memory footprint, etc.), optimal scheduling solutions, and the QoS demands. It employs MLP models to avoid RCliffs intelligently, thus avoiding the sudden QoS slowdown often incurred by the RCliffs in prior schedulers; it predicts the QoS variations and resource margins, and then delivers appropriate resource allocations. It leverages an enhanced DQN to shepherd the allocations and recover from the QoS violation and resource over-provision cases. Moreover, as OSML's models are lightweight and their functions are clearly defined, it is easy to locate the problems and debug them.

(3) An Open-sourced Data Set for Low-overhead ML. We have collected the performance traces for widely deployed LC services (in Table 1), covering 62,720,264 resource allocation cases that contain around 2-billion samples (Sec.4). These data have a rich set of information, e.g., the RCliffs for multiple resources; the interactions between workload features and the mainstream architectures. Our models can be trained and generalized with these data and then used on new platforms with low-overhead transfer learning. People can study the data set and train their models without a long period for data collection.

(4) Real Implementation and Detailed Comparisons. We implement OSML based on latest Linux. OSML is designed as a co-worker of the OS scheduler located between the OS kernel and the user layer. We compare OSML with the most related open-source studies [10,46] and show the advantages.

OSML captures the applications' online behaviors, forwards them to the ML models running on CPU or GPU, and schedules resources accordingly. Compared with [10,46], OSML takes 36~55% less time to meet the QoS targets; and supports 10~50% higher loads in 58% cases where Moses, Img-dnn, and Xapian can be co-scheduled in our experiments. Its ML models have low run-time overheads. OSML project and its ecological construction receive support from industry and academia; a version for research will be open-sourced via https://github.com/Sys-Inventor-Lab/AI4System-OSML.

2 Background and Motivation

The cloud environment has a trend towards a new model [3,18,52], in which cloud applications comprise numerous distributed LC services (i.e., micro/interactive services), such as key-value storing, database serving, and business applications serving [18,19]. Table 1 includes some widely used ones, and they form a significant fraction of cloud applications [18]. These services have different features and resource demands.

In terms of the datacenter servers, at present, new servers can have an increased number of cores, larger LLC capacity, larger main memory capacity, higher bandwidth, and the resource scheduling exploration space becomes much larger than ever before as a result. Table 2 compares the two typical servers used at different times. On the one hand, although modern servers can have more cores and memory resources than ever before, they are not fully exploited in today's cloud environments. For instance, in Google's datacenter, the CPU utilization is about 45~53% and memory utilization ranges from 25~77% during 25 days, while Alibaba's cluster exhibits a lower and unstable trend, i.e., 18~40% for CPU and 42~60% for memory in 12 hours [32,49], indicating that a lot of resources are wasted. On the other hand, the larger resource scheduling exploration space, which consists of more diverse resources, prohibits the schedulers from achieving the optimal solution quickly. Additionally, cloud applications can have dozens of concurrent threads [10,46]. When several cloud applications run on a server, they share and contend resources across multiple resource layers – cores, LLC, memory bandwidth/banks. Previous studies show they may incur severe performance degradation and unpredictable QoS violations, and propose the scheduling approaches at architecture [9,23,44], OS [31,45,50,80], and user-level [10,37,38]. *Yet, do they perform ideally for scheduling co-located LC services on modern datacenter servers?*

3 Resource Scheduling for LC Services

To answer the above question, we study the LC services (Table 1) that are widely deployed in cloud environments.

3.1 Understanding the LC Services - Resource Cliff

We study how sensitive these LC services are to the critical resources, e.g., the number of cores and LLC capacity,

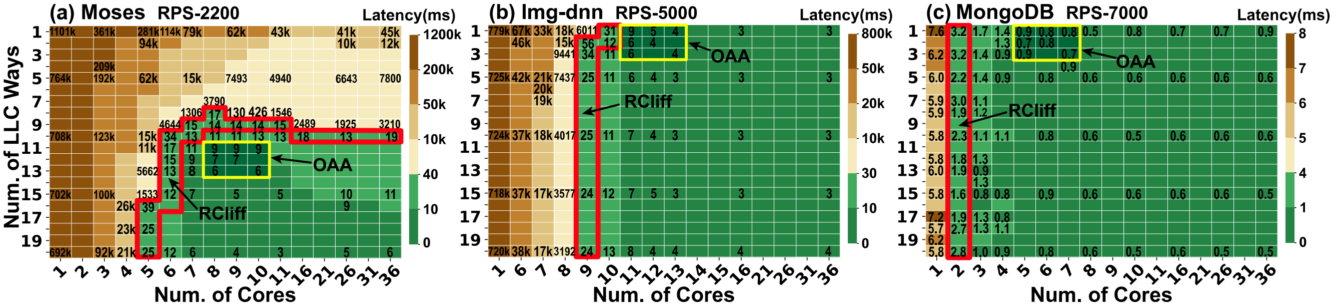


Figure 1: The resource scheduling exploration space for cores and LLC ways. All services here are with 36 threads. These figures show the sensitivity to resource allocation under different policies. Each col./row represents a specific number of LLC ways/cores allocated to an application. Each cell denotes the LC service's response latency under the given number of cores and LLC ways. The Redline highlights the RCliff (can be obtained by selecting the knee solution [69]). The green color cells show allocation policies that bring better performance (low response latency). OAA (Optimal Allocation Area) is also illustrated for each LC service. We test all of the LC services in Table 1, and we find the RCliff and OAA are always existing, though the RPS varies [79]. We only show several of them for the sake of saving space.

on a commercial platform ("our platform" in Table 2). For Moses, as illustrated in Figure 1-a, with the increasing number of cores, more threads are mapped on them simultaneously. Meanwhile, for a specific amount of cores, more LLC ways can benefit performance. Thus, we observe the response latency is low when computing and LLC resources are ample (i.e., below 10ms in the area within green color). The overall trends are also observed from other LC services.

However, we observe the Cliff phenomenon for these services. In Figure 1-a, in the cases where 6 cores are allocated to Moses, the response latency is increased significantly from 34ms to 4644ms if merely one LLC way is reduced (i.e., from 10 ways to 9 ways). Similar phenomena also happen in cases where computing resources are reduced. As slight resource re-allocations bring a significant performance slowdown, we denote this phenomenon as Resource Cliff (**RCliff**). It is defined as the resource allocation cases that could incur the most significant performance slowdown if resources (e.g., core, cache) are deprived via a fine-grain way in the scheduling exploration space. Take Moses as an example, on the RCliff (denoted by the red box in Figure 1-a), there would be a sharp performance slowdown if only one core or one LLC way (or both) is deprived. Figure 1-b and c show RCliffs for Img-dnn and MongoDB, respectively. They exhibit computing-sensitive features and have RCliff only for cores. From another angle, RCliff means that a little bit more resources will bring significant performance improvement. Figure 1-a shows that Moses exhibits RCliff for both core and LLC. Moreover, we test the services in Table 1 across various RPS and find the RCliffs always exist, though the RCliffs vary (8.8% on average) according to different RPS.

The underlying reason for the cache cliff is locality; for the core cliff, the fundamental reason is on queuing theory - the latency will increase drastically when the request arrival rate exceeds the available cores. RCliff alerts the scheduler not to allocate resources close to it because it is "dangerous to fall off the cliff" and incurs a significant performance slowdown, i.e., even a slight resource reduction can incur a severe slowdown. Notably, in Figure 1, we highlight each LC service's **Optimal Allocation Area (OAA)** in the scheduling exploration space, defined as the ideal number of allocated cores and LLC ways to bring an acceptable QoS. More resources than OAA cannot deliver more significant performance, but fewer resources lead to the danger of falling off the RCliff. OAA is the goal that schedulers should achieve.

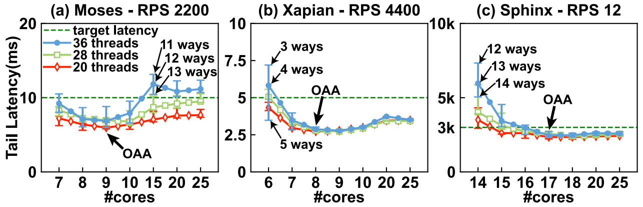


Figure 2: OAA exists regardless of the num. of concurrent threads.

3.2 Is OAA Sensitive to the Number of Threads?

In practice, an LC service may have many threads for higher performance. Therefore, we come up with the question: *is the OAA sensitive to the number of threads, i.e., if someone starts more threads, will the OAA change?*

To answer this question, for a specific LC service, we start a different number of threads and map them across a different number of cores (the num. of threads can be larger than the num. of cores). Through the experiments, we observe - (i) More threads do not necessarily bring more benefits. Take Moses as an example, when more threads are started (e.g., 20/28/36) and mapped across a different number of cores, the overall response latency can be higher (as illustrated in Figure 2). The underlying reason lies in more memory contentions at memory hierarchy and more context switch overheads, leading to a higher response latency [20,36]. (ii) The OAA is not sensitive to the number of concurrent threads. For Moses in Figure 2, even if 20/28/36 threads are mapped to 10~25 cores, around 8/9-core cases always perform ideally. Other LC services in Table 1 also show a similar phenomenon, though their OAAs are different for different applications.

In practice, if the QoS for a specific LC service is satisfied, LLC ways should be allocated as less as possible, saving LLC space for other applications. Similarly, we also try to allocate fewer cores for saving computing resources. Here, we conclude that the OAA is not sensitive to the number

Table 3: The input parameters for ML models.

Feature	Description	Models
IPC	Instructions per clock	A/A’/B/B’/C
Cache Misses	LLC misses per second	A/A’/B/B’/C
MBL	Local memory bandwidth	A/A’/B/B’/C
CPU Usage	The sum of each core’s utilization	A/A’/B/B’/C
Virt. Memory	Virtual memory in use by an app	A/A’/B/B’
Res. Memory	Resident memory in use by an app	A/A’/B/B’
Allocated Cores	The number of allocated cores	A/A’/B/B’/C
Allocated Cache	The capacity of allocated cache	A/A’/B/B’/C
Core Frequency	Core Frequency during run time	A/A’/B/B’/C
QoS Slowdown	Percentage of QoS slowdown	B
Expected Cores	Expected cores after deprivation	B’
Expected Cache	Expected cache after deprivation	B’
Cores used by N.	Cores used by Neighbors	A’/B/B’
Cache used by N.	Cache capacity used by Neighbors	A’/B/B’
MBL used by N.	Memory BW used by Neighbors	A’/B/B’
Resp. Latency	Average latency of a LC service	C

of threads. We should further reveal: *how do the existing schedulers perform in front of OAAs and RCliffs?*

3.3 Issues the Existing Schedulers May Meet

We find three main shortcomings in the existing schedulers when dealing with OAAs and RCliffs. **(1) Entangling with RCliffs.** Many schedulers often employ heuristic scheduling algorithms, i.e., they increase/reduce resources until the monitor alerts that the system performance is suffering a significant change (e.g., a severe slowdown). Yet, these approaches could incur unpredictable latency spiking. For example, if the current resource allocation for an LC service is in the base of RCliff (i.e., the yellow color area in Figure 1-a/b/c), the scheduler has to try to achieve OAA. However, as the scheduler doesn’t know the “location” of OAA in the exploration space, it has to increase resources step by step in a fine-grain way, thus the entire scheduling process from the base of the RCliff will incur very high response latency. For another example, if the current resource allocation is on the RCliff or close to RCliff, a slight resource reduction for any purpose could incur a sudden and sharp performance drop for LC services. The previous efforts [10,32,50,53] find there would be about hundreds/thousands of times latency jitter, indicating the QoS cannot be guaranteed during these periods. Thus, RCliffs should not be neglected when designing a scheduler. **(2) Unable to accurately and simultaneously schedule a combination of multiple interactive resources (e.g., cores, LLC ways) to achieve OAAs in low overheads.** Prior studies [10,31,32,45] show that the core computing ability, cache hierarchy, and memory bandwidth are interactive factors for resource scheduling. Solely considering a single dimension in scheduling often leads to sub-optimal QoS. However, the existing schedulers using heuristic or model-based algorithms usually schedule one dimension resource at a time and bring high overheads on scheduling multiple interactive resources. For example, PARTIES [10] takes around 20~30 seconds on average (up to 60 seconds in the worst cases) to find ideal allocations when 3~6 LC services are co-running. The efforts in [16,41,42] also show the heuristics inefficiency due to the high overheads on scheduling various resources with complex configurations. **(3) Unable to provide accurate QoS predictions.** Therefore, the scheduler can hardly balance the global QoS and resource allocations across all co-located applications, leading to QoS violations or resource over-provision.

An ideal scheduler should avoid the RCliff and quickly achieve the OAA from any positions in the scheduling space. We claim it is time to design a new scheduler, and using ML can be a good approach to handle such complicated cases with low overheads.

4 Leveraging ML for Scheduling

We design a new resource scheduler - OSML. It differs from the previous studies in the following ways. We divide the resource scheduling for co-located services into several routines and design ML models to handle them, respectively. These models work collaboratively in OSML to perform scheduling. OSML uses data-driven static ML models (Model-A/B) to predict the OAA/RCliff, and balances the QoS and resource allocations among co-located LC services, and uses the reinforcement learning model (Model-C) to shepherd the allocations dynamically. Moreover, we collect extensive real traces for widely deployed LC services.

4.1 Model-A: Aiming OAA

Model-A Description. The neural network used in Model-A is a 3-layer multi-layer perceptron (MLP); each layer is a set of nonlinear functions of a weighted sum of all outputs that are fully connected from the prior one [21,24]. There are 40 neurons in each hidden layer. There is a dropout layer with a loss rate of 30% behind each fully connected layer to prevent overfitting. For each LC service, the **inputs** of the MLP include 9 items in Table 3 and the **outputs** include the OAA for multiple interactive resources, OAA bandwidth (bandwidth requirement for OAA), and the RCliff. Model-A has a shadow – A’, which has the same MLP structure and 12 input parameters (Table 3), providing solutions when multiple LC services are running together.

Model-A Training. Collecting training data is an offline job. We have collected the performance traces that involve the parameters in Table 3 for the LC services in Table 1 on “our platform” in Table 2. The parameters are normalized into [0,1] according to the function: $\text{Normalized\_Feature} = (\text{Feature} - \text{Min})/(\text{Max} - \text{Min})$. Feature is the original value; Max and Min are predefined according to different metrics.

For each LC service with every common RPS demand, we sweep 36 threads to 1 thread across LLC allocation policies ranging from 1 to 20 ways and map the threads on a certain number of cores and collect the performance trace data accordingly. In each case, we label the corresponding OAA, RCliff and OAA bandwidth. For example, Figure 3 shows a data collection case where 8 threads are mapped onto 7 cores with 4 LLC ways. We feed the LC services with diverse RPS (Table 1), covering most of the common cases. Moreover, to train Model-A’s shadow (A’), we map LC services on the remaining resources in the above process and get the traces for co-location cases. Note that the resources

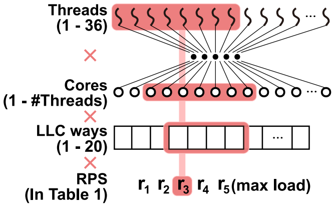


Figure 3: Model-A data collection.

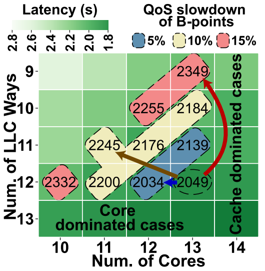


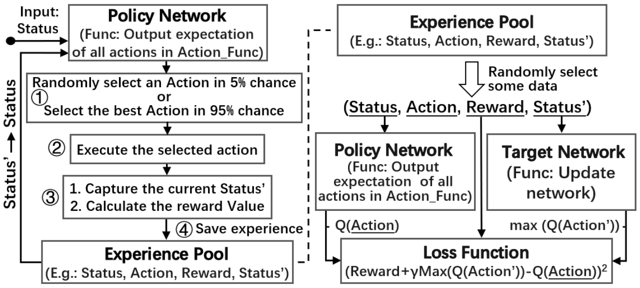
Figure 4: Model-B training.

are partitioned among applications. We test comprehensive co-location cases for LC services in Table 1, and find the LC services' RCliffs vary from 2.8% to 38.9%, and OAAs vary from 5.6% to 36.1%, respectively, when multiple LC services are co-running. Finally, we collect 43,299,135 samples (data tuples), covering 1,308,296 allocation cases with different numbers of cores, LLC ways, and bandwidth allocations. A large amount of traces may lead to higher accuracy for ML models. The workload characteristics are converted to traces consisting of hardware parameters used for fitting and training MLP to provide predictions.

4.2 Model-B: Balancing QoS and Resources

Model-B Description. Model-B employs an MLP with the same structure in Model-A' plus one more input item, i.e., QoS slowdown (Table 3). Model-B **outputs** the resources that a service can be deprived of under allowable QoS slowdown. As the computing units and memory resource can be fungible [10], Model-B's outputs include three policies, i.e., <cores, LLC ways>, <cores dominated, LLC ways> and <cores, LLC ways dominated>, respectively. The tuple items are the number of cores, LLC ways deprived and reallocated to others with the allowable QoS slowdown. The term "cores dominated" indicates the policy using more cores to trade the LLC ways, and vice versa. The allowable QoS slowdown is determined according to the user requirement or the LC services' priority and controlled by the OSML's central logic. We denote Model-B's outputs as B-Points.

Model-B trades QoS for resources. For example, when an LC service (E) comes to a server that already has 4 co-located services, OSML enables Model-A' to obtain <n+, m+>, which denotes at least n more cores and m more LLC ways should be provided to meet E's QoS. Then, OSML enables Model-B and uses the allowable QoS slowdown as an input to infer B-Points for obtaining resources from other co-located services. B-Points include the "can be deprived" resources from E's neighbors with the allowable QoS slowdown. Finally, OSML finds the best solution to match <n+, m+> with B-Points, which has a minimal impact on the co-located applications' current allocation state. Detailed logic is in Algo._1. Besides, we design Model-B' (a shadow of Model-B) to predict how much QoS slowdown will suffer if a certain amount of resources is deprived of a specific service. Model-B' has an MLP with the same structure in Model-A' plus the items that indicate the remaining cache ways and cores after deprivation.



Input: Status
Policy Network
(Func: Output expectation of all actions in Action_Func)
Randomly select an Action in 5% chance
①
or
Select the best Action in 95% chance
② Execute the selected action
③
1. Capture the current Status'
2. Calculate the reward Value
④ Save experience
Experience Pool
(E.g.: Status, Action, Reward, Status')
Status' → Status
Experience Pool
(E.g.: Status, Action, Reward, Status')
Randomly select some data
(Status, Action, Reward, Status')
Policy Network
(Func: Output expectation of all actions in Action_Func)
Target Network
(Func: Update network)
Q(Action)
max (Q(Action'))
Loss Function
(Reward+γMax(Q(Action'))-Q(Action))²

Figure 5: Model-C in a nutshell.

Model-B Training. For training Model-B and B', we reduce the allocated resources for a specific LC service from its OAA by fine-grain approaches, as illustrated in Figure 4. The reduction has three angles, i.e., horizontal, oblique, and vertical, i.e., B-Points include <cores dominated, LLC ways>, <cores, LLC ways>, <cores, LLC ways dominated>, respectively. For each fine-grain resource reduction step, we collect the corresponding QoS slowdowns and then label them as less equal to ($\leq$) 5%, 10%, 15%, and so on, respectively. Examples are illustrated in Figure 4, which shows the B-Points with the corresponding QoS slowdown. We collect the training data set for every LC service in Table 1. The data set contains 65,998,227 data tuples, covering 549,987 cases.

Model-B Function. We design a new loss function:

$$L = \frac{1}{n}\sum_{t=1}^{n}\left(\frac{y_t}{y_t + c} \times (s_t - y_t)\right)^2,$$

in which s_t is the prediction output value of Model-B, y_t is the labeled value in practice, and 'c' is a constant that is infinitely close to zero. We multiply the difference between s_t and y_t by $\frac{y_t}{y_t + c}$ for avoiding adjusting the weights during backpropagation in the cases where $y_t = 0$ and $\frac{y_t}{y_t + c} = 0$ caused by some non-existent cases (we label the non-existent cases as 0, i.e., $y_t = 0$, indicating we don't find a resource-QoS trading policy in the data collection process).

4.3 Model-C: Handling the Changes On the Fly

Model-C Description. Model-C corrects the resource under-/over-provision and conducts online training. Figure 5 shows the Model-C in a nutshell. Model-C's core component is an enhanced Deep Q-Network (DQN) [43], consisting of two neural networks, i.e., Policy Network and Target Network. The Policy and Target Network employ the 3-layer MLP, and each hidden layer has 30 neurons. Policy Network's **inputs** consist of the parameters in Table 3, and the **outputs** are resource scheduling actions (e.g., reducing/increasing a specific number of cores or LLC ways) and the corresponding expectations (defined as Q(action)). These actions numbered 0~48 are defined as Action_Function:$\{< m, n > | m \in [-3, 3], n \in [-3, 3]\}$, in which a positive m denotes allocating m more cores (i.e., add operation) for an application and a negative m means depriving it of m cores (i.e., sub operation); n indicates the actions on LLC ways. Figure 5 illustrates Model-C's logic. The scheduling action with the maximum expectation value (i.e., the action towards the best solution) will be selected in ① and executed in ②. In ③, Model-C will get the Reward value according to the Reward Function. Then, the tuple <Status, Action, Reward, Status'> will be saved in the Experience Pool in ④, which will be used during online training. The terms Status and Status' denote system's status described by the parameters in Table 3 before and after the Action is taken. Model-C can quickly have the ideal solutions in practice (about 2 or 3 actions). Please note that in ①, Model-C might randomly select an Action instead of the best Action with a 5% chance. By doing so, OSML avoids falling into a local optimum [43]. These random actions have a 44% chance of causing QoS violations when Model-C reduces resources. But OSML can handle the QoS violations by withdrawing the action (line 9 in Algo._3).

Model-C's Reward Function. The reward function of Model-C is defined as follow:

If $\mathrm{Latency}_{t-1} > \mathrm{Latency}_t$:

$\mathrm{R}_t = \log(1 + \mathrm{Latency}_{t-1} - \mathrm{Latency}_t) - (\Delta\mathrm{CoreNum} + \Delta\mathrm{CacheWay})$

If $\mathrm{Latency}_{t-1} < \mathrm{Latency}_t$:

$\mathrm{R}_t = -\log(1 + \mathrm{Latency}_t - \mathrm{Latency}_{t-1}) - (\Delta\mathrm{CoreNum} + \Delta\mathrm{CacheWay})$

If $\mathrm{Latency}_{t-1} = \mathrm{Latency}_t$:

$\mathrm{R}_t = -(\Delta\mathrm{CoreNum} + \Delta\mathrm{CacheWay})$,

where $\mathrm{Latency}_{t-1}$ and $\mathrm{Latency}_t$ denote the latency in previous and current status, respectively; ΔCoreNum and ΔCacheWay represent the changes in the number of cores and LLC ways, respectively. This function gives higher reward and expectation to Action that brings less resource usage and lower latency. Thus, Model-C can allocate appropriate resources. Algo._2 and 3 show the logic on using Model-C in detail.

Offline Training. A training data tuple includes Status, Status', Action and Reward, which denote the current status of a LC service, the status after these actions are conducted (e.g., reduce several cores or allocate more LLC ways) and the reward calculated using the above functions, respectively.

To create the training data set for Model-C, we resort to the data set used in Model-A training. The process is as follows. Two tuples in Model-A training data set are selected to denote Status and Status', and we further get the differences of the resource allocations between the two status (i.e., the actions that cause the status shifting). Then, we use the reward function to have the reward accordingly. These 4 values form a specific tuple in Model-C training data set. In practice, as there are a large number of data tuples in Model-A training data set, it is impossible to try every pair of tuples in the data set, we only select two tuples from resource allocation policies that have less than or equal to 3 cores, or 3 LLC ways differences. Moreover, we also collect the training data in the cases where cache way sharing happens and preserve them in the experience pool. Therefore, Model-C can work in resource-sharing cases. To sum up, we have 1,521,549,190 tuples in Model-C training data set.

Online Training. Model-C collects online traces. The training flow is in the right part of Figure 5. Model-C randomly selects some data tuples (200 by default) from the Experience Pool. For each tuple, Model-C uses the Policy Network to get the Action's expectation value (i.e., Q(Action) [43]) with the Status. In Model-C, the target of the Action's expectation value is the Reward observed plus the weighted best expectation value of the next status (i.e., Status'). As illustrated in Figure 5, Model-C uses the Target Network to have the expectation values of Status' for the actions in Action_Function and then finds the best one, i.e., $\mathrm{Max}(\mathrm{Q}(\mathrm{Action}'))$. We design a new Loss Function based on MSE: $(\mathrm{Reward} + \gamma\mathrm{Max}(\mathrm{Q}(\mathrm{Action}')) - \mathrm{Q}(\mathrm{Action}))^2$. It helps the Policy Network predict closer to the target. The Policy Network is updated during online training. The Target Network's weights are synchronized periodically with the Policy Network's weights. Doing so enables the Target Network to provide stable predictions for the best expectation value of Status' within a predefined number of time steps, thus improving the stability of the training and prediction.

Table 4: The Summary of ML models in OSML.

ML	Model	Features	Model Size	Loss Function	Gradient Descent	Activation Function
A	MLP	9	144 KB	Mean Square Error (MSE)	Adam Optimizer	ReLU
A'	MLP	12	155 KB			
B	MLP	13	110 KB	Modified MSE		
B'	MLP	14	106 KB	MSE		
C	DQN	8	141 KB	Modified MSE	RMSProp	

4.4 Discussions on the design of ML Models

(1) Why using MLPs. Table 4 characterizes the ML models used in OSML. We employ three-layered MLPs in Model-A and B, because they can fit continuous functions with an arbitrary precision given a sufficient number of neurons in each layer [67], and we can use extensive training data to improve the accuracy of MLPs for predicting OAAs and RCliffs. Moreover, after offline training, using MLPs brings negligible run-time overheads to OSML. **(2) Why do we need the Three models?** We divide the OSML's scheduling logic into three parts, which the three models cover, respectively. Models work in different scheduling phases, and no single model can handle all cases. For example, model-A predicts the RCliffs and OAAs; Model-B predicts the QoS variations and resource margins in co-location cases. DQN in Model-C learns online to shepherd the scheduling results from Model-A/B. They are necessary and work cooperatively to cover the main scheduling cases. Moreover, they are easier to generalize than other approaches, e.g., a table lookup approach (Sec.6.4). *Why Not Only use the online learning Model-C?* Model-C uses DQN that depends on the start points. It starts with Model-A/B's outputs to avoid exploring the whole (large) scheduling space. Without the approximate OAA provided by Model-A for many unseen cases, only using Model-C will incur more scheduling actions (overheads). **(3) Insights on Generalization for Unseen apps and New servers.** (i) We use "hold-out cross validation", i.e., the training data (70% of the whole data set) excludes the testing data (30%) for each LC service. (ii) We train models with extensive representative traces from many

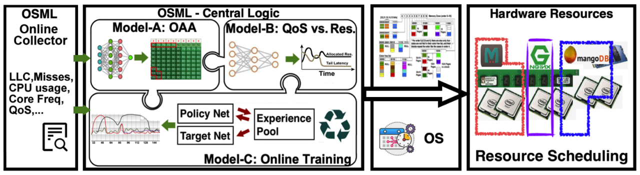


Figure 6: The overview of the system design of OSML.

typical services (e.g., memory/CPU intensive, random memory access, etc. [10,46]), fitting the correlation between architectural hints for mainstream features (e.g., IPC, cache miss, memory footprint, CPU/LLC utilization and MBL), OAA, and the QoS demands. For instance, the spearman correlation coefficient is 0.571, 0.499, and -0.457 between OAA and cache miss, MBL, and IPC, respectively. *On different platforms or for new/unseen applications*, these numbers might be varied; however, this correlation trend is not changed, enabling OSML to generalize to other situations. Even for errors caused by deviations from the training dataset for unseen applications/new platforms, model-C can learn online and correct them. (iii) Using transfer learning, collecting new traces on a new server for several hours will make OSML work well for it (refer to Sec.6.4). (iv) OSML is a long-term project open to the community; we continue adding new traces collected from new applications and new servers to the data set for enhancing models' performance for new cases.

5 OSML: System Design

5.1 The Central Control Logic

The overview of OSML is in Figure 6. OSML is a per-node scheduler. The central controller of OSML coordinates the ML models, manages the data/control flow, and reports the scheduling results. Figure 7 shows its overall control logic.

Allocating Resources for LC services. Algo._1 shows how OSML uses Model-A and B in practice. Figure 7 highlights its operations. For a newly coming LC service, the central controller calls Model-A via the interface *modelA_oaa_rcliff()* to get the OAA and RCliff. Suppose the current idle resources are not sufficient to satisfy the new LC service. In that case, OSML will enable Model-B through the interface *modelB_trade_qos_res()* to deprive some resources of other LC services with the allowable QoS slowdown (controlled by the upper-level scheduler) and then allocate them to the new one. The upper-level scheduler manages the cluster consisting of nodes using OSML. In the depriving process for a specific LC service, OSML reduces its allocated resources and gets close to the RCliff, but it will not easily fall off the RCliff unless expressly permitted (refer to Algo._4).

Dynamic Adjusting. Figure 7 shows the dynamic adjusting of Algo._2 and 3, in which Model-C works as a dominant role. During the run time, OSML monitors each LC service's QoS status for every second. If the QoS violation is detected, the central controller will enable Algo._2 and call Model-C to allocate more resources to achieve the ideal QoS. The interface is *modelC_upsize()*. If OSML finds an LC service is over-provisioned (i.e., wasting resources), Algo._3 will be used to reclaim them, and Model-C will be called via *modelC_downsize()*.

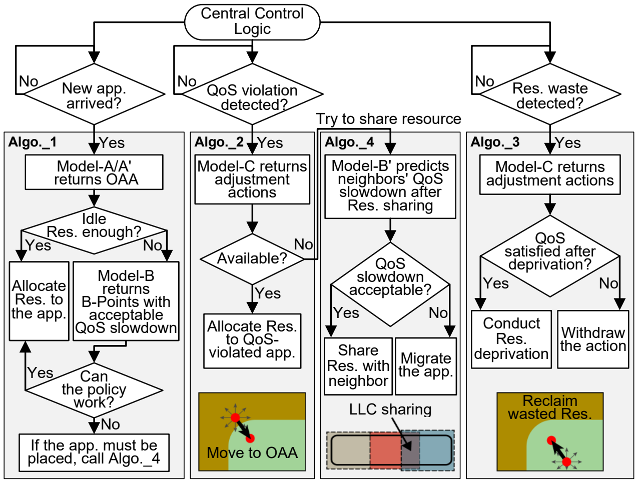


Figure 7: OSML's central logic.

Moreover, Algo._4 will enable resource sharing (the default scheduling is to do hard partitioning of cores/LLC ways), if all of the co-located LC services are close to their RCliff and the upper scheduler still wants to increase loads on this server. Model-A and B work cooperatively to accomplish this goal in Algo._4. In practice, to minimize the adverse effects, resource sharing usually happens between only two applications. Note that Algo._4 might incur the resource sharing over the RCliff, and thus may bring higher response latency for one or more LC services. OSML will report the potential QoS slowdown to the upper scheduler and ask for the decisions. If the slowdown is not allowed, the corresponding actions will not be conducted.

Bandwidth Scheduling. OSML partitions the overall bandwidth for each co-located LC service according to the ratio $\mathrm{BW_j}/\sum \mathrm{BW_i}$. $\mathrm{BW_j}$ is a LC service's OAA bandwidth requirement, which is obtained from the Model-A. Note that such scheduling needs Memory Bandwidth Allocation (MBA) mechanism [4,5] in CPU.

5.2 Implementation

OSML learns whether the LC services have met their QoS targets. OSML monitors the run-time parameters for each co-located LC service using performance counters for every second (default). If the observation period is too short, other factors (e.g., cache data evicted from the cache hierarchy, context switch) may interfere with the sampling results. Moreover, OSML performs well with other interval settings and allows configuration flexibility (e.g., 1.5 or 2 seconds).

We design OSML that works cooperatively with OS (Figure 6). As the kernel space lacks the support of ML libraries, OSML lies in the user space and exchanges information with the OS kernel. OSML is implemented using python and C. It employs Intel CAT technology [4] to control the cache way allocations, and it supports dynamically adjusting. OSML uses Linux's taskset and MBA [5] to allocate specific cores and bandwidth to an LC service. OSML captures the online performance parameters by using the pqos tool [4] and PMU

Algorithm 1: using ML to have OPT resources allocations. In practice, only one policy in OAA will be selected.

```
For a coming LC service, map it on the idle resources and capture its
  run-time parameters in Table 3 for 1 second (default);
Forward these parameters as inputs to Model-A (A' when several LC
  services are co-located);
Model-A outputs: (1) OAA to meet the target QoS; (2) OAA bw (3)
  RCliff in current environment;
if idle resources are sufficient to meet OAA then
    Allocate resources with a specific solution in OAA;
end
if idle resources are not enough then  // Enabling Model-B
    Calculate the difference between the idle resources and its'
      OAA, i.e., <+cores, +LLC ways>  // required resource to meet
      its QoS;
    Calculate the difference between the idle resources and RCliff,
      i.e., <+cores', +LLC ways'>  // should be used carefully;
    for each previously running LC service do
        if the one can tolerate a certain QoS slowdown then
            Use Model-B to infer the B-Points with the acceptable
              QoS slowdown;
            Model-B outputs the B-Points, i.e., <cores, LLC
              ways>, <cores dominated, LLC ways>, and etc.;
        end
    end
    Record each LC service's B-Points with the QoS slowdown;
    Find the best-fit solution to meet OAA/RCliff according to
      B-Points with at most 3 apps involved  // The less the better;
    if the solution could meet OAA or RCliff then
        Adjust allocations according to OAA (RCliff is alternative);
    else
        The LC service cannot be located on this server without
          sharing resources;
    end
end  // Enabling Model-B
Report results to upper scheduler, call Algo._4 for sharing if needed.
```

[5]. The ML models are based on TensorFlow [6] with the version 2.0.4, and can be run on either CPU or GPU.

6 Evaluations

6.1 Methodology

We evaluate the per-node OSML performance on our platform in Table 2. Details on LC services are in Table 1. The metrics involve the QoS (similar to [10], the QoS target of each application is the 99th percentile latency of the knee of the latency-RPS curve. Latency higher than the QoS target is a violation.); Effective Machine Utilization (EMU) [10] (the max aggregated load of all co-located LC services) – higher is better. We first evaluate the scenarios where LC services run at constant loads, and the loads are from 10% - 100%. Then, we explore workload churn. We inject applications with loads from 20% - 100% of their respective max load. Furthermore, to evaluate the generalization of OSML, we employ some new/unseen applications that are not in Table 1 and the new platform in our experiments. If an allocation in which all applications meet their QoS cannot be found after 3 mins, we signal that the scheduler cannot deliver QoS for that configuration.

6.2 OSML Effectiveness

We compare OSML with the most related approaches in [10,46] based on the latest open-source version.

Algorithm 2: handling resource under-provision cases.

```
1  for each allocated LC service do
2      if QoS is not satisfied ∧ latency is increasing then
3          Forward the current running status parameters to Model-C;
4          Model-C selects a specific action in the Action_Fun;
5          Return Model-C's output (<cores+, LLC ways+>) to
             OSML's central controller;
6          if <cores+, LLC ways+> can be satisfied within current
             idle resources then
7              OSML allocates, and GOTO Line 2;
8          else
9              Call Algorithm_4.  // Share resources w/ others?
10         end
11     end
12 end
```

Algorithm 3: handling resource over-provision cases.

```
for each allocated LC service do
    // Over-provision;
    if its QoS is satisfied ∧ no idle resources left in system ∧ at least
      one of its neighbor's QoS is not satisfied then
        Forward current run-time status parameters to Model-C;
        Model-C selects a specific action accordingly;
        Return Model-C's output (<cores-, LLC ways->) to
          OSML's central controller;
        OSML reduces the resources accordingly;
        if its QoS is not satisfied now then
            OSML withdraws the actions.  // Rollback
        end
    end
end
```

Algorithm 4: handling resources sharing among apps.

```
// OSML tries to allocate resources cross over RCiff;
Obtain how many resources a LC service needs, i.e., <+cores, +LLC
  ways>, from the neighbors to meet its QoS using Model-A;
for each potential neighbor App do
    Create sharing policies, i.e.,
      ${< \mathrm{u}, \mathrm{v} > | \forall \mathrm{u} \leq (+\mathrm{cores}) \wedge \forall \mathrm{v} \leq (+\mathrm{LLC\ ways}); \mathrm{u}, \mathrm{v} \geq 0}$;
    Use Model-B' to predict the neighbor's QoS slowdown
      according to ${< \mathrm{u}, \mathrm{v} >}$;
end
if the neighbors' QoS slowdown can be accepted by OSML then
    OSML conducts the allocation;
else
    OSML migrate the LC service to another node.
end
```

PARTIES [10]. It makes incremental adjustments in one-dimension resource at a time until QoS is satisfied – "trial and error" – for all of the applications. The core mechanism in [10] is like an FSM [60].

CLITE [46]. It conducts various allocation policies and samples each of them; it then feeds the sampling results – the QoS and run-time parameters for resources – to a Bayesian optimizer to predict the next scheduling policy.

Unmanaged Allocation (baseline). This policy doesn't control the allocation policies on cores, LLC, and other shared resources for co-located LC services. This policy relies on the original OS schedulers.

ORACLE. We obtain these results by exhaustive offline sampling and find the best allocation policy. It indicates the ceiling that the schedulers try to achieve.

We show the effectiveness of OSML as follow.

(1) OSML exhibits a shorter scheduling convergence time. Using ML models, OSML achieves OAA quickly and efficiently handles cases with diverse loads. Figure 8-a shows the distributions of the scheduling results of 104 loads for OSML,

PARTIES and CLITE, respectively. Every dot represents a specific workload that contains 3 co-located LC services with different RPS. We launch the applications in turn and use a scheduler to handle QoS violations until all applications meet their QoS targets. The x-axis shows the convergence time; the y-axis denotes the achieved EMU. Generally, OSML can achieve the same EMU with a shorter convergence time for a specific load. Figure 8-b shows the violin plots of convergence time for these loads. On average, OSML takes 20.9 seconds to converge, while PARTIES and CLITE take 32.7 and 46.3 seconds, respectively. OSML converges 1.56X and 2.22X faster than PARTIES and CLITE. OSML performs stably – the convergence time ranges from 5.3s (best case) to 80.0s (worst case). By contrast, the convergence time in PARTIES ranges from 5.5s to 111.1s, and CLITE is from 14.0s to 140.6s. OSML converges faster mainly because the start point in the scheduling space provided by Model-A is close to OAA. PARTIES and CLITE take a longer convergence time, indicating that they require high scheduling overheads in cloud environments. In Cloud, jobs come and go frequently; thus, scheduling occurs frequently, and longer scheduling convergence time often leads to unstable/low QoS.

We further analyze how these schedulers work in detail. Figure 9-a/b/c show the actions used in OSML, PARTIES, and CLITE's scheduling process for case A in Figure 8. This case includes Moses, Img-dnn, and Xapian with 40%, 60%, and 50% of their maximum loads. For this load, PARTIES, CLITE and OSML take 14.5 seconds, 72.6 seconds and 8.2 seconds to converge, respectively. Figure 9 highlights scheduling actions using solid red lines to represent increasing resources and blue dotted lines to denote reducing resources. Figure 9-a shows PARTIES takes 7 actions for scheduling cores and 1 action for cache ways. It schedules in a fine-grained way by increasing/decreasing one resource at a time. CLITE relies on the sampling points in the scheduling exploration space. Figure 9-b shows CLITE repeats sampling until the "expected improvement" in CLITE drops below a certain threshold. CLITE only performs five scheduling actions according to its latest open-source version; but it takes the longest convergence time (72.6 seconds). The underlying reason is that CLITE's sampling/scheduling doesn't have clear targets. In practice, the improper resource partitions/allocations during sampling lead to the accumulation of requests, and the requests cannot be handled due to resource under-provision. Therefore, it brings a significant increase in response latency. Moreover, due to the early termination of CLITE's scheduling process, CLITE cannot schedule resources to handle QoS violations in a timely manner, leading to a long convergence time. Figure 9-c shows OSML achieves OAA for each LC service with 5 actions. Compared with prior schedulers, OSML has clear aims and schedules multiple resources simultaneously to achieve them. It has the shortest convergence time – 8.2 seconds.

Moreover, as the scheduling is fast, OSML often supports more loads. Figure 10 shows the OSML's results on scheduling the three LC services – Moses, Img-dnn, and Xapian. For a specific scheduling phase, by using ML to achieve OAA, OSML supports 10~50% higher loads than PARTIES and CLITE in highlighted cells in Figure 10-d, accounting for 58% of the cases that can be scheduled. All schedulers perform better than the Unmanaged (Figure 10-a), as they reduce the resource contentions.

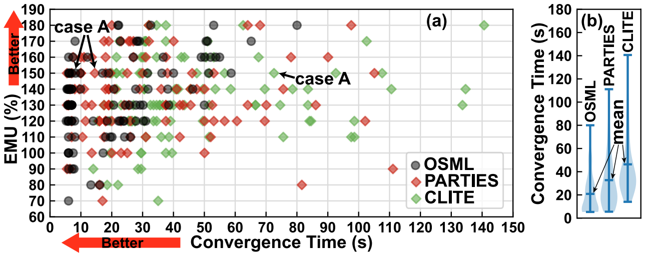


Figure 8: (a) The performance distributions for 104 loads that OSML, PARTIES and CLITE can all converge. (b) Violin plots of convergence time for loads in (a).

(2) Compared with PARTIES and CLITE, OSML consumes fewer resources to support the same loads to meet the QoS targets. On average, for the 104 loads in Figure 8-a, OSML uses 34 cores and 16 LLC ways; by contrast, PARTIES and CLITE exhaust all the platform's 36 cores and 20 LLC ways. As illustrated in Figure 9-a, PARTIES partitions the LLC ways and cores equally for each LC service at the beginning; once it meets the QoS target (using 8 actions), it stops. Thus, PARTIES drops the opportunities to explore alternative better solutions (i.e., using fewer cores or cache ways to meet identical QoS targets). PARTIES allocates all cores and LLC ways finally. CLITE also uses all cores and cache ways shown in Figure 9-b. By contrast, OSML schedules according to applications' resource requirements instead of using all resources. Figure 9-c shows that using Model-A, OSML achieves each LC service's OAA (the optimal solution) after 5 actions. OSML detects and reclaims over-provided resources using Model-C. For example, the last action in Figure 9-c reclaims 3 cores and 2 LLC ways from Xapian. Finally, OSML saves 3 cores and 9 LLC ways. As OSML is designed for LC services that are executed for a long period, saving resources means saving budgets for cloud providers.

(3) Using ML models, OSML provides solutions for sharing some cores and LLC ways among LC services, therefore supporting higher loads. PARTIES and CLITE don't show resource sharing in the original design. Using Algo._4, OSML lists some potential resource sharing solutions, and then enables Model-B' to predict the QoS slowdown for each case. The sharing solution with a relatively lower QoS slowdown is selected. More details refer to Figure 7. Figure 9-d shows how OSML shares resources for the highlighted case B in Figure 10-d. OSML enables Model-C to add resources for Moses in Algo._2 and uses Algo._4 to share 2 CPU cores with Xapian. Finally, the QoS is met. By enabling resource sharing, OSML can support higher loads than PARTIES and CLITE, and can even be close to ORACLE in Figure 10-e. If not

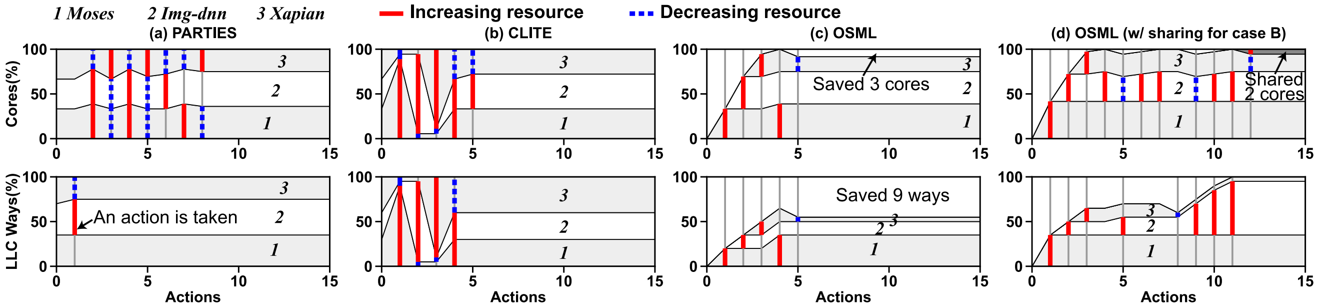


Figure 9: Resource usage comparisons for OSML, PARTIES, and CLITE.

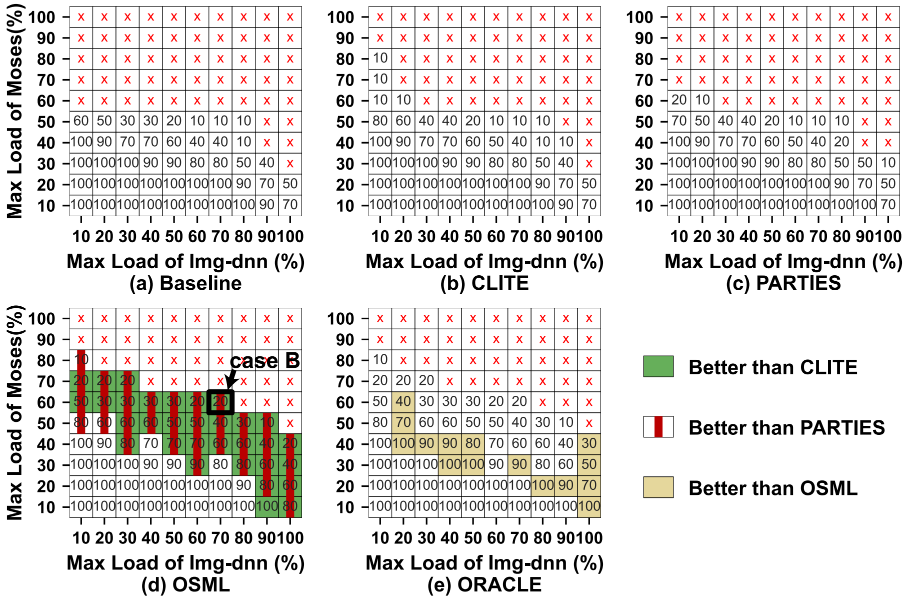


Figure 10: Co-location of Moses, Img-dnn and Xapian. The heatmap values are the percentage of the third LC service's (i.e., Xapian) achieved max load without QoS violations in these cases. The x- and y-axis denote the first and second app's fraction of their max loads (with QoS target), respectively. Cross means QoS target cannot be met. The related studies [10,46] use heat maps to show their effectiveness, so we also use heat maps for comparisons in this work.

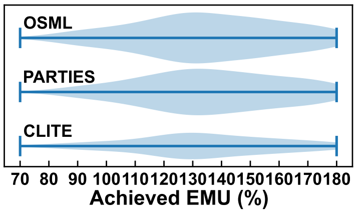


Figure 11: Throughput distribution.

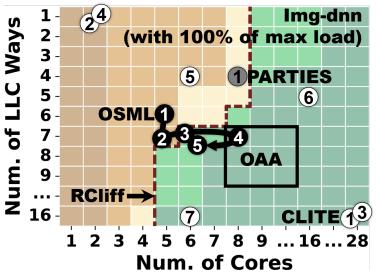


Figure 13: Highlighted the scheduling traces in scheduling space for all schedulers from time point 180 to 228 in Figure 12. Each circle denotes a specific scheduling policy conducted by a specific scheduler. The number in a circle denotes the sequence of these scheduling actions during the scheduling phase.

OSML, however, the "trial and error" approach has to try to share core/cache way in a fine-grain way among applications, and then observes the real-time latency for making a further decision, inevitably incurring higher scheduling overheads and bringing sharp QoS slowdown if falling off the RCliff.

(4) OSML promptly handles the resource under/over-provision and QoS violations using Model-C. Based on Model-A/B's results, Model-C shepherds and adjusts the allocations with several actions for each application in our experiments and converges more quickly than previous approaches. More experiments on dynamic, complicated cases can be found in Sec.6.3. *Can we only use Model-C or only use Model-A/B?* Enabling the three models is necessary for OSML. For the case in Figure 9-c, when OSML uses the three models collaboratively for scheduling, it takes 8.2s and 5 actions to achieve OAA for all applications. By contrast, if only enabling Model-C, it takes 18.5s and 13 actions. Because Model-A/B can at least provide an approximate OAA for scheduling, thus reducing the convergence time. Just using Model-A/B may lead to 4-core errors for an unseen LC service (Table 5). So, Model-C is needed to correct the errors. We cannot disable any models in OSML in practice.

(5) OSML performs well in various cases. Figure 11 shows the EMU distribution for converged loads among 302 loads (each has 3 apps with diverse RPS). EMU reflects the system throughput [10,32]. OSML can have scheduling results for 285 loads; PARTIES and CLITE can work for 260 and 148 loads, respectively. OSML's distribution in Figure 11 is wider than PARTIES and CLITE. This indicates that OSML works for more loads, including those with a high EMU (e.g., 130-180%). Even for the loads that OSML, PARTIES, and CLITE can all converge, OSML can converge faster (Figure 8).

6.3 Performance for Workload Churn

We evaluate how OSML behaves with dynamically changing loads. Each LC service's QoS is normalized to the solely running case. As illustrated in Figure 12-a, in the beginning, Moses with 60% of max load arrives; then Sphinx with 20% of max load and Img-dnn with 60% of max load arrive. We observe their response latency increases caused by the re-

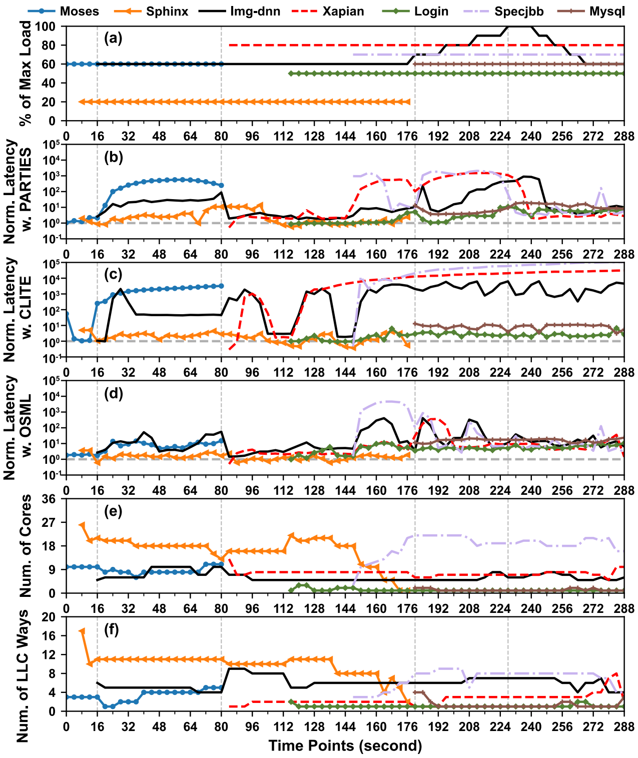


Figure 12: OSML's performance in reality with varying loads.

source contentions among them. In Figure 12-b, PARTIES aids the QoS violations step by step. During the scheduling, Moses always has high latency until it ends at 80 seconds. CLITE's scheduling relies on sampling and Bayesian optimizer. CLITE starts scheduling at time point 16, where all the three services arrive. At 32s, CLITE obtains the scheduling solution for these LC services after five sampling steps. But it doesn't satisfy these services' QoS targets. In Figure 12-c, Moses and Img-dnn still have high latency. By contrast, with Model-A's OAA predictions and Model-C's online scheduling, OSML quickly provides better scheduling solutions at time point 48 for all three services. During the identical scheduling phase (e.g., the time point 16 to 80), we can observe the lowest overall normalized latency in Figure 12-d. Moreover, Figure 12-e and f illustrate OSML's scheduling actions for achieving ideal solutions. In short, within a few scheduling actions (scheduling overheads) that schedule multiple resources, OSML quickly meets the QoS targets.

From 180 to 228, we increase the load for Img-dnn as illustrated in Figure 12-a. OSML meets Img-dnn's changing demands by using Model-C. PARTIES does not reflect quickly for this change, and it works for other services. Thus, as illustrated in Figure 12-b, the QoS violation is not aided until 244s, when Img-dnn's load decreases. For CLITE, it has to sample each time when the load changes. But during the sampling, a specific service might not have sufficient resources to handle the requests; thus, the requests are accumulated, leading to QoS fluctuations/violations during the scheduling (Figure 12-c). Figure 13 highlights the scheduling actions for Img-dnn from 180 to 228. During this phase, PARTIES does not add resources for Img-dnn; but it add more resources for Specjbb and Xapian as they are with higher latency. Img-dnn's response latency keeps increasing. CLITE samples several scheduling policies in the scheduling space, but does not converge and thus incurs QoS fluctuations. By contrast, OSML's Model-C can handle QoS violations of Img-dnn even the load increases. Moreover, as mentioned before, OSML saves resources and thus it can serve more workloads. For example, as shown in Figure 12, Mysql (an **unseen** workload in training) comes at time point 180; OSML allocates the saved cores to it without sharing or depriving other LC services of resources.

Table 5: ML Models' performance on average. The units of errors are the number of cores/LLC ways. For Model B', the unit is the percentage of QoS slowdown.

ML	Outputs	Error		Errors for unseen LC services		Err on new platforms (TL)		MSE	Over-heads
		Core	LLC	Core	LLC	Core	LLC		
A	RCliff	0.589	0.288	1.266	0.198	2.142	0.542	0.0025	0.20s
	OAA	0.580	0.360	1.276	0.191	2.004	0.865		
A'	RCliff	1.072	0.815	3.403	1.835	0.772	0.411	0.0135	0.20s
	OAA	1.072	0.814	3.404	1.835	0.790	0.413		
B	B-Points	0.612	0.053	4.012	0.167	2.320	0.969	0.0012	0.18s
	B-Points,Core dominated	0.314	0.048	3.434	0.937	2.250	0.815		
	B-Points,Cache dominated	0.093	0.462	0.789	0.783	1.868	1.519		
B'	QoS reduction	7.87%		8.33%		11.28%		0.0035	0.19s
C	Scheduling actions	0.908	0.782	0.844	0.841	1.390	1.801	0.7051	0.20s

6.4 New/Unseen Apps and New Platforms

Generalization. Based on our data set, the ML models can be well trained. It will be at most 4-core error for unseen applications (Table 5). We evaluate OSML using unseen applications that are not used in training, i.e., Silo [62], Shore[62], Mysql [70], Redis [77], and Node.js [78]. They exhibit diverse computing/memory patterns. We evaluate the convergence time of OSML with 3 groups of workloads. Each group has 15 workloads, and each workload has 3 LC services. The workloads in Group 1 have 1 unseen application. The workloads in Group 2 and 3 have 2 and all unseen applications, respectively. OSML takes 24.6s, 29.3s, and 31.0s to converge for the 3 groups of workloads, on average, respectively. OSML performs well, even for unseen cases, showing good generalization performance. PARTIES uses 34.9s, 43.6s, and 42.8s; CLITE uses 58.5s, 60.6s, and 46.5s for these applications. Note that PARTIES and CLITE's scheduling don't need previous knowledge for applications, so their performance doesn't depend on whether the application is seen/unseen.

For new platforms, we use fine-tuning in transfer learning (TL). We freeze the first hidden layer of the MLPs; we retrain the last two-hidden layers and the output layer using the traces collected on two new platforms (w/ CPU Xeon Gold 6240M and E5-2630 v4, respectively). For each LC service, based on our data set, collecting new traces on a new platform for several hours will be sufficient (covering the more allocation cases, the better). The model updating can be accomplished in hours. The time consumption will be shorter if using multiple machines in parallel. Table 5 shows the average values of ML models' quality. The new models' prediction errors are slightly higher than the previous models on the original platforms, but OSML still handles them well. By contrast, if we use a table lookup approach instead, we have to use additional memory to store the data tuples, e.g., 60GB will be wasted for the current data set to replace Model-A. More importantly, this approach is difficult to generalize for new/unseen applications or platforms, as their traces and the corresponding OAAs don't exist in the current data set.

Overheads. OSML takes 0.2s of a single core for each time during the interval setting (e.g., 1 second by default in Sec.5.2). 0.01s for ML model and 0.19s for online monitoring. As our models are light-weighted (OSML uses only one core), running them on CPU and GPU has a similar overhead. If models are on GPU, it takes an extra 0.03s for receiving results from GPU. OSML doesn't monopolize GPU. Generally, the overhead doesn't hinder the overall performance. In the cloud, applications' behaviors may change every second due to the diversity of user demands. Thus, OSML plays a critical role during the entire run time. **For building models**, using our current data set on platform in Table 2, it takes 3.3 mins, 5 mins, and 8.3 hours to train Model-A, B, and C for one epoch (all training samples are used to train once), respectively. We train models for ten epochs. Training can be accelerated using the popular Multi-GPU training technology. Doing so is practical in datacenters, and training time will not impede practice.

7 Related Work and Our Novelty

ML for System Optimizations. The work in [55] employs DNN to optimize the buffer size for the database systems. The efforts in [22,56,34] leverage ML to optimize computer architecture or resource management in the network to meet various workloads. The studies in [9,39] use ML to manage on-chip hardware resources. CALOREE [41] can learn key control parameters to meet latency requirements with minimal energy in complex environments. The studies in [26,31,58,59] optimize the OS components with learned rules or propose insights on designing new learned OS. *In OSML, we design an intelligent multi-model collaborative learning approach, providing better co-location solutions to meet QoS targets for LC services faster than the latest work stably.*

ML for Scheduling. Decima [35] designs cluster-level data processing job scheduling using RL. Resource Central [12] builds a system that contains the historical resource utilization information of the workloads used in Azure and employs ML to predict resource management for VMs. The study in [40] uses RL to predict which subsets of operations in a TensorFlow graph should run on the available devices. Paragon [14] classifies and learns workload interference. Quasar [15] determines jobs' resource preferences on clusters. Sinan [74] uses ML models to determine the performance dependencies between microservices in clusters. They are cluster schedulers [14,15,74]. By contrast, OSML deeply studies scheduling in co-location cases. Selecta [72] predicts near-optimal configurations of computing and storage resources for analytics workloads based on profiling data. CLITE [46] uses Bayesian optimization for scheduling on-node resources. The work in [48] applies ML to predict the end-to-end tail latency of LC service workflows. Twig [63] uses RL to characterize tail latency for energy-efficient task management. CuttleSys [76] leverages data mining to identify suitable core and cache configurations for co-scheduled applications. *For complicated co-location cases, OSML can avoid RCliffs and quickly achieve the ideal allocations (OAA) for multiple interactive resources simultaneously for LC services. Moreover, OSML performs well in generalization.*

Resource Partitioning. PARTIES [10] partitions cache, main memory, I/O, network, disk bandwidth, etc., to provide QoS for co-located services. The studies in [17,28,57,71] design some new LLC partitioning/sharing policies. The efforts in [23,27,44,45,73] show that considering cooperative partitioning on LLC, memory banks and channels outperforms one-level memory partitioning. However, the cooperative partitioning policies need to be carefully designed [29,30,37], and [16,32] show the heuristic resource scheduling approach could be ineffective in many QoS-constrained cases. [7,11] study the "performance cliff" on cache for SPECCPU 2006 applications and Memcached. Caladan [75] doesn't involve cache optimizations, and core/cache cliffs cannot be avoided, causing QoS fluctuations in some cases. *By contrast, OSML is the first work that profoundly explores cache cliff and core cliff simultaneously (i.e., RCliff) for many widely used LC services in co-location cases. OSML is a representative work using ML to guide the multiple resources partitioning in co-location cases; OSML is cost-effective in new cloud environments.*

8 Conclusion

We present OSML, an ML-based resource scheduler for co-located LC services. We learn that straightforwardly using a simple ML model might not handle all of the processes during the scheduling. Therefore, using multiple ML models cooperatively in a pipe-lined way can be an ideal approach. More importantly, we advocate the new solution, i.e., leveraging ML to enhance resource scheduling, could have an immense potential for OS design. In a world where co-location and sharing are a fundamental reality, our solution should grow in importance and benefits our community.

Acknowledgement

We thank the reviewers and our shepherd Young-ri Choi for the invaluable comments. This work is supported by the Key-Area R&D Program of Guangdong (No.2021B0101310002), NSFC (No.62072432). L. Liu thanks Hecheng's arriving. X. Dou and Y. Chen are students in Sys-Inventor Lab led by L. Liu.

References

[1] "How 1s could cost amazon $1.6 billion in sales." https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales

[2] "Microservices workshop: Why, what, and how to get there," http://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference

[3] "State of the Cloud Report," http://www.righscale.com/lp/state-of-the-cloud. Accessed: 2019-01-28

[4] "Improving real-time performance by utilizing cache allocation technology," https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf, Intel Corporation, April, 2015

[5] "Intel 64 and IA-32 Architectures Software Developer's Manual," https://software.intel.com/en-us/articles/intel-sdm, Intel Corporation, October, 2016

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "TensorFlow: A System for Large-Scale Machine Learning," in OSDI, 2016

[7] Nathan Beckmann, Daniel Sanchez, "Talus: A Simple Way to Remove Cliffs in Cache Performance," in HPCA, 2015

[8] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, Mor Harchol-Balter, "RobinHood: Tail Latency Aware Caching – Dynamic Reallocation from Cache-Rich to Cache-Poor," in OSDI, 2018

[9] Ramazan Bitirgen, Engin Ipek, Jose F. Martinez, "Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach," in Micro, 2008

[10] Shuang Chen, Christina Delimitrou, José F. Martínez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in ASPLOS, 2019

[11] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, Sachin Katti, "Cliffhanger: Scaling Performance Cliffs in Web Memory Caches," in NSDI, 2016

[12] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, Ricardo Bianchini, "Resource Central: Understanding and Predicting Worloads for Improved Resource Management in Large Cloud Platforms," in SOSP, 2017

[13] Jeff Dean, David A. Patterson, Cliff Young, "A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution," in IEEE Micro, 2018

[14] Christina Delimitrou, Christos Kozyrakis, "QoS-Aware Scheduling in Heterogenous Datacenters with Paragon," in ACM TOCS, 2013

[15] Christina Delimitrou, Christos Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in ASPLOS, 2014

[16] Yi Ding, Nikita Mishra, Henry Hoffmann, "Generative and Multi-phase Learning for Computer Systems Optimization," in ISCA, 2019

[17] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, Daniel Sanchez, "KPart: A hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in HPCA, 2018

[18] Yu Gan and Christina Delimitrou, "The Architectural Implications of Cloud Microservices," in IEEE Computer Architecture Letters, 2018

[19] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, Christina Delimitrou, "Leveraging Deep Learning to Improve Performance Predictability in Cloud Microservices with Seer," in ACM SIGOPS Operating Systems Review, 2019

[20] Mark D. Hill, Michael R. Marty, "Amdahl's Law in the Multicore Era," in IEEE Computers, 2008

[21] Kurt Hornik, "Approximation Capabilities of Multilayer Feedforward Networks," in Neural Networks, 1991

[22] Engin Ipek, Onur Mutlu, José F. Martínez, Rich Caruana, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," in ISCA, 2008

[23] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Michael Sullivan, Ikhwan Lee, Mattan Erez,"Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems,"in HPCA, 2012

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, Doe Hyun Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in ISCA, 2017

[25] Alex Krizhevsky, Ilya Sutskever, Geoffrey Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in Advances in neural information processing systems, 2012

[26] Yanjing Li, Onur Mutlu, Subhasish Mitra, "Operating System Scheduling for Efficient Online Self-Test in Robust Systems," in ICCAD, 2009

[27] Lei Liu, et al, "A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems," in PACT, 2012

[28] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, P. Sadayappan, "Gaining insights into mlticore cache partitioning: bridging the gap between simulation and real systems," in HPCA, 2008

[29] Fang Liu, Yan Solihin, "Studying the Impact of Hardware Prefetching and Bandwidth Partitioning in Chip-Multiprocessors," in Sigmetrics, 2011

[30] Seung-Hwan Lim, Jae-Seok Huh, Yougjae Kim, Galen M. Shipman, Chita R. Das, "D-Factor: A Quantitative Model of Application Slow-Down in Multi-Resource Shared Systems," in Sigmetrics, 2012

[31] Lei Liu, et al, "Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random?" in IEEE Trans. on Computers, 2016

[32] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, Christos Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in ISCA, 2015

[33] Lei Liu, et al, "Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems," in IEEE Trans. on Parallel and Distributed Systems, 2019

[34] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, Srikanth Kandula, "Resource Management with Deep Reinforcement Learning," in HotNet-XV, 2016

[35] Hongzi Mao, Malte Schwarzkopf, Shaileshh B. Venkatakrishnan, Zili Memg, Mohammad Alizadeh, "Learning Scheduling Algorithms for Data Processing Clusters," in SIGCOMM, 2019

[36] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John, "CSALT: Context Switch Aware Large TLB," in Micro, 2017

[37] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, Mary Lou Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations," in Micro, 2011

[38] Jason Mars, Lingjia Tang, Mary Lou Soffa, "Directly Characterizing Cross Core Interference Through Contention Synthesis," in HiPEAC, 2011

[39] Jose F. Martinez, Egin Ipek, "Dynamic multicore resource management: A machine learning approach," in IEEE Micro 29 (5):8-17 (2009)

[40] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, Jeff Dean, "Learning Device Placement in Tensorflow Computations," in Arxiv 1706.04972

[41] Nikita Mishra, Connor Imes, John D. Lafferty, Henry Hoffmann,

"CALOREE: Learning Control for Predictable Latency and Low Energy," in ASPLOS, 2018

[42] Nikita Mishra, Harper Zhang, John Lafferty, Henry Hoffmann, "A probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints," in ASPLOS, 2015

[43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, Demis Hassabis, "Human-level control through deep reinforcement learning," in Nature 518 (7540): 529-533, 2015

[44] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, Thomas Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in Micro, 2011

[45] Jinsu Park, Seongbeom Park, Woongki Baek, "CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers," in EuroSys, 2019

[46] Tirthak Patel, Devesh Tiwari, "CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers," in HPCA, 2020

[47] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout, "Arachne: Core-Aware Thread Management," in OSDI, 2018

[48] Joy Rahman, Palden Lama, "Predicting the End-to-End Tail Latency of Containerized Microservices in the Cloud," in IC2E, 2019

[49] Yizhou Shan, Yutong Huang, Yilun Chen, Yiying Zhang, "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation," in OSDI, 2018

[50] Prateek Sharma, Ahmed Ali-Eldin, Prashant Shenoy, "Resource Deflation: A New Approach For Transient Resource Reclamation," in EuroSys, 2019

[51] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, Demis Hassabis, "Mastering the game of Go with deep neural networks and tree search," in Nature, 529 (7587), 2016

[52] Akshitha Sriraman, Abhishek Dhanotia, Thomas F. Wenisch, "SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale," in ISCA, 2019

[53] Akshitha Sriraman, Thomas F. Wenisch, "µTune: Auto-Tuned Threading for OLDI Microservices," in OSDI, 2018

[54] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scoott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, "Going deeper with convolutions," in CVPR, 2015

[55] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, Rui Zhang, "iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases," in VLDB, 2019

[56] Stephen J. Tarsa, Rangeen Basu Roy Chowdhury, Julien Sebot, Gautham Chinya, Jayesh Gaur, Karthik Sankaranarayanan, Chit-Kwan Lin, Robert Chappell, Ronak Singhal, Hong Wang, "Post-Silicon CPU Adaptations Made Practical Using Machine Learning," in ISCA, 2019

[57] Xiaodong Wang, Shuang Chen, Jeff Setter, Jose F. Martínez, "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support," in HPCA, 2017

[58] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee, "Nimble Page Management for Tiered Memory Systems," in ASPLOS, 2019

[59] Yiying Zhang, Yutong Huang, "Learned Operating Systems," in ACM SIGOPS Operating Systems Review, 2019

[60] Zhijia Zhao, Bo Wu, Xipeng Shen, "Challenging the "Embarrassingly Sequential": Parallelizing Finite State Machine-based Computations through Principled Speculation," in ASPLOS, 2014

[61] Xiaoya Xiang, Chen Ding, Hao Luo, Bin Bao, "HOTL: A higher order theory of locality," in ASPLOS, 2013

[62] Harshad Kasture, Daniel Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in IISWC, 2016

[63] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, Magnus Sjalander, "Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services," in HPCA, 2020

[64] MongoDB official website. http://www.mongodb.com

[65] Memcached official website. https://memcached.org

[66] NGINX official website. http://nginx.org

[67] https://en.wikipedia.org/wiki/Universal_approximation_theorem

[68] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, Christina Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in ASPLOS, 2019

[69] Kalyanmoy Deb, Shivam Gupta, "Understanding Knee Points in Bicriteria Problems and Their Implications as Preferred Solution Principles," in Engineering Optimization, 43 (11), 2011

[70] www.mysql.com

[71] Harshad Kasture, Daniel Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads," in ASPLOS, 2014

[72] Ana Klimovic, Heiner Litz, Christos Kozyrakis, "Selecta: Learning Heterogeneous Cloud Storage Configuration for Data Analytics," in USENIX ATC, 2018

[73] Harshad Kasture, Xu Ji, Nosayba El-Sayed, Xiaosong Ma, Daniel Sanchez, "Improving Datacenter Efficiency Through Partitioning-Aware Scheduling," in PACT, 2017

[74] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, Christina Delimitrou, "Sinan: ML-Based and QoS-Aware Resource Management for Cloud Microservices," in ASPLOS, 2021

[75] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, Adam Belay, "Caladan: Mitigating Interference at Microsecond Timescales," in OSDI, 2020

[76] Neeraj Kulkarni, Gonzalo Gonzalez-Pumariega, Amulya Khurana, Christine A Shoemaker, Christina Delimitrou, David H Albonesi, "Cuttlesys: Data-driven Resource Management for Interactive Services on Re configurable Multicores," in Micro, 2020

[77] Redis official website. https://redis.io/

[78] Node.js official website. https://nodejs.org/en/

[79] Lei Liu, "QoS-Aware Resources Scheduling for Microservices: A Multi-Model Collaborative Learning-based Approach," in arXiv:1911.13208v2, 2019

[80] Lei Liu, et al, "Going Vertical in Memory Management: Handling Multiplicity by Multi-policy," in ISCA, 2014

CJFS: Concurrent Journaling for Better Scalability

Joontaek Oh* Seung Won Yoo* Hojin Nam* Changwoo Min † Youjip Won *
*KAIST † Virginia Tech

Abstract

In this paper, we propose CJFS, *Concurrent Journaling Filesystem.* CJFS extends EXT4 and addresses the fundamental limitations of the EXT4 journaling design, which are the main cause of the poor scalability of EXT4. The heavy-weight EXT4 journal suffers from two limitations. First, the journal commit is a strictly serial activity. Second, the journal commit uses the original page cache entry, not the copy of it, and subsequently any access to the in-flight page cache entry is blocked. To address these limitations, we propose four techniques, namely Dual Thread Journaling, Multi-version Shadow Paging, Opportunistic Coalescing, and Compound Flush. With Dual Thread design, CJFS can commit a transaction before the preceding journal commit finishes. With Multi-version Shadow Paging, CJFS can be free from the transaction conflict even though there can exist multiple committing transactions. With Opportunistic Coalescing, CJFS can mitigate the transaction lock-up overhead in journal commit so that it can increase the coalescing degree – *i.e.*, the number of system calls associated with a single transaction – of a running transaction. With Compound Flush, CJFS minimizes the number of flush calls. CJFS improves the throughput by 81%, 68% and 125% in filebench `varmail`, `dbench`, and `OLTP-Insert` on `MySQL`, respectively, against EXT4 by removing the transaction conflict and lock-up overhead.

1 Introduction

Filesystem scalability gets emphasized further as the computer system is loaded with hundreds of CPU cores [6,7,17,22,28,32,33,35]. A single server machine can simultaneously run hundreds of containers [2,43,50], each of which may frequently synchronize its local filesystem state to the disk [13,31]. The throughput of the server machine hinges upon the scalability of the filesystem journaling of the host filesystem.

In this paper, we address the scalability issue of the EXT4 journaling. EXT4 journaling uses page granularity physical logging [47,48]. EXT4 journaling suffers from two critical drawbacks; serial commit and committing the original page cache entry. In EXT4, journal commit is strictly serial activity. It can commit the following journal transaction only after the preceding journal commit finishes. As a result, in EXT4, there can be at most one running transaction and at most one committing transaction at a time. Moreover, EXT4 uses the original page cache entry in committing the updated contents to the disk. It does not create a copy of the updates for the journal commit. In this paper, we address both issues and propose a new filesystem, CJFS, by *Concurrent Journaling Filesystem.*

A fair amount of works have been dedicated to addressing the scalability issue of the filesystem journaling [6,8,15,17,24,44]. A few works proposed to maintain the multiple running transactions in EXT4 so that contention on the global running transaction is mitigated. ScaleFS [15] allocates a running transaction per each CPU core [6,8], where each core is allocated the separate filesystem partition. Son et al. [44] adopts a lock-free data structure to the journal transaction. Another body of works proposed to maintain the multiple committing transactions. IceFS [24] and SpanFS [15] partition the filesystem into multiple regions. They allocate a separate journal area for each region. The journal commit operations to each journal area can proceed in parallel.

Despite all the sophisticated approaches mentioned above, these variants of EXT4 journaling still fail to address the fundamental limitations of the EXT4 journaling; serial commit and committing the original page cache entry. In these works, the journal commit operations for the same journal region are still serialized [15,24]. Multiple running transactions and multiple committing transactions can conflict with each other and if the concurrent transactions conflict with each other, they are serialized.

In this paper, we address the fundamental limitations of the EXT4 journaling mechanism; serial commit and using the original page cache entry in the journal commit. The contribution of CJFS can be summarized as follows:

- **Dual Thread Journaling**: We separate the journal commit operation into two separate tasks; transferring the log blocks to the disk and making them durable. We allocate a separate thread for each operation. With Dual Thread Journaling, CJFS can commit a transaction while the preceding journal commit is still in progress.
- **Multi-Version Shadow Paging**: CJFS adopts multi-version shadow paging to resolve the transaction conflict. With multi-version shadow paging, CJFS uses the "copy" of the updated page cache entry in journal commit, so the transaction is free from the transaction conflict.
- **Opportunistic Transaction Coalescing**: CJFS adopts *opportunistic coalescing* to mitigate the trans-

action lock-up overhead. To increase the compound degree of the journal transaction, CJFS releases the running transaction from the LOCKED state when it finds that the running transaction conflicts with one of the committing transactions.

- **Compound Flush**: CJFS creates a large number of flush commands since it creates the multiple committing transactions in-flight, each of which issues a flush command separately to make its journal transaction durable. To relieve the overhead of servicing the flush commands, CJFS compounds multiple consecutive flushes from the concurrent transactions into a single flush. Compound flush significantly reduces the latency of the individual `fsync()` calls.

We implement the CJFS in Linux 5.18.18. CJFS yields superior performance not only to Vanilla EXT4 but also to the other recent works including BarrierFS [49], SpanFS [15] and FastCommit [42] in `varmail`, `dbench` and `OLTP-Insert` workloads.

2 Background and Motivation

2.1 Journaling in EXT4

Block granularity physical logging. A journaling filesystem logs the updated metadata either in a block granularity, e.g. EXT4 [36] or in a metadata granularity, e.g. XFS [45]. *Physical block granularity logging* of EXT4 is not only expensive but also unable to scale.

EXT4 maintains a set of page cache entries that need to be logged to the disk for journaling. It is called *running transaction*. When the system call updates the filesystem metadata, it first acquires a lock on the associated kernel object (e.g., directory mutex) and obtains the journal handle. A journal handle is a kind of ticket-like permission to add page cache entries to the running transaction. After the application is granted with the lock and the journal handle, the application modifies page cache entries. After modifying page cache entries, it inserts the updated page cache entries to the running transaction.

EXT4 commits the running transaction either periodically or by the explicit request from the application, e.g., `fsync()`. EXT4 commits the original page cache entry of the modified data. The application that needs to update the filesystem state is blocked if the associated page cache entry is being committed to the disk. We call this situation *transaction conflict* described in S2.2. Block granularity logging leaves the EXT4 journaling under frequent transaction conflict and subsequently under scalability failure.

Serial journal commit. In EXT4, journal commit is strictly serial activity. EXT4 allocates a separate thread for journal commit, *JBD thread*. JBD thread can commit the following journal transaction only after the preceding journal transaction becomes durable.

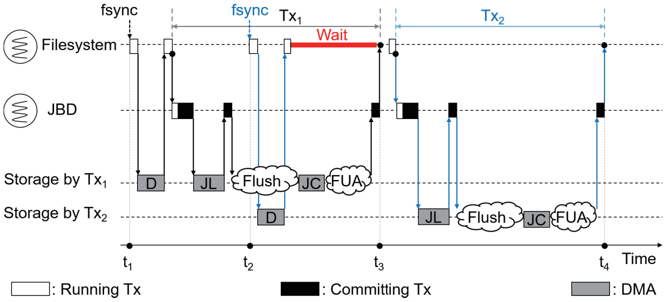


Figure 1: `fsync()` in EXT4

We illustrate the behavior of an `fsync()`. Let `D`, `JL`, and `JC` be file dirty data pages, journal log blocks and journal commit block. When an application thread calls `fsync()`, it writes the dirty data pages (`D`) to the storage, and then it wakes up the JBD thread. The JBD thread changes the transaction state from *running* to *committing* and writes the log blocks (`JL`) to the storage. Once all the log blocks are transferred (*i.e.*, DMA-ed) to the storage, the JBD thread writes the journal commit block (`JC`) to the storage with `REQ_PREFLUSH` and `REQ_FUA` [1] flags to ensure that the journal commit block is made durable only after the dirty data pages and the journal log blocks do. `REQ_PREFLUSH` flag instructs the storage controller to flush the writeback cache in a storage device before servicing the associated write command. `REQ_FUA` command writes the associated data block directly to the storage media bypassing the writeback cache of the storage. Once the write command for commit block returns, the JBD thread finishes committing a transaction.

Figure 1 illustrates the timing diagram of servicing two consecutive `fsync()`'s. The first `fsync()` and the second `fsync()` are issued at t_1 and at t_2, respectively. We mark the journal transactions for the preceding `fsync()` and the following `fsync()` as $\mathtt{Tx}_1$ and $\mathtt{Tx}_2$, respectively. JBD thread starts committing $\mathtt{Tx}_2$ (at t_3) only after it finishes committing $\mathtt{Tx}_1$.

2.2 Concurrency Control in Filesystem Journaling

To partly address the drawback of serial journal commit, EXT4 journaling adopts compound journaling and the shadow paging to increase its concurrency. The compound journaling commits multiple file operations with a single journal commit. The shadow paging allows the file operation and the journal commit operation to proceed in parallel while they share the same page cache entry. Compound journaling inevitably accompanies transaction lock-up phase when it needs to commit the running transaction. Journal commit operation should exclusively lock the page cache entry when it needs to write the page cache entry to the storage. Transaction lock-up and page granularity exclusive locking temporarily blocks the file operations and can severely interfere with the overall system performance. Let us explain the details of individual phases of journaling.

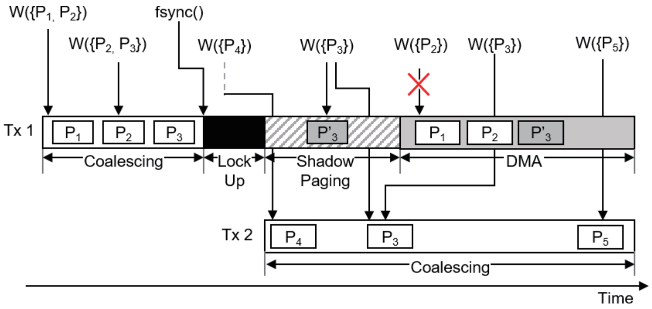


Figure 2: Dissection of EXT4 journaling phases

(i) Coalescing in Running Transaction. In coalescing phase, the application can modify the metadata and insert the associated page cache entry to the running transaction. EXT4 journaling adopts compound transaction which is also known as a group commit [12] to increase the throughput. In Figure 2, a file operation modifying page cache entries $\texttt{P}_1$ and $\texttt{P}_2$ and another file operation modifying page cache entries $\texttt{P}_2$ and $\texttt{P}_3$, sharing the commonly modified page cache entry $\texttt{P}_2$. EXT4 creates a compound transaction of $\texttt{P}_1$, $\texttt{P}_2$, and $\texttt{P}_3$.

(ii) Transaction Lock-Up in Running Transaction. When the JBD thread needs to commit the running transaction, the JBD thread stops issuing the journal handle to prohibit the new file operation to modify the running transaction. Then, it waits for the outstanding file operations which already have a journal handle to finish. Otherwise, starting the journal commit can be postponed indefinitely. When all file operations which already have a journal handle finish, JBD thread changes the transaction state from running to committing. We call this time period during which the JBD thread stops issuing the journal handle as *transaction lock-up* phase. Any file operations that update the metadata are blocked when the running transaction is in lock-up phase. In Figure 2, during the transaction lock-up phase, the file operation that modifies $\texttt{P}_4$ is blocked. The file operation wakes up when the lock-up phase of $\texttt{Tx}_1$ is released and adds $\texttt{P}_4$ to the new running transaction, $\texttt{Tx}_2$.

(iii) Shadow Paging in Committing Transaction. After the transaction state is changed to committing, JBD thread prepares the page cache entries for DMA transfer. EXT4 journaling adopts *very limited* form of *Shadow Paging* to handle the transaction conflict during this time interval. It allows only one shadow page and does not allow more multiples versions. When there occurs transaction conflict when the JBD thread prepares the page cache entries for DMA transfer, JBD thread creates the shadow copy of the conflict page and uses a shadow copy of the original page cache entry for DMA transfer [47]. With shadow paging, a file operation can modify the original page cache entry in the committing transaction without waiting for the completion of the transaction commit. EXT4 journaling can create up to only one shadow page. If two or more transactions attempt to update the same page, only one can proceed and the others are blocked until the associated page is committed to the storage. In Figure 2, during the shadow paging phase, the file operation tries to modify $\texttt{P}_3$, which is in the committing transaction, $\texttt{Tx}_1$. The file operation creates the shadow page, $\texttt{P'}_3$, and adds the original page cache entry, $\texttt{P}_3$ to the new running transaction, $\texttt{Tx}_2$.

(iv) DMA in Committing Transaction. When the log block of the committing transaction is transferred to the storage (DMA), the host establishes an exclusive lock on the associated page cache entry. This is to prohibit the file operation from blindly updating the page cache entry of the committing transaction that is being transferred and from migrating it to the running transaction compromising the atomicity of the journal commit. During DMA phase, any file operations that update the locked page cache entry are blocked. In Figure 2, the DMA phase is marked in gray. While a compound transaction of $\texttt{P}_1$, $\texttt{P}_2$, and $\texttt{P'}_3$ is under DMA, an attempt to modify page $\texttt{P}_2$ will be blocked. Because $\texttt{P'}_3$ is the shadow page and the shadow page is being transferred, the file operation which modifies $\texttt{P}_3$ is not blocked and modifies the original page cache entry, $\texttt{P}_3$. An attempt to modify a page cache entry, $\texttt{P}_5$, which is not in a committing transaction will successfully add the page to the new running transaction, $\texttt{Tx}_2$.

2.3 Existing Solutions to Scale Journaling

A number of approaches have been proposed to increase the concurrency in the filesystem journaling [15, 17, 24, 32, 45, 49]. They can be categorized with respect to the number of threads that are used to handle a single journal commit; (i) single-threaded journal commit and (ii) multi-threaded commit. They also can be categorized with respect to how they allocate the journal transaction in the filesystem; per-core basis or per-region basis. Table 1 summarizes the approaches in the existing scalable filesystem journaling techniques.

Filesystems	Concurrent Transactions		Multi-Threaded Commit
	Per-core	Per-region	
Z-journal [17]	○	○	
SpanFS [15]		○	
IceFS [24]		○	
MQFS [23]		○	
BarrierFS [49]			△
XFS [45]			○
iJournaling [32]	○		
ScaleFS [6]	○		

Table 1: Categories of existing scalable filesystems

In the concurrent transaction approach, the journaling filesystem allows multiple running transactions, multiple committing transactions or both, to proceed in parallel. In per-core basis approach, they allocate the transaction for each CPU core (per-core basis) [32] . In per-region basis approach, filesystem is partitioned into multiple regions and allocates dedicated journal area for each filesystem region [15, 17, 24]. In per-region basis approach, the journaling filesystem maintains the running transaction and/or committing transaction in per-region basis. The filesystem can commit the multiple transactions concurrently for each filesystem region. Per-region approach requires changing the on-disk layout of the existing filesystem partition [15, 17, 32]. In per-core approach, the transactions may conflict with each other, i.e. they modify the same page cache entry. Resolving the transaction conflict accompanies substantial overhead, e.g. [17] compromises `fsync()` durability or journal commit is subject to excessive tail latency [15].

In multi-threaded journal commit approach, the filesystem divides a journal commit operation into multiple phases and allocates the separate threads for handling each of the journal commit phases. With this multi-threaded organization, a thread can start processing the following journal transaction before the preceding journal commit finishes in pipelined manner. In XFS, one thread is responsible for making the journal transaction durable and the other thread is responsible for ensuring that all preceding transactions are durable after the journal transaction becomes durable [16]. In BarrierFS, one thread is responsible for issuing the IO requests for journal commit, and the other thread is responsible for making the journal transaction durable [49]. Both XFS and BarrierFS can start a new journal commit without waiting for the preceding journal commit to finish. In BarrierFS, the journal commit operation is serialized in most cases due to frequent transaction conflict.

To ensure the storage order between the log blocks and the journal commit block in committing a journal transaction, the filesystem interleaves the write requests for the log blocks and the write request for commit block with the FLUSH command. Recently, a number of works have been proposed order-preserving IO stack to mitigate the FLUSH overhead associated with ensuring the storage order in journal commit operation [9, 21, 23, 49]. The order-preserving IO stack consists of order-preserving block layer [23, 49] and order-preserving FTL [9, 21, 49]. Order-preserving FTL can be implemented via exploiting the cache-barrier command [49], via imposing a global sequence number on the IO commands [9] or via exploiting non-volatile cache at SSD [21]. These works show that order-preserving FTL can be realized without substantial overhead and renders the identical performance as legacy FTL.

3 Scalability of EXT4 Journaling

3.1 Workloads

We used four filesystem macro benchmarks – two variants of varmail (`varmail-shared` and `varmail-split`) in `filebench` [26], `dbench` [46], and `OLTP-Insert` [19] – to cover wide variety of real-world application behaviors. Each benchmark has a different mix of file operations (Table 2) and stresses various parts of the filesystem (Table 3).

Benchmarks	`create()`	`unlink()`	`write()`	`read()`	`fsync()`	`rename()`
`varmail`	7.7%	7.7%	15.4%	15.4%	15.4%	0%
`dbench`	16.6%	3.5%	8.6%	27.1%	5.2%	0.7%
`OLTP-Insert`	0%	0%	77.8%	12.2%	10.0%	0%

Table 2: Ratio of filesystem operations in benchmarks

Benchmarks	Directory contention	In-memory logging	On-disk logging
`varmail-shared`	High	Moderate	High
`varmail-split`	No	Moderate	High
`dbench`	No	Moderate	Moderate
`OLTP-Insert`	No	low	low

Table 3: Filesystem contention in benchmarks

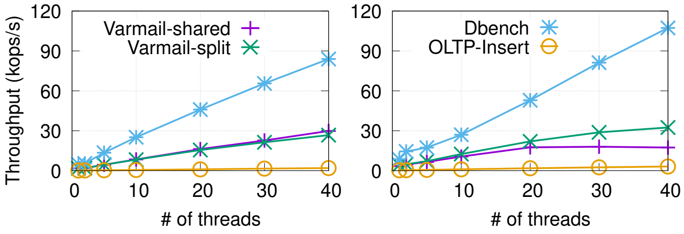
Mail server: varmail [26]. The varmail benchmark simulates the behavior of mail server. In the varmail workload, each thread repeats a set of `create()`, `unlink()`, and `fsync()` operations. Varmail is known for intensive `fsync()` calls. In the original `varmail` workload, all threads share the same directory yielding the lock contention on the shared directory. We call it `varmail-shared`. We modify the varmail workload so that each thread works on its own directory. We call it `varmail-split`. We use `varmail-split` how the filesystem journaling scales in the absence of the contention on the shared directory.

File server: dbench [46]. The dbench simulates the behavior of the fileserver. It is metadata-intensive workload calling `unlink()` and `rename()` followed by `fsync()` (with a `-sync-dir` option enabled). In `dbench`, `fsync()` calls account for 5.2% of all filesystem calls. Dbench calls `read()` and `write()` with various IO sizes; 4KB IO accounts for 60% of the `read()` and `write()`.

OLTP: OLTP-Insert on MySQL [19] OLTP-Insert simulates the server for online transaction processing. In this workload, `write()` followed by `fsync()` is frequently invoked. The write size ranges from 8KB to 32KB; 8KB write accounts for 81%. Among the four workloads, the contention (or transaction conflict) degree of this workload is the lowest. We use this workload to test the behavior of journaling under the circumstances that there is only little contention (or transaction conflict).

3.2 Scalability Results

We compare the performance of the four benchmarks under EXT4 and BarrierFS [49]. BarrierFS is the variant



(a) EXT4 (b) BarrierFS

Figure 3: Scalability of EXT4 and BarrierFS

of EXT4 which can commit multiple transactions concurrently. We used two SSDs – Samsung 860 Pro (MLC Flash, SATA interface) and 970 Pro (MLC Flash, NVMe interface) in this experiment. However, we omitted the results with 860 Pro since the performance trends on these two SSDs are almost identical. Please refer to S6.1 for the details of our evaluation setup.

As Figure 3 shows, the performance and scalability of both filesystems get worse as `fsync()` accounts for more dominant fraction of the entire system calls. The `dbench` which renders the least significant `fsync()` calls is the most performant and scalable.

As shown in Figure 3(b), BarrierFS increases the performance of `dbench`, `OLTP-Insert` and `varmail-split` by 28%, 61% and 21% against EXT4 in forty threads, respectively, thanks to its concurrent journaling scheme. However, the performance of `varmail-shared` is not at all scalable and moreover is even worse than EXT4. We found that the main problem is the transaction conflict. As presented in Table 3, `varmail-shared` has contention on a shared directory. When the modified shared directory pages are under DMA, the other concurrent transaction cannot make progress, significantly limiting scalability until the IO completes.

3.3 Analysis on Scalability Bottleneck

We examine the scalability bottlenecks in filesystem journaling with EXT4 performing serial journal commit and BarrierFS performing concurrent journaling. We identify four main components that affect the performance scalability in EXT4 and BarrierFS; *transaction conflict* (S3.3.1), *serial flush* (S3.3.1), *length of a transaction lock-up interval* (S3.3.2) and *coalescing degree of compound journaling* (S3.3.3). We present `varmail-shared` results only since the other workloads show the similar performance behavior.

3.3.1 Transaction Conflict

EXT4. Figure 4 shows the number of transaction conflicts (`varmail-shared`). The number of transaction conflicts – the number of file operations trying to modify the log blocks that are under DMA. At `varmail-shared`, the number of blocked file operations ranges from 6,360 to 15,809. It accounts for 4.7% of all file operations. Despite the shadow paging feature of EXT4 to resolve the transaction conflict, EXT4 journaling still suffers from a significant amount of transaction conflicts.

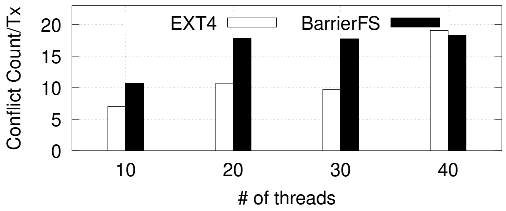


Figure 4: The average number of conflicts in a transaction (EXT4 and BarrierFS, `varmail-shared` workload)

BarrierFS. BarrierFS renders significantly worse performance than EXT4 in `varmail-shared` workload (Figure 3(a) vs. Figure 3(b)). We found that the concurrent journaling design of BarrierFS increases the number of transaction conflicts substantially and it causes the scalability meltdown. BarrierFS can start committing the following transaction before the preceding transaction commit finishes. Technically, there can be multiple committing transactions in-flight in BarrierFS. In reality, BarrierFS fails to commit multiple transactions concurrently. There are two reasons; *transaction conflict* and *serial flush*. We find that most journal transactions share some pages in common, *e.g.*, inode block and bitmap, and is subject to the transaction conflict [17]. The following journal transaction cannot be committed till the preceding transactions which it conflicts with are made durable at the storage. In BarrierFS, the flush thread issues the flush command of the following committing transaction only after the preceding transaction becomes durable. Even though BarrierFS commits multiple transactions concurrently, it flushes each of them with a separate flush command. Since each journal commit yields a separate flush at the storage device, the benefit of concurrent journaling design of BarrierFS is marginal.

Moreover, when running transactions are trying to modify log blocks under flush, they all are conflicted and blocked. Shadow paging (inherited from EXT4) does not help because it can create only one version in a certain condition. As a result, higher concurrency in committing transactions and limited shadow paging causes nearly 100% of file operations suffering from transaction conflicts in all threads.

3.3.2 Transaction Lock-up

One of the main causes of scalability failure in concurrent journaling is the extended lock-up interval.

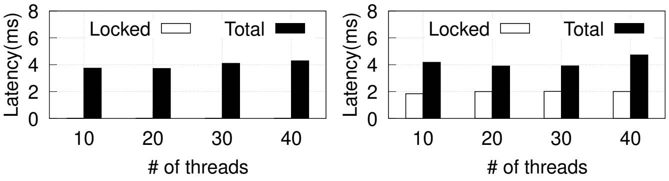
EXT4. In EXT4, the length of transaction lock-up interval is negligible as in Figure 5(a). In EXT4, the lock-up period is just a duration waiting for outstanding file operations to finish, which is very short in general.

Also, as Figure 6(a) shows, `fsync()` latency is high but the latency of `create()` and `unlink()` is still low. In other words, the short lock-up period does not interfere other file operations, `create()` and `unlink()`.

BarrierFS. In BarrierFS, the transaction lock-up latency accounts for approximately half of the entire transaction commit latency (Figure 5(b)). We found that transaction conflict and concurrent journaling negatively interfere with each other and significantly extend the transaction lock-up period. Because the running transaction waits for resolving of transaction conflict in LOCKED state.

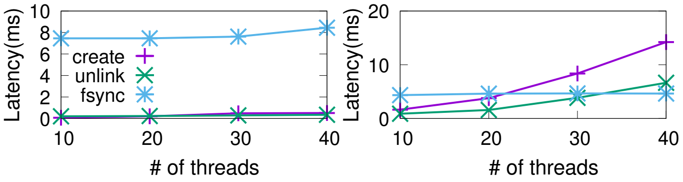
In both EXT4 and BarrierFS, JBD thread first places a running transaction in the *LOCKED* state when it starts committing the running transaction. There is a critical difference between EXT4 and BarrierFS from the aspect of the LOCKED state. In EXT4, when JBD thread places the running transaction in the LOCKED state, the running transaction is guaranteed to be free from transaction conflict. That is because, in EXT4, journal commit is strictly serial activity. In EXT4, the running transaction can be released from the LOCKED state if all outstanding filesystem operations finish.

In BarrierFS, the running transaction can be placed in the LOCKED state while the preceding journal commit is still in flight. BarrierFS can prematurely place the running transaction at the LOCKED state before the running transaction becomes free from the transaction conflict. BarrierFS waits to release the running transaction from the LOCKED state till all outstanding file operations finish and till all conflicts are resolved. As a result, a running transaction stays at the LOCKED state in much longer interval in BarrierFS than in EXT4.



(a) EXT4 (b) BarrierFS

Figure 5: Transaction lock-up interval in `varmail-shared`



(a) EXT4 (b) BarrierFS

Figure 6: Latency of `unlink()`, `create()` and `fsync()` in `varmail-shared`

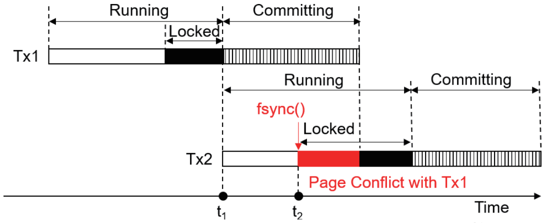


Figure 7: Excessive Lock-Up overhead in Concurrent Journaling (BarrierFS)

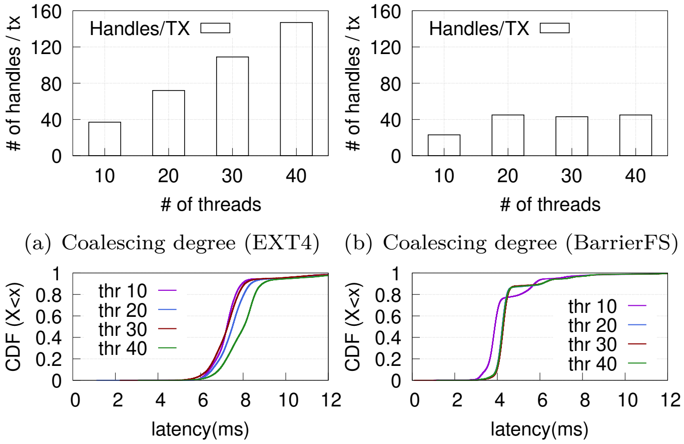
Figure 7 illustrates this situation. The running transaction, `Tx`$_2$, is created at t_1. The application calls `fsync()` at t_2. `Tx`$_2$ is placed on the LOCKED state immediately without waiting for the current committing transaction `Tx`$_1$ is made durable. If `Tx`$_2$ conflicts with `Tx`$_1$ (in most cases it does), `Tx`$_2$ can be released from the LOCKED state only after `Tx`$_1$ is committed to the storage.

3.3.3 Limited Coalescing Degree

The key ingredient that governs the performance scalability of the filesystem journaling is the *coalescing degree* of the journal transaction – the number of filesystem operations in a journal transaction.

EXT4. EXT4 scales well in `varmail-shared` workload (Figure 3). Ironically, the strict serial nature of EXT4 journaling actually helps itself to increase the coalescing degree of the compound journaling. EXT4 can start committing the running transaction only when the preceding journal commit finishes. When the journal commit is in progress, all updates associated with the incoming file operations are inserted at the running transaction. Therefore, there is a higher coalescing opportunity as the number of threads increases. Figure 8(a) confirms that the number of handles (*i.e.*, file operations) in a transaction increases linearly with the number of threads. At the same time, we observe that the journal commit latency increases with the number of threads. This is because journal transaction tends to get larger as the number of threads increases. As shown in Figure 8(c), median and 99.99% latencies increase 11% and 7%, respectively, from 10 to 40 threads.

BarrierFS. BarrierFS fails to scale in `varmail-shared` workload (Figure 3) due to its *limited coalescing degree*. This is because BarrierFS places the running transaction into LOCKED state prematurely and leaves less chance to coalesce the multiple file operations into a single journal transaction. Figure 8(b) confirms that in BarrierFS the coalescing degree remains the same while the number of threads increases. Since the coalescing degree does not increase, the latency of journal commit remains the same irrespective of the increase in the number of threads (Figure 8(d)).



(a) Coalescing degree (EXT4) (b) Coalescing degree (BarrierFS)

(c) `fsync()` latency (EXT4) (d) `fsync()` latency (BarrierFS)

Figure 8: Coalescing degree and CDF of `fsync()` latency in EXT4 and BarrierFS for `varmail-shared` workload

4 Design

In this section, we present the design of CJFS, a *Concurrent Journaling Filesystem*. CJFS consists of four key technical ingredients; (1) *dual thread journaling* (S4.1), (2) *multi-version shadow paging* (S4.2), (3) *opportunistic coalescing* (S4.3), and (4) *compound flush* (S4.4) – to overcome all the bottlenecks discussed in S3.3 and to scale filesystem journaling.

4.1 Dual Thread Journaling

For concurrent journaling, we separate the journal commit procedure into two phases, the commit phase and the flush phase and allocate separate threads, namely *commit thread* and the *flush thread*, for each phase. The commit thread is responsible for issuing the write requests for journal transaction to the storage. Once this completes, the storage device sends an interrupt to the host notifying about the completion of servicing the requests. The flush thread is responsible for making the log blocks and the commit block durable. Once the interrupt arrives, the flush thread wakes up and issues the flush command to the storage to make the log blocks and the commit block durable. Via separating the commit thread and the flush thread, CJFS can commit the following transaction without waiting for the preceding journal commit to finish.

Figure 9 illustrates the mechanism of Dual Thread Journaling. CJFS maintains a single running transaction. In `fsync()`, the flush thread waits till all dirty pages, log blocks, and the commit block are transferred to the disk. Once this completes, it issues the flush command to the storage. Our journaling module leverages the cache barrier command [14, 36, 49], which efficiently preserves the partial order between the issue order and the persist order in a commodity storage device.

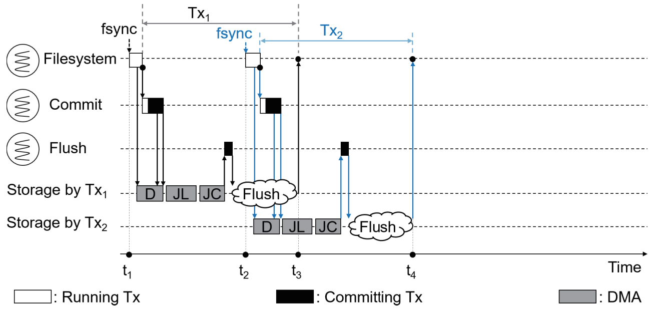


Figure 9: Concurrent Transaction Commit in Dual Thread Journaling. CJFS performs $\texttt{Tx}_1$'s flush phase and $\texttt{Tx}_2$ commit phase concurrently

4.2 Multi-Version Shadow Paging

Most filesystems cluster the filesystem metadata together in their filesystem partition. This is to exploit the spatial locality of the disk access. The filesystem operations, e.g.,`create()` or `write()`, access a few common blocks which contain the popular filesystem metadata, e.g., the allocation bitmap or inode.

EXT4 adopts the page granularity physical logging and uses the original page cache entry. When it commits the journal transaction, it establishes an exclusive lock on the page cache entry associated with the journal transaction till the journal transaction becomes durable. Transaction conflict is particularly harmful to concurrent journaling since it serializes the journal commits. If the transaction conflict happens frequently, the concurrent journaling of CJFS becomes barely effective and resorts to serial journal commit as in EXT4.

To address the transaction conflict, we propose *Multi-Version Shadow Paging (MVSP)*. In multi-version shadow paging, when the commit thread starts the journal commit, it creates the shadow copy of all pages in the journal transaction. In committing the journal transaction, the commit thread uses the shadow copy of each page in the transaction for transferring the journal transaction to the storage device instead of using the original one. Since the journaling module uses the shadow page for the journal commit, the subsequent file operation can update the original page.

There can be multiple shadow copies for a given page cache entry. Assume that the shadow copy of page `P` is being committed to the storage. An application updates the `P` to `P′` and calls `fsync()`. Then, the commit thread creates the shadow copy of `P′` and commits the shadow copy of `P′` to the storage. While the shadow copy of `P′` is being transferred by the journal commit, another application may update the `P′` to `P″` and calls `fsync()`. Then, there exist two shadow copies for `P`, `P′` and `P″`. CJFS defines the maximum number of shadow pages that can be associated with a single page. The maximum number of versions is an administrator-configurable parameter

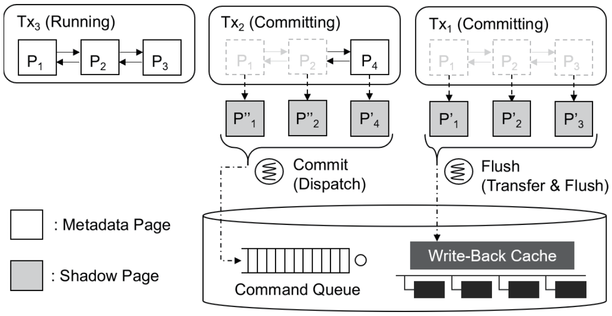


Figure 10: Multi-Version Shadow Paging

and initialized when CJFS is mounted.

Figure 10 illustrates the behavior of CJFS when the transaction conflict exists. Tx_1, Tx_2 and Tx_3 are created in order. Each of them is in a different phase. There are one running transaction, Tx_3, and two committing transactions, Tx_2 and Tx_1. Tx_1 has three pages $\texttt{P}_1$, $\texttt{P}_2$ and $\texttt{P}_3$. When the commit thread commits Tx_1, CJFS creates the shadow copies $\texttt{P}'_1$, $\texttt{P}'_2$ and $\texttt{P}'_3$ for the pages in Tx_1. The subsequent filesystem operation updates $\texttt{P}_1$, $\texttt{P}_2$ and $\texttt{P}_4$. Then, the filesystem triggers another journal commit. The following transaction, Tx_2, consists of $\texttt{P}_1$, $\texttt{P}_2$ and $\texttt{P}_4$. In committing Tx_2, the commit thread creates the shadow copies $\texttt{P}''_1$ (second shadow copy of $\texttt{P}_1$), $\texttt{P}''_2$ (second shadow copy of $\texttt{P}_2$) and $\texttt{P}'_4$ for each page in Tx_2. Two transactions, Tx_1 and Tx_2 are being committed to the storage. Subsequent file operations update $\texttt{P}_1$, $\texttt{P}_2$ and $\texttt{P}_3$. Since these pages are available for the update, the file operations update these pages and insert them to the running transaction.

Multi-version shadow paging in CJFS is a variant of versioning which is widely used in transaction concurrency control [18,30,51]. Multi-version shadow paging of CJFS is different from the versioning in Copy-On-Write filesystems [20,37–40]. These filesystems retain the history of updates for individual file blocks to make the IO workload sequential and/or to construct the filesystem snapshot easily.

4.3 Opportunistic Coalescing

CJFS pre-allocates a fixed number of pages for shadow paging. Since the number of shadow pages is limited, the transaction conflict can still occur if all pre-allocated shadow pages are used to hold the logs. If the transaction conflict occurs, the running transaction is put in the LOCKED state and *all* subsequent file operations that modify the filesystem state are blocked. To resolve this problem, we propose the *Opportunistic Coalescing*. The proposed opportunistic coalescing shares the same idea with `try_lock` [4].

Algorithm 1 shows the pseudo-code for opportunistic coalescing. At first, the commit thread puts the running transaction at the LOCKED state (Line 4), After the running transaction is put at the LOCKED state, the commit thread waits for the outstanding file operation which already has a journal handle to finish (Line 6). When all outstanding file operations finish, the commit thread checks if there exists any conflict (Line 8). If there exists a conflict, the commit thread places the transaction back to the RUNNING state and is blocked (Line 9-11). The running transaction can continue accommodating the newly incoming log blocks while the commit thread is blocked. Each time when the transaction commit finishes, the flush thread wakes up the commit thread. When the commit thread wakes up, it checks if the running transaction is free from the conflicts. If it is free from the conflicts, it changes the state of the transaction to LOCKED state again (Line 12).

Algorithm 1: Opportunistic Coalescing

```
1  function journal_commit_transaction(journal)
2      while true do
3          tx = journal→running_tx;
4          tx → state = LOCKED;
5          if outstanding system calls > 0 then
6              wait (outstanding system calls == 0);
7          end
8          if transaction conflict then
9              tx → state = RUNNING;
10             wakeup(user thread waiting on LOCKED);
11             wait(preceding transaction to commit);
12             continue;
13         end
14         break;
15     end
16     journal → committing_tx = tx;
17     journal → running_tx = NULL;
18     submit_bio_tx(tx);
19     insert_committing_tx_list(tx);
20 end
```

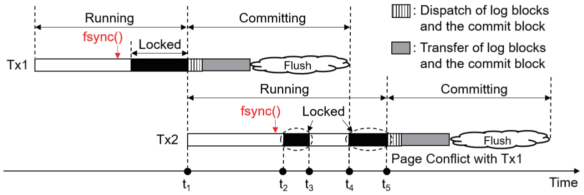


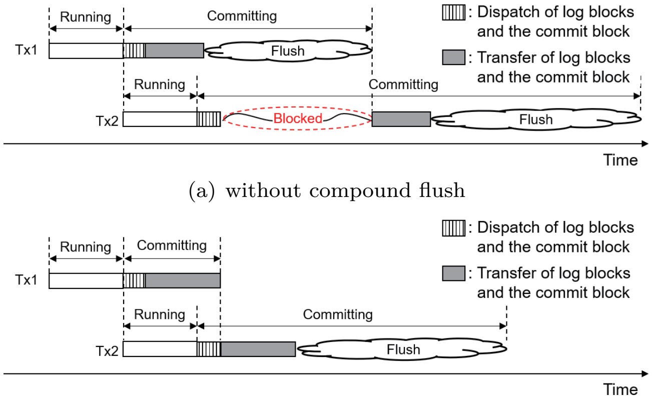
Figure 11: Illustration of Opportunistic Coalescing

Figure 11 illustrates how the opportunistic coalescing works. There arrive two consecutive transactions ($\texttt{Tx}_1$ and $\texttt{Tx}_2$). In Figure 11, $\texttt{Tx}_2$ is put into LOCKED state twice; at t_2 and at t_4. Tx_2 is in `RUNNING` state during the period between two `LOCKED` states. After the state of the running transaction becomes `RUNNING` state, all pending file operations, which were blocked waiting for the journal handle, are issued the journal handles. With Opportunistic Coalescing, CJFS can coalesce larger number of file operations into the running transaction.

4.4 Compound Flush

CJFS splits the journal commit operation into two phases; (i) transferring the log blocks and the commit block (commit thread) and (ii) making them durable (flush thread). For journaling of CJFS to work in a fully concurrent fashion, both the commit thread and the flush thread should be able to handle the associated tasks in a concurrent manner. In CJFS, the commit thread handles the transaction concurrently; it can commit the following transaction while the preceding transaction is in-flight. However the flush thread handles the transaction in serial fashion; it can flush the following transaction only after the preceding transaction is flushed.

To ensure that the journal transactions are made durable in order, the flush thread issues the flush command for the following transaction only after the flush command for the preceding transaction returns. As a result, the behavior of the flush thread is serial, which makes the concurrent journal mechanism of CJFS only partially complete. Figure 12(a) illustrates the concurrent journaling with serial flush. Commit thread can start committing the following transaction $\texttt{Tx}_2$ before the preceding transaction $\texttt{Tx}_1$ commit finishes. However, the flush thread can flush the following transaction $\texttt{Tx}_2$ only after the transaction $\texttt{Tx}_1$ is flushed to the storage.


Running
Committing
Tx1
Flush
▥: Dispatch of log blocks and the commit block
▨: Transfer of log blocks and the commit block
Running
Committing
Tx2
Blocked
Flush
Time

(a) without compound flush

Running
Committing
Tx1
▥: Dispatch of log blocks and the commit block
▨: Transfer of log blocks and the commit block
Running
Committing
Tx2
Flush
Time

(b) with compound flush

Figure 12: Comparison of the flush procedure with and without Compound Flush

To address the serial flush issue of CJFS, we propose *Compound Flush*. Compound Flush exploits the cache barrier command [14, 36]. *Compound Flush* works as follows. When the flush thread is about to send the flush command, it checks if there exist any following committing transactions. If following committing transaction does not exist, it sends the flush command. If the following committing transaction exists, it sends the cache barrier command instead. *Compound Flush* delegates the task of persisting the transaction to the following transaction commit request. An `fsync()` returns only when the associated journal transaction becomes durable. To prevent the *Compound Flush* from delaying the transaction commit indefinitely, we limit the number of transactions that can be flushed with a single flush command. When the number of transactions waiting for the flush reaches its limit or when there is no more committing transactions in-flight, the flush thread sends a flush command to the storage. With cache barrier commands, the storage controller ensures that the log blocks of the individual transactions are made durable in order. When the flush command returns, the flush thread wakes up all application threads that are waiting for their `fsync()` to return.

Figure 12(b) illustrates how Compound Flush works. When the flush thread finishes transferring the transaction $\texttt{Tx}_1$, the flush thread starts transferring the transaction $\texttt{Tx}_2$ instead of calling flush for flushing the transaction $\texttt{Tx}_1$. When the flush thread finishes transferring the transaction $\texttt{Tx}_2$, it finds that there are no other committing transactions in flight. Then, it calls flush to make the transaction $\texttt{Tx}_1$ and transaction $\texttt{Tx}_2$ durable.

5 Discussion

We compare CJFS with the closest filesystem of this sort, BarrierFS [49]. Dual Thread Journaling of CJFS and Dual Mode Journaling of BarrierFS are similar in that both allocate separate threads for transaction commit and transaction flush, respectively. However, BarrierFS's dual thread design is to efficiently support the two journaling modes; "ordered" mode and the "durability" mode. It is not designed for concurrent journaling.

There are three key differences between CJFS and BarrierFS. First is how to handle the transaction conflict. BarrierFS cannot commit the running transaction if the running transaction conflicts with any of the ongoing committing transactions. CJFS can commit the running transaction even if there is a conflict. CJFS uses multi-version shadow paging to resolve the conflict between the running transaction and the committing transactions. The second is how to handle the transaction lock-up. In BarrierFS, transaction lock-up is non-preemptive. Once the running transaction is locked-up, it waits for all committing transactions that it conflicts with to finish. In CJFS, transaction lock-up is preemptive. When a running transaction is locked-up, CJFS checks if the running transaction conflicts with any of committing transactions. If it finds a conflict, the running transaction is unlocked. The third is how to flush the committing transactions. For a set of committing transactions that proceed concurrently, BarrierFS flushes each of them separately. CJFS flushes a number of concurrent transactions together, reducing the flush overhead substantially.

The Opportunistic Coalescing and Compound Flush can be used in the other journaling filesystems such as

XFS [45]. In journal commit, XFS copies the logs in the log list to the log buffer and then flushes the log buffer to the log area in storage. With Opportunistic Coalescing, XFS can insert more logs to the log list by releasing the lock on the log list. With Compound Flush, XFS can flush the multiple log buffers with a single flush command. Dual-Thread Journaling and Multi-Version Shadow Paging are already used widely in other filesystems [20,37,38,45,49] or DBMS [18,29,30].

6 Evaluation

6.1 Experiment Setup

We implemented CJFS [49] on Linux Kernel 5.18.18. We used a 40-core server (two Intel Xeon Gold 6230 processors and 512 GB DRAM) and Samsung 970 Pro SSD (MLC Flash, NVMe) for our experiment. We assume that the SSD supports cache barrier command as a mobile flash products (eMMC) support cache barrier command [3,5]. They do not render any significant performance deficiency against the ones without cache barrier support. Also, previous studies [9,49] showed the FTL overhead of supporting the cache barrier command is less than 2%. Given all these, we carefully believe that it is reasonable to assume that SSD can support cache barrier command without significant performance overhead. We compare CJFS against BarrierFS [49], SpanFS [15], Vanilla EXT4, and EXT4 with Fast-Commit [42]. We used three macro benchmarks; `varmail` for mail server, `dbench` for file server, and `OLTP-Insert` on `MySQL`. Please refer to S3.1 for details of the benchmarks. We set the maximum number of versions in CJFS to five[1].

6.2 Effect of Individual Techniques

Dual Thread Journaling. We examine the command queue depth of the JBD thread (Figure 13) at `varmail-shared`. In result, CJFS shows higher command queue depth than EXT4 and BarrierFS. Because of the serial transaction commit in EXT4, the maximum queue depth of EXT4 is one. Although BarrierFS adopts dual thread design, its maximum queue depth is two due to the transaction conflict. CJFS fully exploits the queue depth of the storage with Dual Thread design. While BarrierFS suffers from the transaction conflict, CJFS resolves the transaction conflict with Multi-Version Shadow Paging. The performance effect of the Multi-Version Shadow Paging is described separately. With the higher queue depth, it renders higher SSD IO utilization.

Multi-Version Shadow Paging. We vary the maximum number of versions in MVSP and examine the throughput, the latency, and the number of conflicts per transaction. We examine the effectiveness of MVSP under three different maximum numbers of versions; one (EXT4 and BarrierFS), three (noted as CJFS-V3), and five (noted as CJFS-V5). Note that EXT4 and BarrierFS can have up to one shadow page. In the absence of any versioning feature, BarrierFS is subject to frequent transaction conflicts. Transaction conflict becomes more harmful when the filesystem allows the concurrent journal commit (BarrierFS) since it extends the transaction lock-up interval. As a result, BarrierFS renders worse performance than EXT4. With forty threads, the performance of BarrierFS is 60% of EXT4.

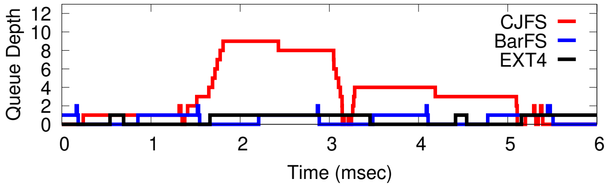


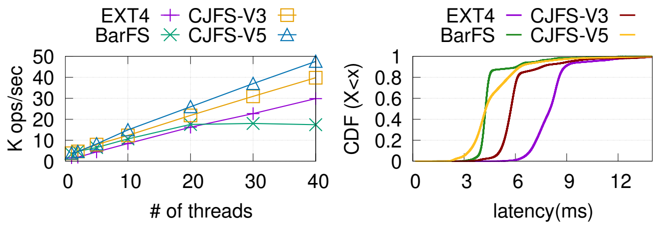
Figure 13: Queue depth of JBD thread in CJFS, BarrierFS (BarFS), and EXT4

Multi-version shadow paging brings additional memory pressure and the overhead of preparing the shadow page. The total memory pressure for multi-version shadow paging corresponds to the sum of the shadow pages associated with the concurrent transactions. The average transaction size is 33 blocks in `varmail-shared` (40 threads). CJFS with five versions (CJFS-V5) consumes 660KByte (5*33*4KByte) additional memory. According to our physical measurement, preparing the shadow page takes approximately 80 usec for a transaction in `varmail-shared` (40 threads). The average transaction commit latency decreases from 4.4 msec in EXT4 to 2.2 msec in CJFS in `varmail-shared` (40 threads). In CJFS, the reduction in the journal commit latency far outweighs the overhead of shadow paging.

We examine the `fsync()` latency of four filesystems (Figure 14(b)). CJFS and BarrierFS yield the shortest latency. The average latency of EXT4, BarrierFS CJFS (V3), and CJFS (V5) are 8.1ms, 4.6ms 6.1ms, and 4.7ms, respectively. CJFS yields the shortest tail latency (99.9%) among the four filesytstems. The tail latency of EXT4, BarrierFS, CJFS (V3), and CJFS (V5) are 17.0ms, 13.5ms, 16.3ms and 11.8ms, respectively. BarrierFS and CJFS-V5 has similar latency but CJFS-V5 has a better throughput than BarrierFS because of the transaction conflict. In BarrierFS, file operations are blocked when the transaction conflict occurs. However, CJFS-V5 is free from the transaction conflict. File operations return without waiting for the transaction conflict.

We examine the number of conflicting blocks. The average number of conflicted blocks in a transaction is eleven or larger in EXT4 and BarrierFS but less than two in CJFS (Figure 15). The number of conflicts per

[1] We found that we do not need more than five shadow pages to eliminate transaction conflict in our experiment setup.



(a) Throughput (b) `fsync()` latency (40 threads)

Figure 14: Throughput and Latency of `varmail-shared`: CJFS, BarrierFS (BarFS), and Vanila EXT4 (EXT4)

transaction is inversely proportional to the benchmark performance. CJFS with five versions (CJFS-V5) outperforms EXT4 1.7× at 40 threads.

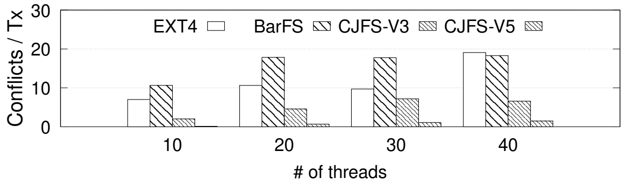
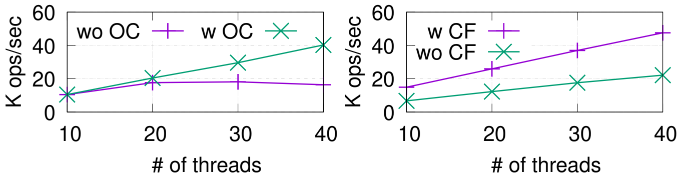


Figure 15: Average number of conflicts per transaction

Opportunistic Coalescing. Opportunistic Coalescing improves the filesystem performance by 2.5× (Figure 16(a)). With Opportunistic Coalescing, the coalescing degree of the journal transaction increases by 3.3× (Figure 17).



(a) Opportunistic Coalescing (b) Compound Flush

Figure 16: Effect of Opportunistic Coalescing and Compound Flush for `varmail-shared` in CJFS

Compound Flush. We ran `varmail-shared` to see the performance impact of Compound Flush. We set the maximum version number to five. Figure 16(b) shows that Compound Flush improves throughput up to 2.14×. By merging the multiple flush commands into one, Compound Flush reduces the average `fsync()` latency from 11.8ms to 4.7ms.

6.3 Macro Benchmarks

Mail server: `varmail-shared` and `varmail-split`. Figure 18(a) and Figure 18(b) shows the throughput of `varmail-shared` and `varmails-split`, respectively. Since all threads share the same directory in `varmail-shared`, the transaction conflicts occur much more frequently. For `varmail-shared`, CJFS outperforms EXT4 and BarrierFS by 62% and 173%, respectively. For `varmail-split`, the throughput of CJFS is 82% and 15% higher than that of EXT4 and BarrierFS, respectively. CJFS manifest itself in `varmail-shared` because MVSP of effectively handles the transaction conflict. In `varmail-shared`, BarrierFS becomes subject to the severe performance degradation due to frequent transaction conflicts.

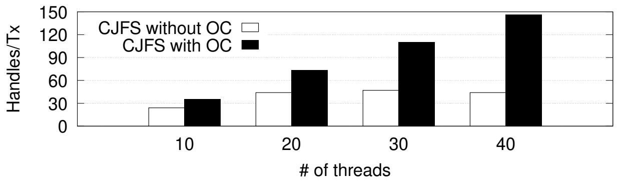
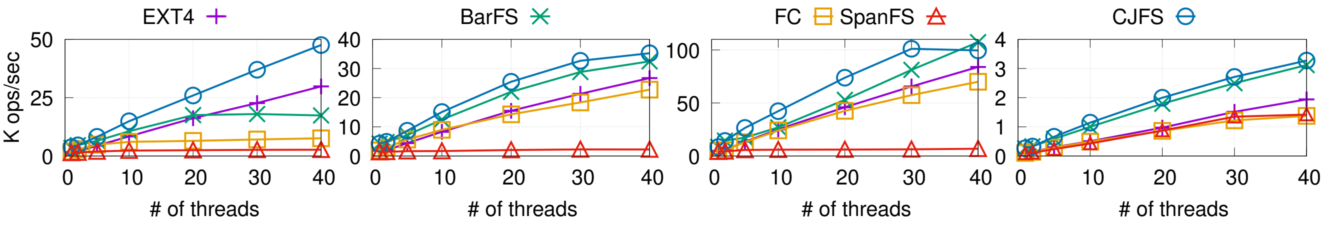


Figure 17: Comparison of coalescing degree with and without Opportunistic Coalescing for `varmail-shared`

File Server: `dbench`. As Figure 18(c) shows, CJFS increases throughput 68% over EXT4. `Dbench` does not have the directory contention (similar to `varmail-split`) and it is less `fsync()`-heavy than `varmail-shared`, incurring less transaction conflicts. Hence BarrierFS scales better in `dbench` than in `varmail` workload.

`OLTP-Insert` on `MySQL` : Here, the transaction conflict rarely occurs. CJFS scales well even when there is little or no transaction conflict. As Figure 18(d) shows, CJFS increases throughput up to 2.25× over EXT4 in ten threads. Moreover, CJFS increases the throughput by 15% compared to BarrierFS in ten threads.

Analysis : Fast commit [42] uses metadata-granularity physical logging. Despite its data structural elegance, Fast commit yields second to lowest throughput among the five. Fast commit trades the `fsync()` throughput with the `fsync()` latency. Due to its finer transaction granularity, Fast commit tends to make the smaller journal transaction. As a result, the `fsync()` latency becomes shorter in Fast commit. However, we observe that the number of flushes, i.e., the number of journal commits, increases significantly when EXT4 employs Fast commit. As a result, in terms of journaling throughput and scalability, Fast commit in EXT4 leaves substantial room for improvement. Fast commit is particularly detrimental to the journaling performance when there are a large number of threads. SpanFS [15] yields the worst performance among the five due to its serial journal commit. SpanFS defines the running transaction for each filesystem region. In SpanFS, these transactions can be committed in parallel. However, when the two or more transactions modify the shared filesystem metadata, e.g. root directory, the following running transaction can only be committed after the preceding running transaction is made durable. When there exists multiple concurrent running transactions (SpanFS), the performance becomes actually worse



(a) varmail-shared (b) varmail-split (c) dbench (d) OLTP-Insert

Figure 18: Throughput: EXT4, BarrierFS (BarFS), Fast commit (FC), SpanFS, and CJFS

than when there allows only one running transaction (EXT4). This is because SpanFS creates large number of small running transactions and all small running transactions are committed in serial fashion. On the other hand, EXT4 commits a large amount of the filesystem updates with a single running transaction.

6.4 Crash Consistency

CJFS uses the same on-disk structure and the recovery routine with EXT4. We use `CrashMonkey` [25] to examine if CJFS recovers the filesystem properly under unexpected system crashes. `Crashmonkey` generates a number of crash scenarios and checks if the filesystem recovers correctly. We use two scenarios, `rename_root_to_sub` and `create_delete`. CJFS passed all 10,000 test cases. We also generate sudden-power-off condition and examine if CJFS recovers the filesystem state into a consistent one. We confirmed that the recovery routine of CJFS correctly replays the transactions in the journal region and places the filesystem state into the consistent state.

7 Related Work

Multiple journal regions. IceFS [24] creates multiple journal regions in a filesystem partition for better isolation. ScaleFS [6], SpanFS [15], and Z-journal [17] manage multiple (or per-core) journal regions to reduce contention on journaling and to achieve high scalability. However, they still serially commit a journal transaction for each journal region and they are subject to transaction conflict when multiple threads access the same storage region. Note that Z-journal compromises the durability of `fsync()` for scalability.

Per-core running transaction. ScaleFS [6] and MQFS [23] maintain per-core running transaction to avoid contention in concurrent journaling. While these works can concurrently commit the multiple transactions in different cores, they commit the transactions in serial fashion in each core. In addition, while this approach minimizes the contention on journaling, it also loses the chance of transaction coalescing, which we found critical in achieving high performance and scalability.

Parallel journal commit. BarrierFS [49] and XFS [45] process a journal transaction commit in a separate thread to make a single journal commit parallel. However, BarrierFS serializes the journal commit when there is a conflict between a running transaction and committing transactions. Also, XFS suffers from excessive flush calls for guaranteeing a write order or a durability [16].

Reducing flush overhead. There have been efforts to reduce costly flush overhead in filesystem journaling. RFLUSH [52] specifies an `fsync()` range and iJournaling [32] performs per-file journaling. IRON filesystem [34] omits flushing the journal commit block by using transactional checksum. BarrierFS [10] leverages a cache barrier command to reduce flush overhead.

Soft Updates. Soft Updates [11,27,41] is an alternative to the filesystem journaling. It enforces write ordering with an expensive transfer-and-flush mechanism [49]. CJFS can guarantee the storage order without using transfer-and-flush mechanism.

8 Conclusion

We propose CJFS, Concurrent Journaling Filesystem. CJFS overcomes the scalability limitations of the heavy-weight EXT4 journaling mechanism with four novel techniques, namely Dual Thread Journaling, Multi-Version Shadow Paging, Opportunistic Coalescing, and Compound Flush. At a high level, CJFS parallelizes the journaling activity (Dual Thread Journaling) and avoids a page under IO being a bottleneck (Multi-Version Shadow Paging). Whenever the contention is inevitable, CJFS actively lowers the overhead by coalescing concurrent requests at thread level (Opportunistic Coalescing) and storage device level (Compound Flush). Our extensive evaluation shows CJFS achieves the significant throughput and latency improvement with multicore scalability and high storage device utilization against the state-of-the-art filesystems.

Acknowledgements We are deeply indebted to our shepherd, Yu Hua, for helping shaping the final version of this paper. We are also grateful to the anonymous reviewers for their comments. This work was supported by NRF, Korea (grant No. NRF-2020R1A2C3008525), and Samsung Electronics (HiPER SCOUT).

References

[1] block: update documentation for REQ_FLUSH / REQ_FUA. https://patchwork.kernel.org/project/dm-devel/patch/20100826095413.GA9750@lst.de/.

[2] The docker containerization platform. https://www.docker.com/.

[3] eMMC5.1 solution in SK hynix. https://www.skhynix.com/kor/product/nandEMMC.jsp.

[4] pthread_mutex_trylock(3) - Linux man page. https://linux.die.net/man/3/pthread_mutex_trylock.

[5] Toshiba Expands Line-up of e-MMC Version 5.1 Compliant Embedded NAND Flash Memory Modules. http://toshiba.semicon-storage.com/us/company/taec/news/2015/03/memory-20150323-1.html.

[6] Srivatsa S Bhat, Rasha Eqbal, Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. *Scaling a file system to many cores using an operation log.* In *Proc. of 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[7] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. *An Analysis of Linux Scalability to Many Cores.* In *Proc. of 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[8] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. *OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019*, 2014.

[9] Yun-Sheng Chang and Ren-Shuo Liu. *OPTR: Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface.* In *Proc. of 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[10] Jonathan Corbet. *Barriers and journaling filesystems.* http://lwn.net/Articles/283161/, 2010.

[11] Gregory R. Ganger, Marshall K. McKusick, Craig A. N. Soules, and Yale N. Patt. *Soft updates: a solution to the metadata update problem in file systems. ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.

[12] Robert Hagmann. *Reimplementing the Cedar file system using logging and group commit.* In *Proc. of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, 1987.

[13] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. *I/O stack optimization for smartphones.* In *Proc. of 2013 USENIX Annual Technical Conference (ATC)*, 2013.

[14] JEDEC Standard JESD84-B51. *Embedded Multi-Media Card (eMMC) Electrical Standard (5.1).* 2015.

[15] 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 *Proc. of 2015 USENIX Annual Technical Conference (ATC)*, 2015.

[16] Dohyun Kim, Kwangwon Min, Joontaek Oh, and Youjip Won. *ScaleXFS: Getting scalability of XFS back on the ring.* In *Proc. of 20th USENIX Conference on File and Storage Technologies (FAST)*, 2022.

[17] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. *Z-Journal: Scalable Per-Core Journaling.* In *Proc. of 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[18] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youji Won. *Resolving Journaling of Journal Anomaly in Android I/O:Multi-Version B-tree with Lazy Split.* In *Proc. of 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[19] Alexey Kopytov. Sysbench manual. *MySQL AB*, pages 2–3, 2012.

[20] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. *F2FS: A New File System for Flash Storage.* In *Proc. of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[21] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. *Write Dependency Disentanglement with HORAE.* In *Proc. of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[22] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. *Max: A Multicore-Accelerated File System for Flash Storage.* In *Proc. of 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[23] Xiaojian Liao, Youyou Lu, Zhe Yang, and Jiwu Shu. *Crash Consistent Non-Volatile Memory Express.* In *Proc. of 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[24] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. *Physical disentanglement in a*

container-based file system. In *Proc. of 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[25] Ashlie Martinez and Vijay Chidambaram. *Crashmonkey: A framework to systematically test file-system crash consistency.* In *Proc. of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2017.

[26] Richard McDougall and Jim Mauro. *FileBench.* http://www.nfsv4bat.org/Documents/nasconf/2004/filebench, 2005.

[27] Marshall K. McKusick and Gregory R. Ganger. *Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem.* In *Proc. of 1999 USENIX Annual Technical Conference (ATC)*, 1999.

[28] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. *Understanding Manycore Scalability of File Systems.* In *Proc. of 2016 USENIX Annual Technical Conference (ATC)*, 2016.

[29] Erik T Mueller, Johanna D Moore, and Gerald J Popek. *A nested transaction mechanism for LOCUS.* In *Proc. of 9th ACM Symposium on Operating Systems Principles (SOSP)*, 1983.

[30] Shojiro Muro, Tiko Kameda, and Toshimi Minoura. *Multi-version concurrency control scheme for a database system. Journal of Computer and System Sciences*, 29(2):207–224, 1984.

[31] Jiaxin Ou, Jiwu Shu, and Youyou Lu. *A high performance file system for non-volatile main memory.* In *Proc. of 11th European Conference on Computer Systems (EuroSys)*, 2016.

[32] Daejun Park and Dongkun Shin. *iJournaling: Fine-grained journaling for improving the latency of fsync system call.* In *Proc. of 2017 USENIX Annual Technical Conference (ATC)*, 2017.

[33] Jeoungahn Park, Taeho Hwang, Jongmoo Choi, Changwoo Min, and Youjip Won. *LODIC: Logical Distributed Counting for Scalable File Access.* In *Proc. of 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[34] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. *IRON File Systems.* In *Proc. of 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[35] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. *HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM.* In *Proc. of 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[36] Nikilesh Reddy. *Using Cache barriers in lieu of REQ_FLUSH and REQ_FUA for emmc 5.1.* https://lists.openwall.net/linux-ext4/2015/09/15/5.

[37] Ohad Rodeh, Josef Bacik, and Chris Mason. *BTRFS: The Linux B-tree filesystem. ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[38] Ohad Rodeh and Avi Teperman. *zFS-a scalable distributed file system using object disks.* In *Proc. of 20th IEEE/11th Conference on Mass Storage Systems and Technologies (MSST)*, 2003.

[39] Mendel Rosenblum and John K Ousterhout. *The design and implementation of a log-structured file system. ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[40] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. An implementation of a log-structured file system for unix. In *Proc. of 1993 USENIX Winter*, 1993.

[41] Margo I. Seltzer, Gregory R. Ganger, Marshall K. McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. *Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems.* In *Proc. of 2000 USENIX Annual Technical Conference*, 2000.

[42] Harshad Shirwadkar. *ext4: add fast commits feature.* https://lwn.net/Articles/826620/.

[43] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. *BabelFish: Fusing address translations for containers.* In *Proc. of 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[44] Yongseok Son, Sunggon Kim, Heon Y Yeom, and Hyuck Han. *High-performance transaction processing in journaling file systems.* In *Proc. of 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[45] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. *Scalability in the XFS File System.* In *Proc. of 1996 USENIX Annual Technical Conference (ATC)*, 1996.

[46] Andrew Tridgell and Ronnie Sahlberg. *DBENCH.* https://dbench.samba.org/.

[47] Stephen Tweedie. *Ext3, journaling filesystem.* In *Proc. of Ottawa Linux Symposium*, 2000.

[48] Stephen C. Tweedie et al. *Journaling the Linux ext2fs filesystem.* In *Proc. of 4th Annual Linux Expo*. Durham, North Carolina, 1998.

[49] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. *Barrier-Enabled IO Stack for Flash Storage.* In *Proc. of 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[50] Haitao Wu, Guohan Lu, Dan Li, Chuanxiong Guo, and Yongguang Zhang. *MDCube: a high performance network structure for modular data center interconnection.* In *Proc. of 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009.

[51] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, mar 2017.

[52] Jeseong Yeon, Minseong Jeong, Sungjin Lee, and Eunji Lee. *RFLUSH: Rethink the Flush.* In *Proc. of 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

Unsafe at Any Copy: Name Collisions from Mixing Case Sensitivities

Aditya Basu[*] aditya.basu@psu.edu
John Sampson[*] jms1257@psu.edu
Zhiyun Qian[†] zhiyunq@cs.ucr.edu
Trent Jaeger[*] trj1@psu.edu

[*]The Pennsylvania State University [†]University of California, Riverside

Abstract

File name confusion attacks, such as malicious symlinks and file squatting, have long been studied as sources of security vulnerabilities. However, a recently emerged type, i.e., ***case-sensitivity-induced name collisions***, has not been scrutinized. These collisions are introduced by differences in name resolution under case-sensitive and case-insensitive file systems or directories. A prominent example is the recent Git vulnerability (CVE-2021-21300) which can lead to code execution on a victim client when it clones a maliciously crafted repository onto a case-insensitive file system. With trends including `ext4` adding support for per-directory case-insensitivity and the broad deployment of the Windows Subsystem for Linux, the prerequisites for such vulnerabilities are increasingly likely to exist even in a single system.

In this paper, we make a first effort to investigate how and where the lack of any uniform approach to handling name collisions leads to a diffusion of responsibility and resultant vulnerabilities. Interestingly, we demonstrate the existence of a range of novel security challenges arising from name collisions and their inconsistent handling by low-level utilities and applications. Specifically, our experiments show that utilities handle many name collision scenarios unsafely, leaving the responsibility to applications whose developers are unfortunately not yet aware of the threats. We examine three case studies as a first step towards systematically understanding the emerging type of name collision vulnerability.

1 Introduction

A fundamental file system design choice is whether it will allow file names to be case sensitive or not, and modern file systems are diverse in their selection. A *case-sensitive file system* is one that allows the definition of multiple files whose names differ only in their case, such as `Foo.c` and `foo.c`. In a *case-insensitive file system*, only one file can be defined whose names differ only in their case. Historically, UNIX file systems are case sensitive, whereas Windows file systems are case insensitive. Further, case-insensitive file systems may be either case preserving (e.g. Apple File System (APFS), NTFS, etc.) or not (FAT), where a *case-preserving file system* preserves the case chosen (i.e., either `Foo.c` or `foo.c`), rather than converting all names to one case choice (e.g., all lowercase). Importantly, while choices in case sensitivity for a single file system may appear to be arbitrary or aesthetically driven, the precise semantics of interactions between two file systems with different case sensitivities can range from subtle to ill-defined, with associated consequences.

Practitioners have long had concerns about the implications of leaving case sensitivity as an open design choice [31] Historically, these concerns were not considered as pressing when file systems were associated with their respective operating systems and associated singular assumptions about case. However, *individual systems now frequently support a mixture of case-sensitive and case-insensitive file systems*, creating opportunities for files to be moved between file systems with different cases and file identifier encodings. More troublingly, *several file systems now support allowing the choice of case for individual directories* [12], complicating file operations by having multiple case and encoding semantics within the same file system.

Security risks related to this design choice therefore appear to be increasing. First, the Windows Subsystem for Linux [58] (WSL) integrates Linux and Windows platforms leading to expectations that files may be routinely copied from Linux (i.e., case-sensitive) to Windows (i.e., case-insensitive) file systems. Second, Linux `ext4` now supports case-sensitive and case-insensitive naming in the same partition, configurable per directory [12, 34]. Linus Torvalds expressed concerns about adding such support to `ext4` [31], stating that such features often cause “actual and very subtle security issues”.

Indeed, security issues caused by moving files from case-sensitive to case-insensitive file systems are starting to appear. For example, the `git` distributed version control system has suffered from multiple vulnerabilities (e.g., CVE-2014-9390, CVE-2021-21300), caused by how `git` clones repositories from case-sensitive file systems to case-insensitive file systems.

To exploit this, an adversary creates a repository in a case-sensitive file system with a directory whose name will *collide* (i.e., only differs in case) with a symbolic link (to another directory) added by `git` when the repository is cloned to a case-insensitive file system. The *name collision* between the directory and the symbolic link enables adversaries to overwrite the scripts that `git` executes. Such attacks can alter both the target resource’s content and/or its metadata, including its permission assignments.

Researchers have long been aware of hazards that may occur during file system name resolution [3, 4], particularly that programmers must validate safe use of symbolic links and check for "squatted" files when creating a new files. Many defenses have been proposed [7–9, 30, 40–42, 50–52, 55]. However, to the best of our knowledge, ours is the first work studying how case interplays cause name collisions that lead to incorrect, and in some cases, vulnerable behaviors. We show that utilities and applications currently do not recognize unsafe use of case-insensitive file systems, leading to these problems. This paper demonstrates the potential implications of the name collision problem, focusing on Linux and its supported file systems, thereby motivating both more and broader (e.g., other OS-FS combination) investigations. We identify potential gaps in the existing contract between the applications and the underlying file system that results in unsafe behaviors (see §8). We make the following contributions:

- We examine the security and correctness implications of *name collisions*, when two distinct file system resources with two distinct names map to to a single name, due to file system case sensitivity and/or encoding mismatches.
- We show that improper handling of case-[in]sensitivity and encoding can result in silent data loss and corruption, symbolic link traversal, unexpected hardlink creation, insecure merging of directory contents, and data disclosure due to incorrect overwriting of file system resources.
- We developed an automated method to test common Linux utilities for unsafe reactions to name collisions, finding a wide variety of responses, many of which are unsafe and possibly exploitable.
- We demonstrate novel exploits on three programs `dpkg`, `rsync`, and Apache `httpd`, showing how they operate incorrectly in the face of name collisions and how they would be exploited when deployed on case-insensitive directories.

2 Background: From Cases to Collisions

Beyond traditional, i.e. operating-system-entailed, decisions made with respect to case sensitivity, even Linux files systems now represent a surprising diversity of case sensitivity decisions. In particular, the desire to support some non-native applications, such as WINE and Samba from Windows systems, has motivated Linux file systems to support the case-insensitive file naming used in these non-native file systems.

The ability to create case-insensitive file systems has long been possible in some Linux file systems, such as ZFS, JFS, and ciopfs. However, these options are applied to the entire filesystem, rather than just the relevant directories for individual applications. In 2019, Linux kernel version 5.2 added support for per-directory case-insensitivity to ext4 [12, 34]. Later in 2019, similar support was added to the Flash-Friendly File System (F2FS) in Linux kernel version 5.4 [13, 14]. For case-insensitive directories, these file systems are case-preserving in nature.

2.1 Motivations for Increasing Case Diversity

Samba Samba [45] implements the Common Internet File System (CIFS) protocol which allows for sharing file systems over a network. Its primary use is sharing files with Windows clients that expect a case-insensitive file system. Hence, Samba implements user-space case-insensitive lookups even if the underlying file system is case-sensitive. Furthermore, it allows turning on/off case-sensitivity and case-preservation on a per-mount basis [46]. Note that this feature only works for non-Windows clients, which means that the actual file system can contain files differing only in case. This can lead to unexpected behaviors where Samba will choose to show only a subset of files. Deleting files which have collisions will now show the alternate versions, thereby giving rise to inconsistent behavior from the end user's perspective.

Samba's requirement of case-insensitive matching, which is done in user-space, incurs a huge performance overhead [37] thereby motivating the support for case-insensitivity in the ext4 file system [34–36]. Other programs/systems such as Wine [57], Network File System (NFS), SteamOS [48, 49] and Android [32, 59] would also benefit from in-kernel case-insensitivity support.

ext4 For ext4, the idea is that the filesystem at large can be configured to be "casefolding," which permits the mixing of case-sensitive and case-insensitive directories in the same file system. When creating an ext4 file system, the *casefold* option is applied, e.g., `mkfs -t ext4 -O casefold /dev/sda`. Setting the `+F` inode attribute on an empty directory makes it case-insensitive, e.g., `mkdir foo; chattr +F foo`. Note that case-insensitive directories can contain case-sensitive directories. This means that for a given path, `/foo/bar/bin/baz`, any of `foo`, `bar` and `bin` can either be case-sensitive or case-insensitive.

tmpfs `tmpfs` recently added case-insensitivity support [33]. The use cases are similar to that of ext4 with the addition of supporting sandboxing and container tools such as Flatpak.

2.2 Name Collisions

A *name collision* occurs when a file system maps two distinct names of two distinct resources to the same name. Name collisions can cause problems to occur if the names of distinct resources *collide* when those resources are replicated to a target directory that does not provide a 1:1 mapping for all replicated objects. Suppose one directory has two files with distinct names in that file system. Should those files be copied to a second directory in which the two file names collide (i.e., are mapped to the same name), then only one file will be created, which may be either of the original files or an unpredictable combination of the two files' content and metadata. Variation in case sensitivity between two file systems is a common origin of collisions, but diversity in other encoding properties, such as character choice (e.g., FAT does not support “, :, ∗, etc. [1]) and canonicalization processes, can lead to the same effect. For example, NTFS uses UTF-16 while APFS (macOS) and ext4 (Linux) use UTF-8 and older file systems can use other encoding schemes, such as iso8859-1.

Modern encoding schemes such as Unicode (e.g., UTF-8, UTF-16, etc.) have support for non-English characters that requires *case folding* [6] to perform case-insensitive matching. Unlike traditional techniques, case folding uses lookup tables to transform each character of the filename to a predetermined case. Furthermore, individual characters in Unicode can have multiple binary representations. Hence, a normalization scheme also needs to be applied to the case folded filename to ensure that the same characters are encoded using identical binary sequences. Consider the filenames `floß`, `FLOSS` and `floss`. All can coexist on a case-sensitive file system supporting reasonable character encodings, but since case-folding for both `floß` and `FLOSS` is `floss`, attempting to move these files to a case-insensitive system may only preserve one of the original triple.

In addition, *case folding rules and normalization techniques can differ across file systems*. The *locale* (or language) also influences the case folding rules. Due to such differences, ‘temp_200K’ (where K = Kelvin Sign or Unicode code point U+212A) and ‘temp_200k’ are considered identical on NTFS and APFS, but on ZFS[2] these filenames are considered different when using case-insensitive lookups. As a result, when two files of these names are copied from a ZFS file system to an NTFS file system, they will collide and only one filename and only one file will be created. For clarity and conciseness, we will use examples of ASCII-based, case-insensitive matching throughout the rest of the paper.

We propose a taxonomy for *name confusions*, shown in Figure 1, that captures the types of incorrect program behaviors that may stem from the ambiguous uses of names for file system resources. *Name collisions* are a subset of this broader class. Name confusions may be caused by three reasons: (1) because multiple names may refer to the same resource (i.e., *aliasing*); (2) because an adversary may create a resource of that name before the victim (i.e., *squat*); and (3) because the multiple resources may be associated with the same name (i.e., *collisions*). Of these, however, name collisions are the least explored for their correctness and security implications. As Linux is adding more support for case-insensitivity, it is crucial to understand the pitfalls and problems such functionality may incur. This work aims to study these issues.

3 From Collisions to Calamities

Name collisions can impair system functionality by modifying the content and/or metadata of files and directories in unexpected ways. Some name collisions have already led to security vulnerabilities [24]. In this section, we define the conditions in which a name collision occurs, the conditions under which such a collision may be exploitable by an adversary, and describe a known vulnerability that is caused by a name collision.

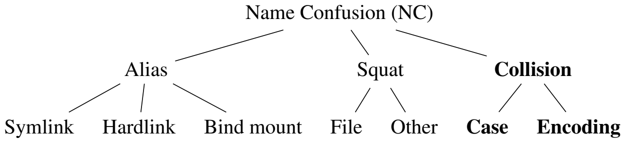


Figure 1: Taxonomy of *name confusion vulnerabilities* divided into *alias* (i.e., multiple names for a resource), *collision* (i.e., multiple resources for a name), and squat (temporal ambiguities in names vs. resources) classes

3.1 Causes of Name Collisions

A process may cause a name collision under the following conditions.

- There exists a *source resource* (e.g., file or directory) in a case-sensitive file system, whose name is *source name*.
- The process uses a relocation operation to place the source resource in a *target directory*, where the target directory is a case-insensitive or case-preserving directory. Examples of relocation operations include copy (e.g., `cp`, `rsync`, or an archive operation, such as `tar` or `unzip`) or move (e.g., `mv`).
- The relocation operation produces a *destination name* from the source name for the name of the source resource when placed in the target directory.
- There is a *target resource* with a *target name* whose name differs from the source name, but maps to the same name as the destination name does in the target directory (e.g., due to differences in case-folding rules between the source and target directory).
- If the process is authorized to modify the target resource, the process's relocation operation results in a name collision between the target and source resources.
- If the relocation operation proceeds despite the name collision, then the target resource's content and/or its metadata may be modified using the source resource content and/or metadata.

When these conditions are met, a name collision occurs such that the target resource in the target directory will be modified using the source resource. In most cases, modifying a target resource using a source resource of a different name is an unexpected result. We test how common Linux utilities react to name collisions and examine case studies where name collisions cause incorrect operation.

Given the above conditions, there are several clear scenarios where the movement of files involving the following types of file systems (following the categorization in §2.2) could result in name collisions:

[1] `http://elm-chan.org/fsw/ff/doc/filename.html`

[2] By default, the ZFS file system does not perform normalization. We use this default behavior for the given example.

- Case-sensitive and case-insensitive file systems.
- Two distinct case-insensitive file systems with different case folding rules, e.g. ZFS to NTFS, etc.
- Two file system whose locales are different but they still use the same file system format (such as ext4).
- A single file system that supports per-directory case-insensitivity, e.g. ext4.

Clearly, name collisions may impact system functionality by causing collateral damage to resources supposedly unrelated to the operation, even removing the target resource entirely. In addition, name collisions may be used to exploit the process performing the relocation operation in a version of a *confused deputy attack* [25]. An adversary only requires write access to the source directory to produce source names that may lead to name collisions to perform an attack. We note that adversaries require fewer permissions to perform attacks using name collisions than other name confusion classes, which require write access to a directory used in name resolution of the target resource [54]. Thus, remote attacks using file system archives, such as tarballs and zip files, as well as file repositories, such as GitHub, can be the sources of attacks.

In practice, to perform a successful attack using a name collision, the victim process has to help the adversary in two ways. First, the victim process has to use the source resource in a relocation operation planted by an adversary as described above. In addition to archives, other activities, such as backups, may provide opportunities for exploitation of name collisions. In addition, ad hoc user actions copying files, e.g., from Linux to Windows in the Windows Subsystem for Linux, may result in unexpected and exploitable collisions. Second, the target directory of the relocation operation has to be predictable by the adversary to enable them to produce a source name that leads to a colliding destination name. Archives make this task much easier because the archive itself may be crafted to provide the target resource that is exploited by creating a collision with another archive file. A recent vulnerability in the git distributed revision control system demonstrates exactly this, as described below.

3.2 An Example Collision Vulnerability

Security vulnerabilities due to filename collisions across different file systems have been demonstrated in the wild. Consider a recent vulnerability in the git distributed version control system (CVE-2021-21300). This vulnerability results in remote code execution after cloning a maliciously crafted repository created on a case-sensitive file system to a case-insensitive file system.

Figure 2 depicts the maliciously crafted repository structure. Note that this directory structure works correctly on a case-sensitive file system. However, on case-insensitive file systems, the presence of the 'a' (small) and 'A' (capital) directories creates a collision that exposes a vulnerability. This collision results in a vulnerability when using git's out-of-order checkout machinery. Git Large File Storage (LFS) uses out-of-order checkouts for downloading binaries in the background. Say the repository creator (adversary) marks 'A/post-checkout' for an out-of-order checkout. When a user clones this repository to a case-insensitive file system (e.g., NTFS), git performs a sequence of operations that: (1) replaces 'A' with the symbolic link 'a' and (2) writes the script file 'A/post-checkout' to '.git/hooks/post-checkout' due to the symbolic link 'a'. After the files are downloaded, git runs the script '.git/hooks/post-checkout' that the adversary provided, which is obviously undesirable.

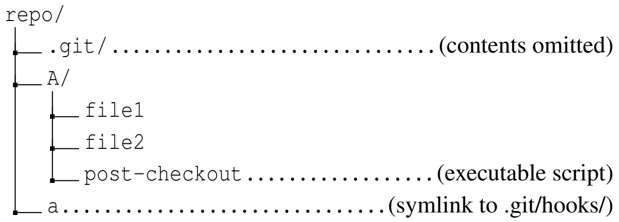


Figure 2: Example for Git CVE-2021-21300

In this case, a maliciously crafted git repository can be designed to provide a target resource of the symbolic link 'a', which when collided by 'A' in resolving the source resource 'A/post-checkout' redirects the operation to a directory chosen by the adversary using the symbolic link.

3.3 The State of Defenses for Name Confusions

Currently, operating systems provide no innate defenses to prevent name collisions, leaving the challenge to programmers. However, researchers have studied problems due to other types of name confusions extensively, proposing a variety of defenses [7–9, 30, 40–42, 44, 50–52]. However, researchers have shown that comprehensive program defenses are expensive [55] and that system-only defenses will always be prone to some false positives [5]. Leveraging limited program information [28, 53] still results in some false positives.

As a result, library commands for opening files have been extended in a variety of ways to prevent name confusions from occurring. The open command has been extended with flags to detect file squats (i.e., O_EXCL|O_CREAT to detect the presence of an existing file during file creation) and prevent unexpected use of aliases (i.e., O_NOFOLLOW to prevent following symbolic links). However, the use of squats and aliases is desirable in some applications, despite their risks. Further complicating the matter is that adversaries may exploit the gap between when a program validates a file system resource and opens that resource to create name confusions, known as time-of-check-to-time-of-use (TOCTTOU) attacks [4, 39]. The openat command has been added to enable programmers to avoid TOCTTOU attacks, by opening a file from a validated directory (i.e., file descriptor to the directory of the desired file). However, the successful use of openat requires the programmer to check for unwanted squats or aliases themselves. An alternative is proposed by the `openat2` command instead controls *how* files may be opened, such as requiring all file components accessed to be descendants of the directory from which the operation originates. However, `openat2` cannot prevent name confusions for some cases (e.g., using links across file systems). `openat` and `openat2` reduce the attack surface of squat and alias attacks, but do not eliminate them entirely, and depend on the programmer's additional actions to configure these commands and to check for TOCTTOU attacks.

At present, the above commands make no effort to help programmers address name collisions. As a result, utilities to perform copy and move operations and applications that may utilize file systems with multiple or mixed (e.g., ext4 and F2FS) case sensitivities or encodings may not detect and resolve name collisions correctly. We will examine the possible defenses for name collisions in §8.

4 Overview

In this paper, we aim to explore the impact that name collisions may have on file system security. To do this, we propose to examine three research questions.

RQ1: *How do applications invoke utilities that may allow unsafe name collisions?* In §6, we examine Linux packages to determine the most common options that applications employ for the utilities used to perform copies. We examine how frequently application packages use utilities in copy operations by scanning their scripts for such operations, as shown in Table 1.

RQ2: *When do the utilities for performing copy operations allow unsafe name collisions?* Recall that §3.1 defines the conditions under which an unsafe name collision may occur. This research question asks whether the utilities that applications may use to perform copy operations (e.g., `cp` and `tar`) prevent unsafe effects when a name collisions occur. For these utilities and the common options found in RQ1, we examine a variety of name collision scenarios to determine whether the utilities allow name collisions and their unsafe effects to occur as shown in Table 2a.

RQ3: *What correctness and security problems are caused by name collisions?* In §7, we examine three case studies where we show how name collisions cause programs to behave incorrectly. In particular, we show concretely how applications can be vulnerable to name collisions when target resources are deployed on case-insensitive or case-preserving file systems.

Impacts: A preview of our result is that: (1) many applications rely on these utilities to copy file system resources and repositories/archives; (2) the utilities used to copy file system resources and repositories/archives often allow unsafe name collisions, although the specific responses vary in ad hoc ways; and (3) applications currently lack defenses against name collisions, which can lead to incorrect operation and exploitable vulnerabilities.

5 Testing for Name Collisions

This section details an automated tool for testing the responses of common Linux utilities used for relocation operations to name collisions. As described in §3.1, a name collision is caused by creating a source name that will be converted to a destination name by the relocation operation that is equal to a target name in the target directory of the operation. Thus, our aim is develop a method to automate the generation of source resources with names that will lead to name collisions when relocated to case-insensitive targets and identify when operations allow the name collision to occur, detecting the effects of those operations.

5.1 Test Case Generation

The individual test cases are generated to test file system resources of various types, including regular files, directories, symbolic links (to files and directories), hard links, pipes, and devices. In addition, we have found that creating collisions in non-trivial directory structures may also lead to incorrect behaviors. Figure 3 shows an example test case where the directories as well as their contents result in a collision when transferred to a case-insensitive file system. As a result, we aim to generate test cases that result in name collisions at different depths of the directory being copied, as evidenced by the collision between directory names at depth 2 (i.e., "dir" and "DIR") and the impact on colliding resources of different types (i.e., a regular file "foo" and a pipe "foo").

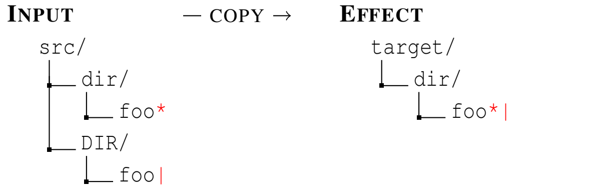


Figure 3: An example of squashing case-sensitive directory names and file names of two different types. Here, '*' means a regular file and '|' means file type is a named pipe.

Since we are testing the behavior of various utilities that perform relocation operations, we can control the source and target names in creating test cases. As a result, the choice of names is trivial. We create source directories that contain both the target resource (i.e., a resource copied first from the source to the target) and the source resource (i.e., a resource copied later by the utility to collide with the target resource (i.e., now in the target directory). This is similar to the way name collisions would occur when copying an archive or repository that causes a collision, as the `git` vulnerability. Since different utilities may process resources in different orders, we generate test cases with both orderings of resources that may cause collisions.

The only decisions then are what are the resource types of the source and target resources and where to place them in the source directory hierarchy to cause the desired collision to be created. Symbolic links, pipes, and devices only create interesting behaviors when used as target resources. For symbolic links, the unsafe effect is to follow the link to another file system resources, which only happens with the symbolic link as the target resource. For pipes and devices, the unsafe effect is to send the source resource's content to the pipe or device, which also is only possible if these are target resources.

As a result, the automated test generation produces test cases consisting of source and target resources of all combinations of potentially unsafe resource types and places these test cases at depth one and/or two of the file system hierarchy. For `rsync`, we specifically found an issue caused by a collision at depth two, but not at depth one (see §7.2).

5.2 Detecting Collision Effects

The key idea is to record the file system operations sufficiently to detect that an unsafe name collision has occurred. Since we design the test cases to create a name collision on a relocation operation, we want to detect when such an operation is a successful collision. Then, we need to determine the impact of the operation to classify the effect according to one of the ten effect options defined in §6.1.

We monitor file system operations using `auditd` to detect successful collisions. An example of a log indicating a collision is shown in Figure 4. In this example, a *create* operation creates a target resource named "root" using `openat` command, but a later *use* operation to the same resource (i.e., same device-inode pair, see below) is associated with a name "ROOT", which differs from the name used when the resource was created. Note that although the target resource was created on a case-insensitive file system, multiple names may be used that are resolved to the same name.

We say that a collision is successful when we detect a *use* of a target resource with a different name than that used to create the target resource. To detect such collisions, we first identify the file system operations that *create* a target resource, recording its combination of device[3] and inode identifiers, which form a unique resource identifier and its pathname. In Figure 4, the name component "root" will be important to detecting the collision. We then capture all the file system operations that *use* the target resource. In Figure 4, the pathname of the *use* operation differs between "root" and "ROOT", indicating a name collision.

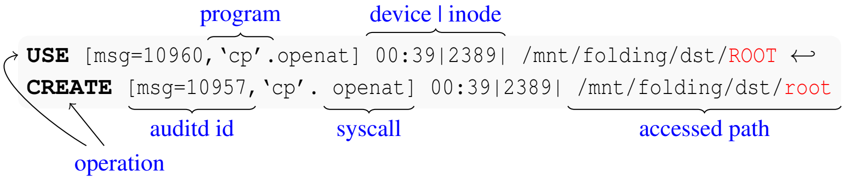
We also record a positive when a *use* operation deletes and replaces a resource from a prior *create* operation, as some collisions may cause the target resource to be deleted and the source resource to replace it. We validate that there is a *create* operation for the colliding destination name to verify the cause of the deletion is a collision.

To detect the effect of a name collision, we examine the resulting resource that now maps to the target name. We compare the source resource and target resource content and metadata to the resultant resource to determine whose content and/or metadata (i.e., source, target, or neither) the resource has. For tests on directories and hardlinks, we examine the directories and the resultant directory entries.

6 Name Collisions on Linux Copy Utilities

In this section, we examine how common Linux utilities that applications use to copy files from one part of the file system to another react when the copy operation causes a name collision in a case-insensitive directory. We note that the impact on move operations is similar because in most cases it simply performs a copy first and then deletes the source. However, when both the source and target are on the same file system, the underlying file system may directly relocate the contents of the source. This can result in unusual consequences on file systems that support per-directory case-[in]sensitivity. E.g., on ext4, moving a case-sensitive directory into a case-insensitive directory will preserve case-sensitive characteristics of the moved (or source) directory. However, when copying, the directories are newly created and these directories inherit the case-[in]sensitive characteristic from the parent directory. If the copy does not preserve attributes on directories, then all new directories will be case-insensitive under this scenario. Even though move works differently in certain cases, the collisions that may result from move have the same effect as that of copy. Hence, we only assess Linux utilities that perform copy operations below.

To quantify the ubiquity of these utilities, we survey their use by packages on Debian 11.2.0. We retrieve all packages from the Debian installation DVD and count the number of times the copy utilities are used inside the packages' scripts. The results are summarized in Table 1. Note that the listed uses of these utilities are lower bounds because we do not parse executable binaries. Hence, we miss uses where the utilities are invoked via system calls such as `system(...)`, `execve(...)`, etc.



```
USE    [msg=10960,'cp'.openat] 00:39|2389| /mnt/folding/dst/ROOT
CREATE [msg=10957,'cp'. openat] 00:39|2389| /mnt/folding/dst/root
```

Figure 4: Example violation reported by name collision testing.

[3] On Unix-like systems, each device is assigned a major and minor number. `auditd` reports these numbers (in hexadecimal) as XX:YY, where XX is the minor number and YY is the major number. Each file system mount point can be uniquely identified using these numbers.

Table 1: Prevalence of copy utilities

`tar`		`zip`		`cp`		`cp*`		`rsync`	
10	mc	21	texlive-plain-generic	78	hplip-data	12	dkms	28	mariadb-server
8	perl-modules	15	aspell	32	dkms	2	udev	5	duplicity
7	libkf5libkleo-data	11	libarchive-zip-perl	22	libltdl-dev	2	debian-reference-it	4	texlive-pictures
6	pluma	7	texlive-latex-recommended	20	autoconf	2	debian-reference-es	2	vim-runtime
6	mc-data	5	texlive-pictures	18	ucf	1	zsh-common	1	rsync
	...		...		...		...		...
107	TOTAL	69	TOTAL	538	TOTAL	25	TOTAL	42	TOTAL

We calculate the number of times that each command (tar, zip, etc.) is used inside scripts from various packages. We investigate 4752 .deb packages from the installation disk (DVD #1) of *Debian 11.2.0*. Only the top-five packages are shown (entries are sorted in descending order for each command).

6.1 Collecting Responses to Name Collisions

The name collision test cases and the responses of copy utilities are shown in Table 2a. The 'Target Type' column represents the resource type of the target resource that may be overwritten. The 'Source Type' represents the resource type of the source that collides with the target. The rest of the columns represent individual utilities and their responses to name collisions between a source resource of the source type and a target resource of the target type.

Below is a comprehensive list of the types of responses observed. Only "Deny" and "Rename" prevent name collisions from causing unsafe and possibly exploitable behaviors, although both may block legitimate functionality in some cases. "Ask the User" may result in an unsafe response if the user allows the target resource to be overwritten. Note that more than one response is possible for each test case.

Delete & Recreate (×) *Delete* the target resource and *create* a new resource based on the source resource. The new resource's type, as well as its data and metadata, is determined by the source resource. The target resource is lost without any notification.

Overwrite (+) *Overwrite* the data and metadata of the target resource using the source resource. Unlike *Delete & Recreate*, the name of the target resource is preserved. If file `foo` is being overwritten with file `FOO`, then the final file will be named `foo` but will have the contents and metadata of file `FOO`.

Corrupt (*C*) Contents of a resource that is not involved in name collision (i.e., not the target resource) is modified. For a more in-depth discussion, refer to §6.2.5.

Metadata Mismatch (≠) After a successful copy of a given source resource, some metadata, such as its name, UNIX permissions, user or group ID, extended attributes, or timestamp, remain from the target resource, creating a resource with a *mismatch* between the data (from the source) and the metadata (from the target).

Follow Symlink (*T*) Follow (or traverse) *symbolic links*, *even when explicitly directed not to do so*.

Rename (*R*) The source name is *renamed* automatically to avoid creating a name collision, such as by appending a counter, resulting in a copy of the source resource in the target resource's directory with a non-colliding name.

Ask the User (*A*) To resolve a collision, *ask the user* to choose from a list of actions, such as to overwrite the target resource, skip copying the source resource, rename the target resource, abort, etc.
Note that the user can still choose a response that results in adverse consequences. For instance, if the user chooses to overwrite the target, the target's data and metadata are modified using the source.

Deny (*E*) *Deny* the copy associated with a collision and report an error.

Crashes (∞) Collisions can result in the program hanging (e.g., going into an infinite loop) or *crashing*.

Unsupported file type (−) Does not support copying a resource if the source resource is of this file type. Note that if hardlinks are not recognized by a utility, then it simply creates a fresh copy of the underlying file.

The exact command-line flags used used to generate Table 2a are listed in Table 2b. To identify these flags, we analyzed 4,752 `.deb` packages on Debian 11.2.0's installation DVD. We found that the most commonly used flags enabled the following functionality.

- Support recursively copying all directories.
- Support copying symbolic-links and hard-links as-is but *do not follow* them.
- Preserve metadata such as UNIX permissions, extended attributes (xattr), timestamps, and owner/group IDs (uid/gid).

Before examining the responses in Table 2a, we briefly note some additional context for two of the columns.

cp vs. cp* Both of these represent the same executable binary. The difference is in the way the command-line arguments are passed to the binary. Specifically, the format of specifying the source directories is different.

Consider that the source directory (to be copied) is `foo`. For `cp`, we will pass it as `foo/` while for `cp*` we will use `foo`. Note the trailing / is missing in the latter case. Just this difference significantly changes the behavior of `cp` as noted in Table 2a.

We use the `cp*` method of invocation coupled with shell completion, e.g., '`cp src/* /target`' where the shell re-

Table 2: Name Collision Responses for Popular Linux Utilities

Name Collision between							
Target Type	Source Type	`tar`	`zip`	`cp`	`cp*`	`rsync`	Dropbox
file	file	×	*A*	*E*	+≠	+≠	*R*
symlink (to file)	file	×	*A*	*E*	+*T*	+≠	*R*
pipe/device	file	×	–	*E*	+	+	–
hardlink	file	×	–	*E*	+≠	+≠	–
hardlink	hardlink	*C*×	–	*E*	*C*×	*C*+≠	–
directory	directory	+≠	+≠	*E*	+≠	+≠	*R*
symlink (to directory)	directory	+	∞	*E*	*E*	+*T*	*R*

(a) This table shows results of copying files/directories from a case-sensitive to a case-insensitive file system. `cp*` refers to `cp` being used with shell completion. For e.g., '`cp * /target`' which copies all items from the current directory to `/target` directory.

Utility	Version	Flags
`tar`	1.30	`-cf/-x`
`zip`	3.0	`-r -symlinks`
`cp`	8.30	`-a`
`rsync`	3.1.3	`-aH`

(b) This table lists the version of utilities and command-line flags used for the experiments. For `tar`, `-cf` was used to create the archive and `-x` to expand the archive.

×	Delete existing file and create new file
+	Overwrite existing file. For directories, merge their contents.
≠	Mismatch between content and metadata
A	Ask user to resolve the collision
T	Follow (or traverse) symlink
C	Corrupts non-colliding files
E	Deny operation and report error
∞	Program crashes, or hangs
–	Ignore unsupported file type (for hardlinks create regular file instead)
R	Rename colliding file/directory

places `src/*` with each individual entry present inside `src` sans the trailing `/`. When testing the `cp` method, we change the command to '`cp src/ /target`'.

Dropbox Strictly speaking, *Dropbox* [11] is not a copy utility but a popular file synchronization utility. It is intended to replicate entire directories across multiple machines and file systems.

We mention Dropbox to highlight its distinct response to handling *potential* name collisions. Even when the underlying file system is case-sensitive, Dropbox treats it as *case-insensitive*. It proactively renames the files and directories to avoid name collisions that could occur if they were transferred to a case-insensitive file system. Note, however, that its renaming strategy is not even uniform across platforms: For example, the Dropbox application appends "(Case Conflicts)", "(Case Conflicts 1)", etc. to the file/directory names in case of a potential collision, whereas, when using their web-based interface, they append "(1)", "(2)", etc. instead.

6.2 Unsafe Responses to Name Collisions

Several responses shown in Table 2a demonstrate that utilities often allow unsafe responses to name collisions. In this section, we examine some of the more concerning responses to show how utilities delegate responsibility for security against name collisions to the applications that invoke them. For the examples in upcoming sections, `src/` and `target/` are on case-sensitive and case-insensitive file systems respectively.

6.2.1 Silent data loss with *tar, cp* & rsync*

Name collisions involving files generally result in silent data loss. From Table 2a, we can see that `tar` deletes and recreates (×) files when collisions occur. Hence, when there is a name collision between `foo` and `FOO`, only one of these files will remain in the target directory. The other file is permanently lost without any notification.

Similar to `tar`, `cp*` and `rsync` also lose files silently. However, their behavior of overwriting (+) files results in other problems that are discussed later in this section.

Unlike `tar`, `zip` and `cp` will ask a user for next steps (*A*) or report an error (*E*) respectively. Hence, they are not prone to silently losing files.

6.2.2 Merge directories with *tar, zip, rsync & cp**

Name collisions involving two directories results in their contents (files, directories, etc. inside the directory) being merged. All of `tar`, `zip`, `rsync`, and `cp*` will silently merge directory contents without notifying the user. Figure 5 highlights this issue using a directory listing.

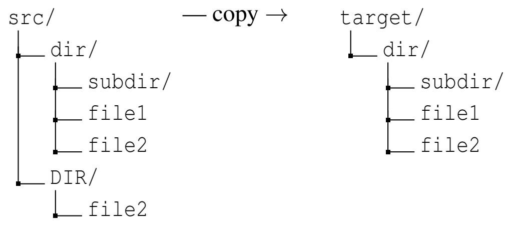


Figure 5: Impact of merging directories

In this example in Figure 5, the data of file `file2` is overwritten by the content written last in the copy operation. For example, if `src/DIR`'s contents are written last, then its content for `file2` is preserved and `src/dir`'s is lost.

Furthermore, when the colliding directories have different UNIX permissions, a collision results in metadata mismatch (≠). With respect to Table 2a, the UNIX permissions of the target resource are overwritten with permissions of the source resource.

In Figure 5, consider `src/dir/` with perms=700 and an adversary who creates `src/DIR/` with perms=777. After a copy (using any of the above utilities), `target/dir/` will have perms=777 effectively giving the adversary permission to the contents of the original `src/dir/`.

6.2.3 Stale names

Whenever utilities resort to overwriting (+), we end up with stale file/directory names. For example, consider a name collision between a target resource `foo` (file content: 'bar') and a source resource `FOO` (file content: 'BAR'). After copying with `rsync` or `cp*`, we will end up with file `foo` whose contents are 'BAR'.

The problem with such name collisions is that to the end user (or other programs), it will appear that `foo` was successfully copied while in reality `FOO` was copied. Just using the filename is not enough to discern which files were successfully copied. This is especially true for case-preserving file systems where the user has the expectation of the filenames being preserved. Hence, it is not unreasonable for the user to expect `foo` should contain `bar`.

6.2.4 Symbolic link traversal at target

Name collisions between *symlink (to file) and a regular file* results in `cp*` following the symlink (T) and overwriting (+) its target's contents with that of the regular source file. With regards to Table 2a, if the target resource is a symbolic link and the source resource is a file, then `cp*` ends up following the symlink and writing data to the resource referenced by the symlink.

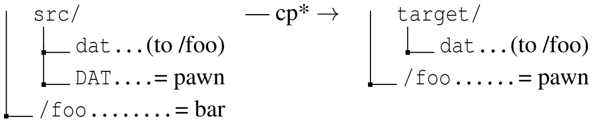


Figure 6: Following symlink

Figure 6 illustrates this case with an example. `src/dat` is a symbolic link to `/foo` and `/foo` contains 'bar'. Mallory (our adversary) does not have write access to `/foo` but does have access to `src/`. She creates `src/DAT` which contains 'pawn'.

Then the administrator starts the copy using: `cp -a src/* target/`. At this point, `cp` first creates the symlink `target/dat`. Then it overwrites (+) this symlink with the contents of `src/DAT`, effectively updating the file `/foo`. After the copy has completed, `/foo` contains 'pawn'.

*cp** has no command-line options to prevent traversal of symbolic links at the target. Only link traversal at the source can be turned off via command-line flags.

6.2.5 The case of *hardlink* – *hardlink* name collisions

During a copy when hardlinks (whose targets are different) collide, it can corrupt (C) other non-colliding files and create spurious hardlinks. Table 2a shows that this behavior is exhibited by `tar, cp*`, and `rsync`. An interesting observation is that, regardless of whether the utility's behavior is *Delete & Recreate* (×) or *Overwrite* (+), this problem affects both.

To understand this scenario, consider Figure 7 that uses `rsync` to perform the copy. The same color coding represents files that are hard-linked to each other. So `src/hfoo` and `src/zzz` are hard-linked, representing the same file. These files contain 'foo'. Similarly, `src/hbar` and `src/ZZZ` are hard-linked and they contain 'bar'.

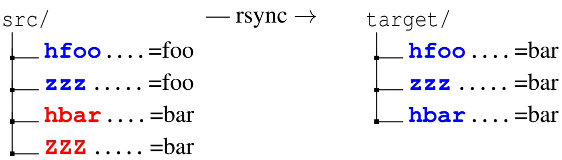


Figure 7: *hardlink* – *hardlink* name collision

After copying using `rsync`, `target/` contains three files that are all hard-linked to each other. Unlike the `src/` directory, `target/hfoo`, `target/hbar`, `target/zzz` are all hardlinks of each other and they contain 'bar'.

Additionally, note that although the name collision happened between `zzz` and `ZZZ`, the contents of `hfoo` were replaced. Even `tar`, which deletes the old file and recreates it, exhibits this behavior.

The following order of operations undertaken by `rsync` result in this behavior.

1. Copy `src/hbar` to `target/hbar`. Now `target/hbar` contains 'bar'.
2. Copy `src/zzz` to `target/zzz`. Now `target/zzz` contains 'foo'.
3. In `target/`, hardlink `ZZZ` to `hbar`. Due to name collision, this effectively changes `zzz` to be hard-linked to `hbar`. Now `target/zzz` contains 'bar'.
4. In `target/`, hardlink `hfoo` to `zzz`. Now `target/foo` contains 'bar'. Additionally, all three files inside `target/` are hard-linked to each other.

The above copy is semantically different from the `src/`. Specifically, name collision results in *distinct* sets of files getting hard-linked with each other at the `target/`.

7 Case Studies

In this section, we examine case studies where name collisions cause unsafe behaviors, some of which are exploitable.

7.1 `dpkg` Package Manager

`dpkg` is the package manager on Debian OS and its derivatives such as Ubuntu. `dpkg` packages are compressed tarballs with extension `.deb`. When `dpkg` processes a package, it tracks all files it creates during package installations in a database. Before installing a new package, `dpkg` leverages this database to ensure that any files of previously installed packages will not by overwritten by this new package thereby preventing potentially malformed packages from corrupting the system.

On the other hand, we have observed that `dpkg` will allow a package installation to replace any file whose name is not in its database, even privileged user files. Thus, as long as a file in a package has a filename that does not match the filename of another package's file, `dpkg` will install the file, silently replacing any existing file.

However, regardless of the underlying file system, the above database is matched in a *case-sensitive* manner. This allows new packages to *replace files* of previously installed packages via name collisions effectively circumventing the safeguards in dpkg.

In addition, and perhaps even more seriously, dpkg may allow an adversary to replace a package's customized config file with the default, reverting important changes. deb packages can mark certain files as configuration (or config) files. During package upgrades, if dpkg spots modifications to these config files then it prompts the user to review the changes.

However, the config files are also matched in a case-sensitive manner. Under name collisions, dpkg will just replace the original package's config file with the config file of the new package. For services, such as sshd, httpd, etc., config files are critical to their security, so such overwrites can potentially make the system vulnerable..

Reporting We have reported these issues to the maintainers of dpkg. The maintainers of dpkg have since updated their package documentation [10] to warn end user communities not to use dpkg where targets may be case-insensitive (i.e. specific directories, or entire file systems).

During our discussions, we analyzed 74,688 packages and found 12,237 filenames from those packages would collide if a case-insensitive file system were used, breaking multiple packages that contain these files. The name collision problem is fundamentally entrenched into the way dpkg is implemented because it reasons about *names* without involving the underlying file system(s).

7.2 Rsync

rsync demonstrates vulnerable behavior when processing name collisions involving *directories*. During copy, the default behavior of rsync is to simply recreate the symbolic links present at source. However, when colliding directories contain sub-directories and symbolic links with the same name, the collision causes rsync to suffer from link traversal[4].

Consider the source directory listed in Figure 8. Here, the directories topdir/ and TOPDIR/ only differ in case. So when copying to a case-insensitive file system, rsync will encounter a name collision.

```
src/
├── topdir/
│   └── secret/ .......................... symlink to /tmp
└── TOPDIR/
    └── secret/
        └── confidential ..................... regular file
```

Figure 8: Case-sensitive source that rsync is copying

We use the following command to perform the copy:

```
rsync -a src/ dst/
```

where,

-a recursively copy directories, preserve symlinks, timestamps, and discretionary access control permissions

src/ is case-sensitive

dst/ is case-insensitive

After the copy is completed, the newly created files are shown in Figure 9. Note that the file named confidential ends up in /tmp.

```
dst/
├── TOPDIR/
│   └── secret/ .......................... symlink to /tmp
└── /tmp/confidential ......................... link traversal
```

Figure 9: After copying to case-insensitive destination

rsync has created the /tmp/confidential file by following the symbolic link dst/TOPDIR/secret.

Below, we describe how this situation can be exploited. Consider an adversary who wants to access a confidential file in TOPDIR/ to which she lacks any access. However, she knows that TOPDIR/ is processed by a backup operation using rsync. If she can create a sibling directory topdir/, to which she will have read-write access, she can direct rsync to write the confidential file (inside TOPDIR/) to any directory of her choosing by creating a symbolic link inside topdir/ to that directory.

Reporting We reported this issue to the rsync maintainers, and they told us that user's should not use rsync with non-case honoring file systems. However, we have concerns about the user community following such a recommendation in this case, since rsync is often used by individuals.

In the course of these discussions, we learned the cause of the incorrect behavior. rsync assumes a one-to-one mapping of directories between source and target file systems. When a name collision results in two source directories being mapped to a single directory in the target, rsync can be tricked into incorrectly predicting the target file type. In the presented scenario, a symbolic link src/topdir/secret (to a directory) is incorrectly inferred to be a regular directory src/TOPDIR/secret.

rsync uses the O_NOFOLLOW flag with open() to prevent link traversal and uses openat()/openat2() to contain link traversals within a directory hierarchy, but this strategy fails when the symbolic link is treated as a directory.

7.3 Apache httpd

Security of certain applications relies on the security parameters of the underlying file system. One such application is Apache's httpd. It allows access to the underlying file system via the HTTP protocol, relying on the UNIX Discretionary

[4] In this case, the name collision makes the alias exploitable, again combining name confusions.

Access Control (DAC) permissions[5] to mediate the access. For example, files are accessible over HTTP only if: (i) its UNIX group is www-data and has read permission for the group, or (ii) has world-readable UNIX permissions.

Using utilities for copying directories between systems can silently alter these DAC permissions in unintended ways, leading to serious security lapses. We illustrate this scenario using Apache httpd and migration of its data using tar. To study the impact of name collisions on the security parameters, we assume that the migration happens from a case-sensitive to a case-insensitive file system. The behavior of tar discussed below draws from the discussion of Table 2a.

To protect sensitive directories, httpd can be configured to only allow authenticated users to access specific directories. A commonly used approach is to configure authentication via the .htaccess file [1] which lists the valid users/groups allowed to access a specific directory over HTTP. All subdirectories inside the sensitive directory are also protected. We show that the use of additional security-oriented files can be exploited under the presence of name collisions.

Scenario httpd serves the contents of www/ (of Figure 10) over HTTP. Initially, www/ is stored on a case-senstive file system. The directory hidden/ is inaccessible over HTTP since the *others* permissions are cleared. Next, protected/ is configured to be accessible only to specific users using the .htaccess file.

```
www/
├─hidden/ ............................. perm=700
│  └─secret.txt
├─protected/.........group=www-data, perm=750
│  ├─.htaccess............(only allow valid users)
│  └─user-file1.txt
└─index.html
```

Figure 10: www/ on case-sensitive file system

Adversary A UNIX user called Mallory has read-write access to www/ directory. However, DAC permissions prevent her from accessing hidden/ directory because its owner is another user. Additionally, protected is inaccessible since Mallory does not belong to the group www-data.

She modifies www/ as shown in Figure 11 and adds the HIDDEN/ and PROTECTED/ directories with the intent of gaining access to hidden/ and protected/ via a name collision.

Vulnerability tar is used to migrate the adversary-modified www/ directory to another system that uses a *case-insensitive* file system. Figure 12 shows the state of the file system once the tarball (archive format of tar) is extracted.

Now, the previously inaccessible hidden/ directory is now accessible over HTTP. Additionally, since the .htaccess file is cleared, unauthenticated users will be allowed to view protected/ over HTTP.

```
www/
├─hidden/..................................perm=700
│  └─secret.txt
├─HIDDEN/..................................perm=755
├─protected/ ..............group=www-data, perm=750
│  ├─.htaccess.................(only allow valid users)
│  └─user-file1.txt
├─PROTECTED/ ...............................perm=755
│  └─.htaccess............................(empty file)
└─index.html
```

Figure 11: Adversary modified www/ on the case-sensitive file system

```
www/
├─hidden/..................................perm=755
│  └─secret.txt
├─protected/ ...............................perm=755
│  ├─.htaccess............................(empty file)
│  └─user-file1.txt
└─index.html
```

Figure 12: www/ after migrating to case-insensitive file system

Reporting We have reported this scenario to the Apache maintainers, but have not yet reached a resolution. Using Table 2a, we can reason about the above problems. Under a *directory* – *directory* collision, tar incorrectly modifies metadata. This happens for the hidden/ – HIDDEN/ collision. Here, DAC permissions of the latter are applied to the former resulting in the leakage of secret files.

For *directory* – *directory* collisions, tar will also merge contents of both directories. For protected/ – PROTECTED/ collision, this merger results in the empty .htaccess file overwriting the original one that restricts access to authorized users. The end result is that all users are now allowed access to the new protected/ directory.

8 Potential Defenses

As discussed in the context of name confusion attacks in general in §3.3, it can be difficult to produce defenses to prevent name collisions as well. In this section, we discuss some options and their limitations.

Name collisions are due to differences in case folding rules among file systems, e.g., case sensitivity and encodings, so it is difficult to ensure that name collisions cannot happen. Suppose a system has only one file system. Even then, an archive constructed on another file system using conflicting case folding rules may cause name collisions to occur when expanding the archive. Since user-space programs cannot determine the case-folding rules that may be applied to a file, user-space solutions alone will be unreliable. In addition, they may be prone to TOCTTOU attacks [3, 4]. Thus, extending library calls like realpath to detect name collisions will not sufficiently solve the problem. In addition, system solutions

[5] If the system supports Mandatory Access Control (MAC), then DAC is used in conjunction with MAC.

lack knowledge of the programmer intent that caused the collision and hence, a systems-only defense for name confusion will suffer from false positives [5].

For example, one idea may be to write a wrapper to vet archives prior to expansion operations (e.g., `tar` and `zip`) to validate that each file in the archive will result in a distinct file after expansion. One way to do this is to check for name collisions among all the files in the archive. Although the notion that no two files in an archive should collide seems intuitively reasonable, there are critical drawbacks to this defense. First, the target directory may already have files that may result in collisions, limiting its utility. Second, targets that support per-directory case-sensitivity can switch between case-sensitive and case-insensitive lookups when resolving a filepath, leading to incorrect assumptions about case-sensitivity and being prone to race conditions. Finally, the case folding rules applied by such a wrapper are not guaranteed to be the same as those of the target directory.

As a result, we envision that defenses for name collisions will evolve in a manner similar to defenses for name confusions that utilize the `open` commands (i.e., `openat` and `openat2`). Consider that these commands have flags to check whether a file of a corresponding name exists at creation time, only opening that file when created anew (i.e., `O_CREAT|O_EXCL`). This call prevents a name collision from overwriting an existing file, but it may be too strong a defense. Suppose one really wants to overwrite files of the same name, but prevent name collisions from modifying files that actually have differing names (i.e., that only match due to case folding). In this case, a new flag is necessary, such as `O_EXCL_NAME`, which prevents opening a file when the names differ, but not when such names match. Using this flag would enable the virtual file system to compare names in a case-insensitive manner (i.e., based on the case folding and normalization for target directory) to detect collisions and compare names in a case-sensitive manner to determine matches. However, at present, the virtual file system cannot choose the type of matching (case-sensitive or case-insensitive), nor can it identify the type of matching done by the underlying physical file system.

Unfortunately, even with variants of the `open` command and other defenses, such as FileProvider classes in Android, programmers continue to make mistakes that lead to errors and vulnerabilities. The challenge is for programmers to determine the intent of their operation, understand the threats faced in such an operation, and configure these complex, low-level commands in such a way that they block the threats while satisfying the intent. Until file system APIs enable this combination of requirements, errors will remain common.

9 Related Work

Researchers have proposed defenses to thwart name confusion attacks for alias and squat cases. To the best of our knowledge, no defenses for name collisions have been proposed.

System Defenses Researchers have long known about name confusion attacks [3,4] and have proposed a variety of system defenses [7–9, 30, 40–42, 44, 50–52]. In a system defense, the operating system aims to enforce an invariant that prevents name confusion attacks from succeeding. However, as discussed in §8, without programmer intent such defenses will suffer from false positives [5]. Hybrid defenses have also been proposed [53, 55] where the operating system introspects into the process to leverage program state along with file system state in enforcement. Even though false positives are reduced, these techniques lack explicit programmer intent to fully eliminate all false positives.

Program Defenses As a result, systems provide APIs for programmers to decide how to handle name confusion attacks. Several file system APIs include flags to avoid using symbolic links entirely (e.g., `O_NOFOLLOW flag` for the `open` system call), but in many cases programmers want to be able to use symbolic links. Researchers have proposed program-specific defenses to configure APIs or program frameworks for preventing name confusion attacks [27, 43, 47, 56]. More advanced commands for file allow programmers to manage *how* files are open, including the impact of symbolic links. For example, the `openat` system call enables the user to open a directory first to validate its legitimacy before opening the remaining path. `openat2` explicitly constrains how name resolution is performed to reduce the potential for attacks.

10 Conclusion

Interactions among file systems with differing encoding/case-sensitivity semantics can lead to name collisions when performing maliciously crafted, or even ostensibly benign, copy operations. We explored the impact that these name collisions can have on file system security. Current operating systems do not directly prevent name collision-based attacks, delegating that responsibility to the programmers. In investigating the utilities used to copy file system resources and repositories/archives, we demonstrate that they often allow unsafe name collisions and lack the sort of uniformity in name-collision handling against which safer use policies could be easily crafted. Further, we show that many applications rely on potentially unsafe use of these utilities, opening themselves up to exploitable vulnerabilities. We examine three case studies demonstrating concrete vulnerabilities to name collisions. Finally, we suggest directions for future research to systematically defend against name collision attacks.

Artifacts

The artifacts produced during the work can be found at `https://github.com/mitthu/name-confusion`. It contains scripts to generate the test cases and run commands required to create Table 2a. Furthermore, it contains the tool for analyzing `auditd` traces and extracting relevant create-use pairs (see §5.2). Finally, there are proof-of-concept scripts to reproduce the vulnerabilities in `dpkg` and `rsync`.

References

[1] Apache HTTP Server: Authentication and authorization. https://httpd.apache.org/docs/2.4/howto/auth.html#gettingitworking.

[2] Bazaar's handling of case insentitive file systems. http://doc.bazaar.canonical.com/bzr.1.12/developers/case-insensitive-file-systems.html.

[3] Richard Bisbey, Gerald Popek, Jim Carlstedt, et al. Protection errors in operating systems: Inconsistency of a single data value over time. Technical report, University Of Southern California Marina Del Rey Information Sciences, 1975.

[4] Matt Bishop, Michael Dilger, et al. Checking for race conditions in file accesses. *Computing Systems*, 2(2):131–152, 1996.

[5] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting UNIX file-system races via algorithmic complexity attacks. In *2009 30th IEEE Symposium on Security and Privacy*, pages 27–41. IEEE, 2009.

[6] Case mapping vs. case folding. https://www.w3.org/TR/charmod-norm/#definitionCaseFolding.

[7] Suresh Chari, Shai Halevi, and Wietse Z. Venema. Where do you want to go today? Escalating privileges by pathname manipulation. In *NDSS*. Citeseer, 2010.

[8] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[9] Drew Dean and Alan J. Hu. Fixing races for fun and profit: How to use access (2). In *13th USENIX Security Symposium (USENIX Security 04)*, pages 195–206, 2004.

[10] dpkg FAQ: diff of updated documentation. https://wiki.debian.org/Teams/Dpkg/FAQ?action=diff&rev2=78&rev1=77.

[11] Dropbox. https://www.dropbox.com/.

[12] Linux kernel documentation (v5.2): ext4 support. https://www.kernel.org/doc/html/v5.2/admin-guide/ext4.html.

[13] Linux kernel documentation: Flash-friendly file system (F2FS). https://docs.kernel.org/filesystems/f2fs.html.

[14] F2FS: Support case-insensitive file name lookups (patch). https://patchwork.kernel.org/project/linux-fsdevel/patch/20190719000322.106163-3-drosen@google.com/.

[15] ciopfs: case insensitive on purpose filesystem. https://www.brain-dump.org/projects/ciopfs/.

[16] ext3ci – case insensitive ext3 filesystem for Linux 2.6.32. http://bill.herrin.us/freebies/.

[17] Linux kernel documentation: FUSE. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[18] IOMap. https://www.mono-project.com/docs/advanced/iomap/.

[19] JFS filesystem (man page). https://www.unix.com/man-page/redhat/8/mkfs.jfs/.

[20] Linux kernel documentation: NTFS3. https://docs.kernel.org/filesystems/ntfs3.html.

[21] NTFS-3G driver. https://github.com/tuxera/ntfs-3g.

[22] XFS filesystem (man page). https://manpages.ubuntu.com/manpages/trusty/man8/mkfs.xfs.8.html.

[23] ZFS on Linux project. https://zfsonlinux.org/.

[24] Git's patch for CVE-2021-21300. https://github.com/git/git/commit/684dd4c2b414bcf648505e74498a608f28de4592.

[25] Norman Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22:36–38, October 1988.

[26] Daniel Kachakil. Multiple vulnerabilities in Android's Download Provider (CVE-2018-9468, CVE-2018-9493, CVE-2018-9546). https://ioactive.com/multiple-vulnerabilities-in-androids-download-provider-cve-2018-9468-cve-2018-9493-cve-2018-9546/.

[27] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. *ACM SIGOPS Operating Systems Review*, 41(6):321–334, 2007.

[28] James A. Kupsch and Barton P. Miller. How to open a file and not get hacked. In *2008 Third International Conference on Availability, Reliability and Security*, pages 1196–1203. IEEE, 2008.

[29] Yu-Tsung Lee, William Enck, Haining Chen, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Daimeng Wang, Giuseppe Petracca, and Trent Jaeger. PolyScope: Multi-Policy access control analysis to compute authorized attack operations in android

systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2579–2596, 2021.

[30] Kyung-Suk Lhee and Steve J. Chapin. Detection of file-based race conditions. *International Journal of Information Security*, 4(1):105–119, 2005.

[31] Linus Torvalds's comments on case-insensitive file systems. https://patchwork.kernel.org/project/linux-fsdevel/cover/20181206230903.30011-1-krisman@collabora.com/#22369005.

[32] Eliminating Android wrapfs "hackery". https://lwn.net/Articles/718640/.

[33] mm: shmem: Add case-insensitive support for tmpfs. https://lwn.net/Articles/850214/.

[34] Case-insensitive ext4. https://lwn.net/Articles/784041/.

[35] Filesystems and case-insensitivity. https://lwn.net/Articles/772960/.

[36] Case-insensitive filesystem lookups. https://lwn.net/Articles/754508/.

[37] Network filesystem topics. https://lwn.net/Articles/685431/.

[38] Slava Makkaveev. Man-in-the-Disk: Android apps exposed via external storage. https://research.checkpoint.com/2018/androids-man-in-the-disk/.

[39] William S. McPhee. Operating system integrity in OS/VS2. *IBM Systems Journal*, 13(3):230–252, 1974.

[40] OpenWall Project - Information security software for open environments. http://www.openwall.com/.

[41] Jongwoon Park, Gunhee Lee, Sangha Lee, and Dong-Kyoo Kim. RPS: An extension of reference monitor to prevent race-attacks. In *Pacific-Rim Conference on Multimedia*, pages 556–563. Springer, 2004.

[42] Mathias Payer and Thomas R. Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 215–226, 2012.

[43] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 161–176, 2009.

[44] Calton Pu and Jinpeng Wei. A methodical defense against TOCTTOU attacks: The EDGI approach. In *Proceedings of the 2006 International Symposium on Secure Software Engineering*, 2006.

[45] Samba: Implementation of SMB/CIFS protocol. https://www.samba.org/.

[46] smb.conf.5 (man). https://www.samba.org/samba/docs/4.15/man-html/smb.conf.5.html.

[47] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, 1999.

[48] Case-sensitive filesystems not supported on Mac. https://help.steampowered.com/en/faqs/view/0395-A862-13F3-6E82.

[49] SteamOS. https://store.steampowered.com/steamos.

[50] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *FAST*, volume 8, pages 1–18, 2008.

[51] Eugene Tsyrklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[52] Prem Uppuluri, Uday Joshi, and Arnab Ray. Preventing race condition attacks on file-systems. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 346–353, 2005.

[53] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. JIGSAW: Protecting resource access by inferring programmer expectations. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 973–988, 2014.

[54] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. STING: Finding name resolution vulnerabilities in programs. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 585–599, 2012.

[55] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. Process firewalls: Protecting processes during resource access. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 57–70, 2013.

[56] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium (USENIX Security 10)*, 2010.

[57] Wine: Wine is not an emulator. https://www.winehq.org/.

[58] What is the Windows subsystem for Linux? https://docs.microsoft.com/en-us/windows/wsl/about.

[59] Diving into SDCardFS: How Google's FUSE replacement will reduce I/O overhead. https://www.xda-developers.com/diving-into-sdcardfs-how-googles-fuse-replacement-will-reduce-io-overhead/.

ConfD: Analyzing Configuration Dependencies of File Systems for Fun and Profit

Tabassum Mahmud, Om Rameshwar Gatla, Duo Zhang, Carson Love, Ryan Bumann, Mai Zheng
Department of Electrical and Computer Engineering, Iowa State University, Ames, IA

Abstract

File systems play an essential role in modern society for managing precious data. To meet diverse needs, they often support many configuration parameters. Such flexibility comes at the price of additional complexity which can lead to subtle configuration-related issues. To address this challenge, we study the configuration-related issues of two major file systems (i.e., Ext4 and XFS) in depth, and identify a prevalent pattern called multilevel configuration dependencies. Based on the study, we build an extensible tool called ConfD to extract the dependencies automatically, and create six plugins to address different configuration-related issues. Our experiments on Ext4 and XFS show that ConfD can extract more than 150 configuration dependencies for the file systems with a low false positive rate. Moreover, the dependency-guided plugins can identify various configuration issues (e.g., mishandling of configurations, regression test failures induced by valid configurations).

1 Introduction

File systems (FS), such as Ext4 [54] and XFS [89] on Linux-based operating systems (OS) and NTFS [76] on Windows OS, play an essential role in modern society. They directly manage various files on desktops, laptops, and smartphones for numerous end users [12]. Moreover, they often serve as the local storage backend for distributed storage systems (e.g., Lustre [63], GFS [1], HopsFS [22], MySQL NDB Cluster [73]) to enable storage management at scale.

To meet diverse needs, many file systems are designed with a wide range of configuration parameters controllable via utilities [41, 45, 48, 52, 56, 58, 94, 100], which enables users to tune the systems with different tradeoffs. For example, Ext4 contains more than 85 configuration parameters which can be modified through a set of utilities called `e2fsprogs` [52]. The combination of the configuration parameters represents over 10^{37} configuration states [32].

While configuration parameters have improved the system flexibility, they introduce additional complexity for reliability. Subtle correctness issues often rely on specific parameters to trigger [6, 13]; consequently, they may elude intensive testing and affect end users negatively. For example, in December 2020, users of Windows OS observed that the checker utility of NTFS (i.e., `ChkDsk` [45]) may destroy NTFS on SSDs [60, 88]. The incident turned out to be configuration-related: two specific parameters must be satisfied to manifest the issue, including the '`/f`' parameter of `ChkDsk` and another (unnamed) parameter in Windows OS [87].

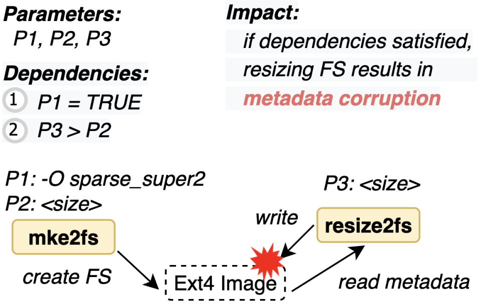


Figure 1: **A Configuration-Related Issue of Ext4.** When `sparse_super2` feature is enabled and the `size` parameter of `resize2fs` is larger than the Ext4 size, expanding the file system results in metadata corruption.

Similarly, Figure 1 shows another configuration-related issue involving Ext4 and its `mke2fs` and `resize2fs` utilities [52]. Two conditions must hold to trigger the bug: (1) the `sparse_super2` feature is enabled in Ext4 (via `mke2fs`); (2) the value of the `size` parameter of `resize2fs` must be larger than the size of Ext4 (i.e., expanding the file system). Once triggered, the bug will corrupt the Ext4 metadata with incorrect free blocks. The root cause behind the issue was logical: with the specific configuration, the free block count of the last block group of Ext4 was calculated before adding new blocks for expansion.

Due to the combinatorial explosion of configuration states and the substantial time needed to scrutinize a file system under each configuration state, it is practically impossible to exhaust all states for thorough testing today [9]. Moreover, with more and more heterogeneous devices and advanced features being introduced [65, 83, 86], the configuration states are expected to grow. Therefore, effective methods to help improve configuration-related testing and identify critical configuration issues efficiently are much needed.

1.1 Limitations of the State of the Art

There are practical test suites to ensure the correctness of file systems under different configurations (e.g., `xfstests` [95]). Unfortunately, their coverage in terms of configuration is limited: fewer than half of configuration parameters are used based on our study, which reflects the need for better tool support. Also, configuration-related issues have emerged in other software systems and have received much attention [4, 13, 24, 33, 35]. But unfortunately, existing efforts mainly focus on relatively simple configuration issues (e.g., typos [4]) within one single application, which is limited for addressing the file system configuration challenge involving multiple programs. Please refer to §2 for more details.

1.2 Our Efforts & Contributions

This paper presents one of the first steps to address the increasing configuration challenge of file systems. Inspired by a recent study [33] on configuration issues in Hadoop [40] and OpenStack [77], we focus on *configuration dependency*, which describes the dependent relations among configuration parameters [33]. Such dependency has been identified as a key source of complexity caused problems, and capturing the dependency is essential for improving configuration design and tooling [13, 19, 33].

While the basic concept of configuration dependency has been proposed in the literature (see §2), the understanding of specific dependency patterns and implications in the context of file systems is still limited. Therefore, we first conducted an empirical study on 78 configuration-related issues in two major file systems (i.e., Ext4 and XFS). By scrutinizing real-world bugs and the relevant source code, we answer one important question: What critical configuration dependencies exist in file systems?

Our study reveals a prevalent pattern called *multilevel configuration dependencies*. Besides the relatively simple configuration constraints (e.g., value range [13]), there are implicit dependencies among parameters from different utilities of a file system. The majority (96.2%) of issues in our dataset requires meeting such deep configuration dependencies to manifest. Interestingly, the workloads applied to the file system do not have to be configuration-specific: 71.8% issues only involve generic file system operations.

Based on the study, we built an extensible framework called CONFD to extract the multilevel configuration dependencies automatically and leverage dependency-guided configuration states for further analysis. One key challenge is how to establish the correlation between parameters specified through different utilities which have different ways of configuration handling. We address the challenge by metadata-assisted taint analysis, which leverages the fact that all utilities of a given file system share the same metadata structures. Moreover, based on the dependencies extracted, we created six plugins to help address configuration-related issues in file systems from different angles.

Our experiments show that CONFD can extract 154 different configuration dependencies with a low false positive rate (8.4%) for Ext4 and XFS. Moreover, with the dependency guidance, the CONFD plugins can identify various configuration-related issues, including inaccurate documentations, configuration handling issues, and regression test failures induced by valid configurations.

In summary, this paper makes the following contributions:

- Deriving a taxonomy of critical configuration dependencies of file systems based on real-world issues.
- Building the CONFD prototype [1] to extract configuration dependencies and expose relevant issues in file systems.
- Integrating with multiple practical tools (e.g.,fault injector [25], fuzzer [29], regression test suites [51, 95]) to improve their configuration coverage and effectiveness.
- Evaluating the methodology on two widely used file systems and demonstrating the effectiveness.

The rest of the paper is organized as follows: §2 introduces the background and related work; §3 presents the empirical study and findings; §4 describes the CONFD framework; §5 shows experimental results; §6 discusses limitations and potential extensions; §7 concludes the paper.

2 Background & Related Work

2.1 Background

File System Configurations. The configuration methods of file systems are different from that of many applications, which makes the problem arguably more challenging. As shown in Figure 2, a typical file system may be configured through a set of utilities at four different stages:

- **Create**. When creating file systems, the `mkfs` utility (e.g., `mke2fs` for Ext4) generates the initial configurations.
- **Mount**. When mounting file systems, certain configurations can be specified via `mount` (e.g., '`-o dax`' to enable the Direct Access or DAX feature [65]).

[1] CONFD is on `https://github.com/data-storage-lab/ConfD`

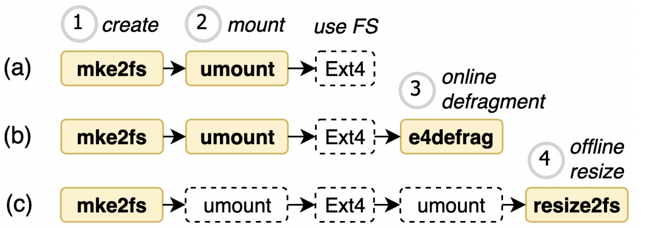


Figure 2: **Methods of Configuring File Systems**. This figure shows four typical stages to configure a file system: (a) at creation (e.g., `mke2fs`) or mount time (`mount`) before usage; (b) via online utilities (e.g., `e4defrag`); (c) via offline utilities.

- **Online**. Many utilities can change the configurations of a mounted file system directly by modifying the metadata online (e.g., Ext4 defragmenter `e4defrag` [53], Windows NTFS checker `ChkDsk` [45]).
- **Offline**. Offline utilities can also modify file system images and change the configurations (e.g., `resize2fs` [79], `e2fsck` [50])

Note that all the utilities have different configuration parameters to control their own behaviors, which will eventually affect the file system state. Moreover, the configuration parameters may affect the behavior of the file system long after the FS image is created, and some configurations cannot be changed later. Also, the validation of parameters may occur at both user level and kernel level. For example, the '`-O inline_data`' parameter of `mke2fs` and the '`-o dax`' of `mount` are further validated in the `ext4_fill_super` function of Ext4. Therefore, we believe it is necessary to consider the file system itself as well as all the associated utilities as an *FS ecosystem* to address the configuration challenge. For simplicity, we call the file system and utilities as *components* within the FS ecosystem.

The multi-stage configuration method is common among file systems. As listed in Table 1, many popular file systems follow similar modular designs and can be configured via different utilities at different stages. Therefore, we believe that the multi-component configuration challenge is general. We focus on Ext4 and XFS in this work because they are two major file systems on Linux and they support the latest DAX [65] configuration for non-volatile memories (NVM). We leave the others as future work (§6).

FS Test Suites. Practical test suites have been created to ensure the correctness of file systems under various configurations. Unfortunately, due to the complexity of configurations, their coverage in terms of configuration is limited. As shown in Table 2, fewer than half of configuration parameters are used in the standard test suites of Linux file systems (i.e., `xfstests` [95], `e2fsprogs/tests` [51]) based on our study. Since each parameter may have a wide range of values representing different states, the total number of missed configuration states is much more than the number of unused parameters, which implies the need for better tool support.

FS (OS)	Four Stages of Configuration			
	Create	Mount	Online	Offline
Ext4 (Linux)	[66]	[69]	[53], [79]	[50], [79]
XFS (Linux)	[68]	[69]	[91], [92]	[90], [93]
BtrFS (Linux)	[67]	[69]	[41], [43]	[42]
UFS (FreeBSD)	[75]	[70]	[59], [80]	[49], [58]
ZFS (FreeBSD)	[96]	[98]	[99], [100]	[97]
NTFS (Windows)	[55]	[72]	[45], [46]	[45], [81]
APFS (MacOS)	[48]	[71]	[48]	[48], [57]

Table 1: **Examples of configuration methods for different file systems**. The last four columns list example utilities that can affect the file system configuration states.

Test Suite	Target Software	# of Conf. Param.	
		Total	Used
`xfstests`	Ext4	>85	29 (< 34.1%)
`e2fsprogs/tests`	`e2fsck`	>35	6 (< 17.1%)
	`resize2fs`	>15	7 (< 46.7%)

Table 2: **Configuration Coverage of Test Suites.**

Configuration Constraints & Dependencies. Configuration *constraints* specify the configuration requirements (e.g., data type, value range) of software [13]. Intuitively, such information can help identify important configuration states, and it has proved to be effective for addressing configuration-related issues in a wide range of applications [4, 8, 13, 14, 33]. Configuration *dependency* is one special type of constraint describing the dependent correlation among parameters [13, 33], which has shown recently to be critical for addressing complex configuration issues in cloud systems [33]. For simplicity, we use constraints and dependencies interchangeably in the rest of the paper. Note that although the basic concepts have been proposed, there is limited understanding of them in the context of file systems. This paper attempts to fill the gap.

2.2 Related Work

Analysis of Software Configurations. Configuration issues have been studied in many software applications [4, 6, 7, 13, 14, 24, 33, 35]. For example, ConfErr [4] manipulates parameters to emulate human errors; Ctests [35] detects failure-inducing configuration changes. In general, these works do not analyze deep dependencies within the software. The closest work is cDEP [33], which notably observes *inter-component dependencies* in Hadoop [40] and OpenStack [77]. Unfortunately, their solution is largely inapplicable for file systems. This is because their target components share configuration specifications (e.g., XML) and libraries [39], which makes them equivalent to one single program in terms of configuration. In contrast, the configuration dependencies in file systems may cross different programs and the user-kernel boundary, which requires non-trivial mechanisms to extract.

In addition, cDEP relies on a Java framework [82] which cannot handle C-based file systems.

Reliability of File Systems. Great efforts have been made to improve the reliability of file systems [2, 10, 17, 18, 29] and their utilities [3, 25, 26, 27, 78]. For example, Prabhakaran et al. [2] apply fault injection to analyze the failure policies of file systems and propose improved designs based on the IRON taxonomy; Xu et al. [30] and Kim et al. [29] use fuzzing to detect file system bugs; SQCK [3] and RFSCK [25] improve the checker utilities of file systems to avoid inaccurate fixes. While effective for their original goals, these works do not consider multi-component configuration issues. On the other hand, the configuration dependencies from this work may be integrated with these existing efforts to improve their coverage (see §4.2). Therefore, we view them as complementary.

Configuration Management Tools. Faced by the increasing challenge, practitioners have created dedicated frameworks for configuration management [31, 44]. For example, Facebook HYDRA [31] supports managing hierarchical configurations elegantly. While helpful for developing new applications, refactoring FS ecosystems to leverage such frameworks would require substantial efforts (if possible at all). Notably, the framework supports running a program with different compositions of configurations automatically. Nevertheless, since it does not understand configuration dependencies, it may generate many invalid configuration states (see §5.2.3). This work aims to address such limitations.

3 Configuration Dependencies in File Systems

In this section, we present a study on the Ext4 and XFS ecosystems to understand the potential patterns of configuration issues and guide the design of solutions. We discuss the methodology and findings in §3.1 and §3.2, respectively.

3.1 Methodology

Our dateset includes two parts: (1) the source code of Ext4 and XFS and seven important utilities including `mke2fs`, `mount`, `e4defrag`, `resize2fs`, `e2fsck`, `mkfs.xfs`, and `xfs_repair`, which are described in Table 3; (2) a set of 78 configuration-related bug patches for the two FS ecosystems, which are collected from the commit histories of their source code repositories via a combination of keyword search (e.g., 'configuration', 'parameter', 'option'), random sampling, and manual validation. Note that the patch collection method is inspired by previous studies of real-world bugs [5, 12, 36]. While time-consuming, it has proved to be valuable for driving system improvements [5, 12]. On the other hand, similar to previous studies, the findings of our study should be interpreted with the method in mind. For example, the 78 patches only represent a subset of issues that have been triggered and fixed; there are likely other configuration-related issues not yet discovered (see §6 for further discussion).

3.2 Findings

Based on the dataset, we analyzed each patch and the relevant source code in depth to understand the logic, which enables us to identify the configuration usage scenarios as well as configuration constraints that are critical. We summarize our findings in Table 3 and Table 4 and discuss them below.

Finding #1: *The majority of cases (96.2%) involve critical parameters from more than one component.* The first column of Table 3 shows six typical usage scenarios of file systems which cover all bug cases in our dataset (78 in total). 96.2% of the bug cases require specific parameters from at least two key utilities (i.e., the utilities in bold in each usage scenario) to manifest. This reflects the complexity of the issues and suggests that we cannot only consider one single component.

Finding #2: *There is a hierarchy of configuration dependencies.* We classify the configuration constraints derived from our dataset into three major categories as follows:

- **Self Dependency (SD)** means individual parameters must satisfy their own constraints (e.g., data type or value range). For example, the `blocksize` parameter of `mke2fs` has a value range of 1024 - 65536 and must be a power of 2.
- **Cross-Parameter Dependency (CPD)** means multiple parameters of the same component must satisfy relative relation constraints (e.g., two `mke2fs` parameters `meta_bg` and `resize_inode` cannot be used together).
- **Cross-Component Dependency (CCD)** means the parameters or behaviors of one component depend on the parameters of another component. Both dependencies in Figure 1 belong to this category becasue they involve parameters of `mke2fs` and the (buggy) behavior of `resize2fs` depend on them.

As summarized in Table 4, each major category may contain a couple of sub-categories which describe more specific constraints. Together, these constraints form a hierarchy which we call *multilevel configuration dependencies*. Note that we only observed 7 out of 8 sub-categories in the dataset. We include the unseen "Value" sub-category in CPD based on the literature [13] for completeness.

Moreover, among all the dependencies, there is a subset which directly contribute to the manifestation of the bugs in our dataset: the relevant parameters are explicitly mentioned in the bug patches, and modifications to the corresponding functionalities are needed to fix the bugs (i.e., they are related to the root causes). We call this subset of dependencies as *critical dependencies*. The count of the critical dependencies for each sub-category is shown in the last column of Table 4. We are able to derive 168 critical dependencies manually in total, which is larger than the number of bug cases. This is because multiple critical dependencies may be needed to

FS Usage Scenarios (key configuration utilities are in bold)		**Description**	**# of** **Bug**	**Multilevel Config. Dependencies**		
				SD	**CPD**	**CCD**
1	**mke2fs** - **mount** - Ext4	create & mount an Ext4 to use	13	13 (100%)	1 (7.7%)	13 (100%)
2	**mke2fs** - **mount** - Ext4 - **e4defrag**	online defragmentation	1	1 (100%)	–	–
3	**mke2fs** - mount - Ext4 - umount - **resize2fs**	resize an umounted Ext4	17	17 (100%)	–	17 (100%)
4	**mke2fs** - mount - Ext4 - umount - **e2fsck**	check Ext4 & fix inconsistencies	36	36 (100%)	4 (11.1%)	34 (94.4%)
5	**mkfs.xfs** - **mount** - XFS	create & mount an XFS to use	5	5 (100%)	2 (40%)	5 (100%)
6	**mkfs.xfs** - mount - XFS - umount - **xfs_repair**	check XFS & fix inconsistencies	6	6 (100%)	1 (16.7%)	6 (100%)
		Total	78	78 (100%)	8 (10.3%)	75 (96.2%)

Table 3: **Distribution of Configuration Bugs in Six Scenarios.** This table shows the distribution of 78 configuration bugs in six typical usage scenarios of file system. The last three columns shows the percentages of bug cases that involve Self-Dependency (SD), Cross-Parameter Dependency (CPD), and Cross-Component Dependency (CCD), respectively.

Multilevel Config. Dependencies		**Description**	**Observed?**	**Count**
Self Dependency (SD)	Data Type	parameter P must be of a specific data type (e.g., integer)	Y	44
	Value Range	P must be within a specific value range (e.g., $P < 4096$)	Y	41
Cross-Parameter Dependency (CPD)	Control	$P1$ of $C1$ can be enabled iff $P2$ of $C1$ is enabled/disabled	Y	5
	Value	$P1$'s value depends on $P2$'s value (e.g., $P1 < P2$)	N	–
	Behavioral	component $C1$'s behavior depends on $P1$ and $P2$ of component $C1$	Y	1
Cross-Component Dependency (CCD)	Control	$P1$ of $C1$ can be enabled iff $P2$ of $C2$ is enabled/disabled	Y	1
	Value	$P1$'s value depends on $P2$ from another component	Y	1
	Behavioral	component $C1$'s behavior depends on $P2$ of $C2$	Y	75
		Total	7/8	168

Table 4: **Multilevel Configuration Dependencies.** This table describes the multilevel configuration dependencies observed. *Pn* means parameter, *Cn* means component. The last column shows the count of each sub-category of dependency observed.

trigger a bug. For example, both dependencies in Figure 1 are critical dependencies for this bug case.

As shown in the last three columns of Table 3, SD and CCD are almost always involved in all scenarios (100% and 96.2% respectively), while CPD is non-negligible (10.3%). This is because SD represents relatively simple constraints which always need to be satisfied first to make the target component work (e.g., correct spelling). SD is relatively easy to check and has been the focus of previous work [4]. However, this does not mean that 100% of the bugs could be avoided if SD is checked or satisfied. For example, a bug related to both the `bigalloc` and `extent` parameters (i.e., there is a CPD involved) may still occur even if the two parameters are spelled correctly. In other words, only considering simple constraints of individual parameters is not enough.

Interestingly, we observed both CPD and CCD between the DAX feature and other seemingly irrelevant configurations. In one case, a corruption was triggered when '`-O inline_data`' was used in `mke2fs` and the image was mounted with '`-o dax`' subsequently. In another case, the DAX feature conflicted with the '`has_journal`' configuration, which may lead to corruptions when changing the journaling mode online. Such unexpected dependencies implies the complexity of adding the DAX support to the Linux kernel.

Finding #3: *Configuration parameters are handled in heterogeneous ways in an FS ecosystem.* We identified four major sources of heterogeneity in FS configurations. First, different parameters may be mapped to different types of variables in the code. For example, the parameters of Ext4 may be stored in (at least) four different ways including (i) a local variable, (ii) a global variable, (iii) a bit in a bitmap accessed via bit operations, and (iv) directly in the superblock. Second, within the superblock, parameters may be kept either in one single field (e.g., `s_log_block_size`) or as one member of a compound field. Third, parameters can be loaded from the superblock either directly or through library calls. Lastly, different components may use different functions for handling configurations (e.g., `resize2fs` uses the "main" function, while `mke2fs` invokes a special function called "PRS"). Such heterogeneity makes previous solutions mostly inapplicable.

Finding #4: *The majority of cases (71.8%) do not require configuration-specific workloads to manifest.* Interestingly, despite the complexity, many bugs can be triggered without applying configuration-specific workloads. This suggests that we may re-use existing efforts on stressing file systems [51, 95] to analyze configuration-related issues effectively.

4 Extracting & Using Multilevel Configuration Dependencies

Based on the study, we built an extensible framework called ConfD to leverage the dependency information to address configuration-related issues. As shown in Figure 3, ConfD consists of two main parts: (1) *ConfD-core* (yellow box) for extracting multilevel configuration dependencies and generat-

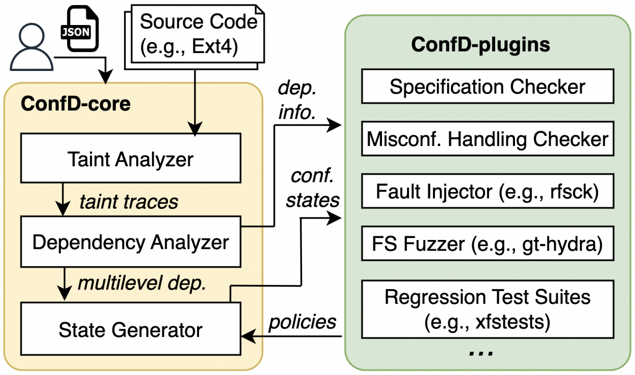


Figure 3: **Overview of CONFD**. There are two parts: (1) ConfD-core (yellow) for extracting configuration dependencies and generating critical states; (2) ConfD-plugins (green) for detecting various configuration-related issues.

ing critical configuration states, which further contains three sub-modules (i.e., *Taint Analyzer*, *Dependency Analyzer*, and *State Generator*); (2) *ConfD-plugins* (green box) for detecting various configuration-related issues based on the generated configuration states. We elaborate on the two parts in the following two subsections respectively.

4.1 Extracting Configuration Dependencies

4.1.1 Metadata-assisted Taint Analysis

As the first step, the *Taint Analyzer* of CONFD performs metadata-assisted taint analysis and generates taint traces to capture the propagation flow of configuration parameters in the target FS ecosystem.

It takes the source code of the target system as input, and uses the LLVM compiler infrastructure [85] to generate intermediate representation (IR) of the source code. It then tracks the propagation of each configuration parameter along the data-flow paths in IR based on the classic taint analysis algorithm [21]. We maintain a set to keep the initial configuration variables and any variables derived from the initial configuration variables while traversing the IR. When a new variable is added to the set, we add the corresponding IR instruction to the taint trace. We maintain a mapping between each configuration parameter and the variables derived from it to enable tracking if a variable may be derived from multiple parameters, which is essential for establishing the correlation across parameters. Our taint analysis is context-sensitive and can handle both intra-procedural and inter-procedural analysis. Context-sensitivity is important for inter-procedural analysis because one function can be called from different contexts, which is also crucial for deriving accurate dependency across different taint traces (§4.1.2).

One unique challenge we encounter is how to establish the mapping between parameters of different components of the FS ecosystem. As mentioned in §3.2, the components in the FS ecosystem tend to load configurations in different ways and process equivalent FS information using different variables or functions. We address this challenge based on one key observation: all components need to access the same FS metadata structures. We can leverage shared metadata structures to connect relevant parameters of different components.

More specifically, the parameter values relevant to the FS configuration are (eventually) stored in the superblock structure of the file system. For example, the parameter `-I inode-size` from `mke2fs` is stored as the 27th member of the superblock (`s_inode_size`). When another component (e.g., `e2fsck`) loads the `s_inode_size` from the superblock to access it, it is essentially dependent on the `-I inode-size` parameter of `mke2fs`. We map the `mke2fs` parameter values to relevant superblock fields by tracking where the parameter value is being written in the superblock. Similarly, the accesses to the superblock in other components are also tracked. Based on the mapping to the same superblock fields (e.g., `s_inode_size`), we can establish the connection between taint traces from different components.

Note that since CONFD implements the taint analysis at the LLVM IR level, any file system that can be compiled to LLVM IR may benefit from it for configuration dependency analysis. The current prototype uses the Clang frontend of LLVM which supports C/C++/Objective-C languages [85].

4.1.2 Multilevel Dependency Analysis

Given the taint trace of every configuration parameter, the *Dependency Analyzer* further analyzes the potential correlations between parameters based on the multilevel dependencies derived from our study (§3).

Specifically, the self-dependency (SD) for each parameter is derived from their individual taint traces based on the data type and value range of the variables. We also examine the error statement immediately following a range check based on the observation that an error statement may indicate an invalid range. For CPD and CCD, we compare taint traces of multiple parameters. If there are common lines (which are context-sensitive), we consider them to be dependent. Moreover, after getting the dependent parameters, we also leverage the subsequent error statements to further analyze the specific types of dependency (e.g., should be enabled or disabled together). For example, the two parameters `resize_inode` and `meta_bg` from `mke2fs` cannot be enabled together, so there must be a common error statement immediately following the condition check shared by the two taint traces. All of the extracted dependencies are stored in the JSON format [61] to describe both the parameters and the corresponding dependent relations concisely.

4.1.3 Dependency-guided State Generation

With the dependency information, the *State Generator* generates concrete configuration states for further analysis. Instead of randomly generating combinations of configurations which may easily lead to useless states (§5.2.3), it leverages the extracted multilevel dependencies to generate states selectively.

Specifically, the *State Generator* uses a tree structure to maintain different configuration states. The root of the tree represents a default configuration state, and each child node on the tree represents a configuration state with exactly one modification made from its parent. The module operates similar to a Depth First Search (DFS) on a tree, except it leverages the dependency information to guide which children nodes are worth pursuing. For example, given the cross-parameter dependency (CPD) between the `bigalloc` and `blocksize` parameters of `mke2fs`, if the current node modifies `bigalloc`, then the child node to consider will be a state with a modification to `blocksize`.

Moreover, the module has a number of options that allow for tuning based on needs. The first option is 'depth', which dictates how deep the DFS is allowed to go. A larger value results in a greater number of states being generated. The default 'depth' is 3 which worked well in our experiments. Another option is the 'policy' under which the State Generator operates. There are two basic policies as follows:

Following Dependency. Under this policy, we always honor the extracted multilevel dependencies when creating a configuration state. For example, `sparse_super` should always be enabled if `resize_inode` is enabled for `mke2fs` according to the multilevel dependency, so the module may generate a state with both parameters (i.e., '`mk2fs -O resize_inode,sparse_super`'). Essentially, this policy only generates *valid* configuration states involving critical parameters for the target FS ecosystem, which is the basic requirement for running many FS applications or tools properly. Note that this policy is consistent with recent work on testing configuration changes which shows that *valid* configuration changes may induce production failures [35].

Violating Dependency. Under this policy, we intentionally violate the multilevel dependencies when creating a configuration state. For example, the `resize_inode` and `sparse_super` parameters of `mke2fs` have a cross-parameter dependency (CPD): `sparse_super` must be enabled if we want to enable `resize_inode`. To violate the CPD, the module may intentionally generate a state which disables the `sparse_super` parameter while enabling `resize_inode` (i.e., '`mke2fs -O resize_inode,^sparse_super`'). By generating *invalid* configuration states on purpose, we enable examining the (mis)configuration handling of the target system. Note that this policy is inspired by the previous work on simulating human errors in configuration [4]. However, different from the relatively shallow violations (e.g., typos) which have been largely handled in matured systems, we consider more subtle violations that involve non-trivial dependencies.

In addition, to provide more flexibility for different use cases, the *State Generator* supports customizing the two basic policies further with different tradeoffs (e.g., the number of parameters to consider, the type of dependency (i.e., SD/CPD/CCD) to use). As mentioned, a key challenge with analyzing configurations of file systems is that the space is too huge to exhaust. For example, `mke2fs` itself has more than 8 trillion possible parameter combinations. With the dependency guidance, CONFD can reduce the space to hundreds or tens of thousands depending on the use case (§4.2), which makes the configuration testing much more manageable in practice. And as will be shown in §5.2.1, the dependency-guided state generation will be more effective than dependency-agnostic alternatives for exposing configuration issues.

4.1.4 User Input

ConfD-core needs three types of input information from the user, which can be specified in one single JSON file. First, to start the taint analysis, the *Taint Analyzer* needs a function name as the entry point. In the case of a utility program, the function (which may invoke sub-functions) is expected to be the major function for processing configurations. In the case of the file system itself, the function can be either a function for processing configurations, or a function that is interesting (e.g., a newly added FS function). Second, the taint analysis also requires the names of the variables representing the configurations and the superblock in the source code, which are often different across programs based on our experience on Ext4 and XFS ecosystems. Third, to generate valid configuration states, the *State Generator* needs the command-line syntax of FS configurations. Note that all the input can be specified in the JSON format, and it is a one-time effort for each program to be analysed.

4.2 Leveraging Configuration Dependencies

The dependency information and the dependency-guided configuration states may be used in different ways to address different issues [4, 13, 33]. As mentioned in §2.2, there are existing efforts to improve FS ecosystems which cover a wide range of techniques including fault injection [2, 25], fuzzing [29, 30], regression test suites [51, 95], etc. While these tools are excellent for their original design goals, they are mostly agnostic to configuration dependencies and thus cannot address tricky configuration-related issues effectively. The CONFD plugin interface is designed to bridge the gap by introducing dependency awareness to the traditional methodologies and thus amplify the effectiveness.

The current prototype of CONFD includes six plugins. As summarized in Table 5, the first two plugins (#1 and #2) are built from scratch, the next two plugins (#3 and #4) are based on open-source research prototypes (R), and the last two (#5 and #6) are designed for enhancing standard test suites (S). We discuss them in more details below:

Plugin ID	Description	Base Tool (type)	ConfD Plugin
#1	Configuration specification checker for Linux file systems	N/A	`ConfD-specCk`
#2	Misconfiguration handling checker for Linux file systems	N/A	`ConfD-handlingCk`
#3	An open-source fault injector for file system utilities	`rfsck` [25] (R)	`ConfD-rfsck`
#4	An open-source fuzzer for file systems	`gt-hydra` [29] (R)	`ConfD-gt-hydra`
#5	Regression test suite for Linux file systems	`xfstests` [95] (S)	`ConfD-xfstests`
#6	Regression test suite for Ext4 utilities	`e2fsprogs/tests` [51] (S)	`ConfD-e2fsprogs`

Table 5: **Summary of ConfD Plugins.** 'Base Tool' means existing tools that have been integrated with ConfD through the corresponding plugins; 'R' means open-source Research prototype, 'S' means Standard test suites for file systems and utilities.

Plugin #1: Configuration Specification Checker. The specifications for the configurations of Linux file systems are maintained through the Linux man-pages project [84]. Unfortunately, due to a variety of reasons (e.g., constant system upgrades, feature additions, bug fixes), the specifications may become inaccurate easily, which may confuse end users and/or lead to configuration-induced failures [35, 64]. The `ConfD-specCk` plugin is designed to mitigate the problem. It parses the Linux man-pages related to the file system configurations (e.g., `mke2fs`, `mkfs.xfs`) and checks a subset of multilevel dependencies (Table 4) based on keywords. For example, `resize_inode` and `meta_bg` cannot be enabled together for `mke2fs` (i.e., CPD), so `meta_bg` should appear in the description of `resize_inode` with 'disable' (or similar keywords) and vice versa. Similarly, value ranges (i.e., SD) and other value dependencies (e.g., `cluster_size` needs to be 'equal' or 'greater' than `block_size`) should also be specified in the descriptions accordingly. Such dependencies from man-pages are stored in the JSON format for further comparison with the dependencies extracted from the source code by *ConfD-core* (§4.1). A mismatch implies a potential specification issue.

Plugin #2: Misconfiguration Handling Checker. A well designed file system should be able to handle wrong configurations from end users (either by mistake or by intention) gracefully. Failing to handle misconfigurations elegantly implies *misconfiguration vulnerabilities* that could hurt system reliability and/or security [13]. The `ConfD-handlingCk` plugin is designed to expose the potential issues in misconfiguration handling. Thanks to the built-in 'Violating Dependency' policy (§4.1.3), the plugin can directly leverage the invalid configuration states generated by ConfD which violate inherent configuration dependencies. It applies such automatically generated misconfigurations to drive the target file systems and utilities, and records the symptoms accordingly for postmoterm analysis.

Plugin #3: Dependency-aware Fault Injector. Fault injection techniques have been applied to improve both file systems and utilities [2, 15, 25, 28, 51]. By systematically generating corrupted file system states, they enable analyzing the robustness of FS ecosystems thoroughly. However, given the complexity of file system metadata, one open challenge is how to generate vulnerable states efficiently. To mitigate the challenge, we integrate one open-source fault injector `rfsck` [25] with ConfD through the `ConfD-rfsck` plugin. Instead of relying on the default configuration, `ConfD-rfsck` leverages dependency-guided configurations to generate input images to initiate the fault injection campaign. Since the input images are configured with dependent parameters identified by ConfD, they represent more complicated states that are more difficult to remain consistent under fault. Note that the plugin only needs to provide an FS image with a different configuration as input to `rfsck`. No modification to the source code of `rfsck` is required. As will be shown in §5.2, this simple strategy can help trigger vulnerabilities effectively.

Plugin #4: Dependency-aware FS Fuzzer. Fuzzing techniques have also been applied to improve the reliability of file systems [11, 29]. Nevertheless, fuzzing file systems is still challenging due to the lengthy state exploration time needed to exercise a practical file system under each configuration (e.g., it may take multiple weeks to trigger one bug [29]). In other words, the time penalty for exploring a less interesting configuration state is high. To mitigate the challenge, we integrate one open-source fuzzer `gt-hydra` [2] with ConfD through the `ConfD-gt-hydra` plugin. Similar to plugin #3, `ConfD-gt-hydra` leverages dependency-guided configurations generated by ConfD to create FS images with more complicated dependencies and thus more chances of vulnerability for fuzzing. The plugin only changes the configurations of the input images for `gt-hydra`; no modification to the source code of the base tool is needed.

Plugin #5 & #6: Dependency-aware Regression Test Suites. Besides research prototypes, there are standard regression test suites developed for file systems (e.g., `xfstests` [95] and `e2fsprogs/tests` [51]), which include carefully designed workloads and test oracles to ensure the quality of the target. Nevertheless, existing test suites only use a subset of configuration parameters and they are mostly dependency-agnostic. To address the limitation, we create two plugins: `ConfD-xfstests` and `ConfD-e2fsprogs`, for `xfstests` and `e2fsprogs/tests` respectively. The plugins scan the test scripts and automatically replace the built-in FS configurations of the test cases with the configuration states generated

[2] To avoid confusion, we use `gt-hydra` to refer to the Hydra fuzzing framework created by GaTech researchers [29], and use FB-HYDRA to refer to the Hydra configuration management framework created by Facebook [31].

Target FS	Self Dependency (SD)		Cross-Parameter Dep. (CPD)		Cross-Component Dep. (CCD)		All Level Combined	
Ecosystem	Extracted	FP	Extracted	FP	Extracted	FP	Extracted	FP
Ext4	17	0	48	1 (2.1%)	46	3 (6.5%)	111	4 (3.6%)
XFS	18	2 (11.1%)	10	3 (30.0%)	15	4 (26.7%)	43	9 (20.9%)
Total	35	2 (5.7%)	58	4 (6.9%)	61	7 (11.5%)	154	13 (8.4%)

Table 6: **Multilevel Configuration Dependencies Extracted by ConfD.** This table shows the numbers of multilevel dependencies extracted from Ext4 and XFS ecosystems automatically. 'FP' means False Positive rate.

Target FS	# of Uncorrectable Images Reported			
Ecosystem	`rfsck` (1)	`ConfD-rfsck` (25)		
Ext4	11	< 11 (4)	= 11 (4)	> 11 (17)

Table 7: **Comparison of Two FS Fault Injectors.** `rfsck` explores 1 default configuration state and reports 11 uncorrectable images. `ConfD-rfsck` explores 25 configuration states; it reports > 11 uncorrectable images (i.e., better than `rfsck`) in 17 out of 25 configuration states.

ID	Symptom of	Triggered?	
	Uncorrectable Corruption	`rfsck`	`ConfD-rfsck`
1	Unable to mount the FS	N	Y (6)
2	Invalid file data	N	Y (24)
3	Truncated file data	Y (11)	Y (250)
	Total	11	280

Table 8: **Comparison of Corruption Symptoms Triggered.** `ConfD-rfsck` triggered ('Y') more types of corruptions. The counts are in parentheses.

by ConfD. The two plugins use the 'Follow Dependency' policy of ConfD to drive the test cases deeply into the target functionalities without early termination due to superficial configuration errors. In doing so, we reuse the well designed test logic and enhance the test suites with dependency awareness. If any test case fails with the *valid* configurations provided by ConfD, the result is saved for postmortem analysis.

Note that ConfD plugins are not limited to the six above. By modularizing the core module of ConfD (Figure 3), we expect that other software may benefit from ConfD conveniently via plugins (see §6 for more discussion).

5 Experimental Results

In this section, we describe the experimental results of applying ConfD to analyze Ext4 and XFS. First (§5.1), we show that ConfD can extract 154 multilevel configuration dependencies from the target systems effectively with a low false positive rate (8.4%). Second (§5.2), we demonstrate that ConfD can help address configuration-related issues more effectively compared to existing dependency-agnostic solutions. Through the experiments, we have identified various configuration-related issues including 17 specification issues, 18 configuration handling issues, and 10 regression test failures induced by valid configurations.

5.1 Can ConfD extract multilevel dependencies?

Table 6 summarizes the multilevel configuration dependencies extracted by ConfD from Ext4 and XFS automatically. As shown in the table, we were able to extract 154 unique dependencies in total, including 35 Self Dependency (SD), 58 Cross-Parameter Dependency (CPD), and 61 Cross-Component Dependency (CCD). The multilevel dependencies have been observed on both Ext4 and XFS, which is consistent with our manual study (§3).

We manually examined all the 154 dependencies extracted by ConfD automatically and found that the overall false positive rate is 8.4% (13/154), which is similar to that of the previous work on analyzing configuration constraints in other software systems [13, 33]. Note that ConfD is designed to handle the unique configuration methods of FS ecosystems (§2 and §3.2) which is arguably more challenging to analyze compared to the targets of existing work.

5.2 Can ConfD help address configuration issues?

5.2.1 Dependency-agnostic vs. Dependency-guided

In this section, we compare the effectiveness of two open-source research prototypes (i.e., `rfsck` [25] and `gt-hydra` [29]) with and without ConfD support. We focus on the two research prototypes and the corresponding plugins for comparison because they provide quantitative metrics to measure the effectiveness straightforwardly. We defer the results of other plugins to the next section.

In the first experiment, we applied fault injectors `rfsck` and `ConfD-rfsck` to analyze Ext4 and its checker utility `e2fsck`. The fault injectors interrupt the checker operation and examine if the interrupted checker could lead to uncorrectable corruptions on the file system (i.e., cannot be fixed by another run of checker). They report the number of repaired FS images containing uncorrectable corruptions (i.e., "uncorrectable image"). Each uncorrectable image implies a vulnerability in the FS ecosystem that could lead to data loss [25].

The result of the experiment is summarized in Table 7. `rfsck` reports 11 uncorrectable images with the default configuration. `ConfD-rfsck` can explore different configuration states and we analyze the reports generated under 25 configuration states for comparison. In 4 out of the 25 states,

Target FS	# of Issues Reported (in two weeks)	
	`gt-hydra`	`ConfD-gt-hydra`
Ext4	1	17

Table 9: **Comparison of Two FS Fuzzers.** `ConfD-gt-hydra` reports more hangs given the same fuzzing time.

`ConfD-rfsck` generates less than 11 uncorrectable images; in 4 states, `ConfD-rfsck` generates the same amount of uncorrectable images (i.e., '= 11'); in the majority states (17), `ConfD-rfsck` generates more uncorrectable images (i.e., '> 11'), which suggests it is more effective in exposing potential vulnerabilities in the FS ecosystem.

Table 8 further compares the symptoms of uncorrectable corruptions triggered by `rfsck` and `ConfD-rfsck`. Overall, `ConfD-rfsck` triggers three different types of symptoms, while `rfsck` only triggers one symptom in our experiment. Since different symptoms typically imply different vulnerabilities in metadata protection and/or recovery in the FS ecosystem, the result also suggests that the dependency-guided configuration states used by `ConfD-rfsck` can help improve the effectiveness of `rfsck`.

In the second experiment, we applied `gt-hydra` and `ConfD-gt-hydra` to fuzz the Ext4 file system. The fuzzers systematically generate various inputs (i.e., FS metadata corruptions and system calls) to explore different code paths in the file system for triggering latent bugs [29]. We run each fuzzer continuously for two weeks. The fuzzers report the number of reliability issues detected on the target file system within the running period. The issues may include different types depending on the bug checkers used. We use the default SymC3 checker which can detect crash inconsistency bugs. Meanwhile, since the fuzzers are based on the AFL fuzzer [38], they also report crash and hang issues (detected by AFL) by default. Note that the only difference `ConfD-gt-hydra` introduces is the dependency-guided configurations, i.e., it does not change the test logic or criteria for reporting issues. Therefore, both the types of issues (e.g., 'crash', 'hang', 'crash inconsistency') and the number of issues reported can be used as the metric to evaluate effectiveness.

The result of the fuzzing experiment is summarized in Table 9. To make the comparison fair, we limit the two fuzzers to the same total execution time (i.e., two weeks each). We set the `ConfD-gt-hydra` to switch to a new dependency-guided configuration state every 12 hours, which leads to 28 critical configuration states being explored within two weeks. While each configuration in `ConfD-gt-hydra` is explored with only 1/28 of the time used by `gt-hydra` for its configuration, the overall result of `ConfD-gt-hydra` is better: `gt-hydra` only detects 1 issue on Ext4 by the end of the two week period, while `ConfD-gt-hydra` detects 17 issues in total. Interestingly, all issues reported in the experiment are 'hang'. This is expected because triggering more complicated semantic bugs may require multiple weeks.

ConfD Plugin (Type of Issue Reported)	# of Issue Reported		
	Ext4	XFS	Total
`ConfD-specCk` (undoc./wrong dep.)	13	4	17
`ConfD-handlingCk` (bad reaction)	13	5	18
`ConfD-xfstests` (test case failure)	5	4	9
`ConfD-e2fsprogs` (test case failure)	1	N/A	1
`ConfD-rfsck` (uncorrectable image)	280	–	280
`ConfD-gt-hydra` (hang)	17	–	17

Table 10: **Summary of Issues.** This table summarizes configuration-related issues observed via ConfD plugins.

Target FS Ecosystem	# of Undocumented/Wrong Dep.			Total
	SD	CPD	CCD	
Ext4	7	4	2	13
XFS	2	2	0	4
Total	9	6	2	17

Table 11: **Specification Issues.** This table summarizes the undocumented or wrong dependencies observed. 'SD', 'CPD', and 'CCD' are defined in Table 4.

In summary, the two sets of comparison experiments above show that ConfD can amplify the effectiveness of existing FS tools for identifying vulnerabilities quickly, which is particularly valuable for time-consuming methodologies like fault injection or fuzzing. Note that in all experiments, we do not randomly generate combinations of configurations. This is because a naive algorithm without any knowledge of inherent dependencies can easily lead to time-wasting configurations, as will be demonstrated further in §5.2.3.

5.2.2 Summary of Configuration Issues

Table 10 summarizes the configuration-related issues triggered by ConfD plugins in our experiments. Overall, we observed more than 300 issues of various types. The issues are diverse because the plugins are created for different purposes or based on different base tools (Table 5). Note that all the issues require dependency-guided configuration states generated by ConfD to manifest. In other words, continuously running the original research prototypes or standard test suites cannot expose the issues. Also, since we do not change the test logic of the base tools, the enhancement is purely contributed by the dependency information from ConfD. Since `ConfD-rfsck` and `ConfD-gt-hydra` have been discussed in §5.2.1, we focus on others below.

Table 11 summarizes the specification issues detected by `ConfD-specCk`. We have identified 17 inaccurate specification issues in total. The issues mainly manifest as undocumented critical dependencies or wrong dependencies, which may occur to both Ext4 and XFS and involve SD, CPD, and CCD. For example, there is a CPD extracted by ConfD which specifies that two parameters of `mke2fs` (i.e., `meta_bg` and `resize_inode`) cannot be used together, but this CPD is missing from the Linux man-pages. As another example, there

ID	Reaction	Description	Observed?
1	Early Termination	the utility program exits w/o pinpointing the configuration error	Y
2	Functional Failure	the utility fails functional testing w/o pinpointing the configuration error	Y
3	Silent Violation	the system changes input configurations to different values w/o notifying users	Y
4	Silent Ignorance	the system ignores input configurations	N
5	Crash/Hang	the system crashes or hangs	N
6	Partial Report	the utility partially identify the violated configuration dependencies	Y

Table 12: **Suboptimal Reaction of Configuration Dependency Violation.** This table summarizes the bad handling behaviors observed when the configuration dependencies are violated. The first five are based on the definitions from [13].

is a CCD which implies that `resize2fs` may not be used for Ext4 when the `bigalloc` feature is enabled through `mke2fs`. Violating the CCD may corrupt the file system, which is unfortunately not mentioned in the specification.

Table 12 summarizes the suboptimal handling of misconfigurations identified through `ConfD-handlingCk`. We follow the criteria in the literature [13]: when a misconfiguration occurs (i.e., a dependency is violated), the system should pinpoint either the offending parameter's name/value or its location information; failing to do so implies misconfiguration vulnerabilities. Specifically, there are six types of misconfiguration vulnerabilities based on different reactions, including 'Early Termination', 'Functional Failure', 'Silent Violation', 'Silent Ignorance', 'Crash/Hang', and 'Partial Report'. The first five types are based on the definitions from [13], while the last one is unique in our study because we consider more complicated multilevel dependencies.

As an example, the `mke2fs` parameter `-E encoding` enables the `casefold` feature and set the encoding in Ext4. But if the user tries to disable the `casefold` feature when using the `-E encoding`, instead of showing an error or warning, the utility enables the `casefold` feature silently without informing the user. We consider this as 'Silent Violation'.

When more than one dependency is violated, utilities often only show a partial message (i.e., 'Partial Report'). For example, the `mkfs.xfs` parameter `sunit` involves two dependencies: (1) it does not allow unit suffixes, and (2) it cannot be specified together with `su`. But when both dependencies are violated, the utility may only show one of the violations.

In total, we have observed 4 out of the 6 types of suboptimal reactions, which suggests that FS ecosystems are not immune from misconfiguration vulnerabilities reported in other practical systems. Note that `ConfD-handlingCk` leverages the static analysis of ConfD to violate specific dependencies carefully, which avoids many duplicate and valid configuration states for testing. This reduces the manual effort needed for the post-mortem analysis.

In terms of `ConfD-xfstests` and `ConfD-e2fsprogs`, we have observed 10 new test case failures which can be induced by *valid* configuration states generated by ConfD. For example, `ConfD-xfstests` triggers an Ext4 corruption when applying the online defragmentation tool `e4defrag` to the file system with the `bigalloc` feature enabled. Note that a FS test case may involve multiple utilities. Due to the complexity of the test case and the FS ecosystems, a test case may fail for various subtle reasons (e.g., timing at `mount`) in practice, which is time-consuming to diagnose even for developers [47]. In our experiments, we observed more than 10 newly failed test cases after changing with valid configurations. We only count the cases that we have manually verified and reproduced at the time of this writing. Also, since ConfD limits the change to the configuration states without modifying the test logic, it may help narrow down the root cause of a test case failure to the configuration-related code paths.

Framework	# of States	# of Duplicate	# of Invalid
FB-HYDRA	56,592	42,745 (75.5%)	15,146 (26.8%)
ConfD	30	0	0

Table 13: **Comparison of State Generation.**

5.2.3 State Generation: FB-HYDRA vs. ConfD

One unique feature of ConfD is it generates configuration states based on multilevel dependencies, which is critical for analyzing configuration issues given the huge configuration space. To the best of our knowledge, the FB-HYDRA configuration management framework [31] provides the most similar functionality. It includes a "multirun" feature to support running an application with different configurations in different runs automatically. We compare the configuration states generated by FB-HYDRA and ConfD in this section to demonstrate the difference.

Table 13 shows the states generated by FB-HYDRA and ConfD for the same program (i.e., `mke2fs`) given the same set of configuration parameters. For simplicity, we only use 10 parameters with limited ranges in this experiment. As shown in the table, even with this simplified scenario, FB-HYDRA may generate many duplicated or invalid states. This is because FB-HYDRA is agnostic to the configuration constraints of `mke2fs`. Specifically, FB-HYDRA maintains a list for each parameter and its possible values. It passes all lists to the `itertools.product()` function which returns the cartesian product of the values in the lists. Such a simple algorithm is incompatible with FS ecosystems. For example, '`mke2fs -b 1024 -C 2048`' and '`mke2fs -C 2048 -b 1024`' are equivalent in practice but are considered as different in FB-HYDRA. Moreover, invalid states can easily be created by FB-HYDRA due to violation of dependencies, which suggests the importance of dependency analysis.

Note that FB-HYDRA has other features that CONFD does not have (e.g., Python library support). Also, FB-HYDRA supports plugins which makes it possible to benefit from the state generation of CONFD (see §6 for more discussion). Therefore, we view FB-HYDRA and CONFD as complementary.

6 Limitations & Future Work

No study or tool is perfect, and our work is no exception. We discuss the limitations of our work as well as a few promising extensions in this section.

Limitations of the multilevel taxonomy. As briefly mentioned in §3.1, the multilevel configuration dependencies should be interpreted with the study methodology in mind, because they are derived from an incomplete set of configuration-related issues from two FS ecosystems. It is likely that there are more complex dependencies in FS ecosystems, which deserves further investigation.

Limitations of the CONFD framework. The current prototype requires a few user inputs (§4.1.4) to guide the automated dependency analysis, which we hope to reduce through more sophisticated state analysis. Also, CONFD can only handle a subset of LLVM IR for taint analysis and it only considers two parameters at a time for CPD and CCD, which may lead to incomplete dependency or false positives. We hope to improve these through more advanced software engineering efforts in the future, which will likely improve the effectiveness further. Similarly, there are limitations in plugins. For example, `ConfD-handlingCk` only induces at most two violations for one configuration state for simplicity; there may be more issues if we consider more than two. `ConfD-xfstests` only transforms a subset of the test suite due to the irregular configuration handling. Despite the limitations, CONFD has been effective in analyzing dependencies and exposing configuration-related issues in our experiments, so we believe that it will be valuable to the community.

Integration with other file systems and tools. As mentioned in Table 1, many file systems can be configured through different utilities, which could potentially benefit from the multilevel dependency analysis of CONFD after minor customization (e.g., providing FS-specific inputs in JSON format 4.1.4). Also, CONFD is complementary to other modern tools besides the base tools used in current plugins. For example, FB-HYDRA [31] uses YAML files to store configurations which is compatible with the JSON files used by CONFD. Moreover, it supports a set of plugins called "Sweepers" to manipulate the selection of parameters. The dependency-based state generation in CONFD could be implemented as one special "Sweeper" for FB-HYDRA [31]. Similarly, the configurations generated by CONFD could potentially be integrated into CI/CD frameworks [62] to enable pipelined configuration-oriented testing and deployment. We leave the integration with other file systems (e.g., ZFS) and tools as future work.

Support for other software. Configuration dependency is not limited to file systems. For example, NDCTL [74] is a utility to configure the `libnvdimm` subsystem in Linux. We expect that adding NDCTL to the dependency analysis will likely help address NVM-specific configuration issues more effectively. Also, researchers have observed functionality or correctness dependencies between local file systems and other software (e.g., databases [16], distributed storage systems [20, 23, 34, 37]), many of which are also related to configurations. The dependencies studied in this work may serve as a foundation for investigating such configuration-related issues beyond file systems. Also, since LLVM supports compiling a wide set of languages (e.g., C++, Rust, Swift) to IR through various frontends [85], the core analysis of CONFD is expected to be applicable to software written in other languages as well.

Better configuration design. An alternate perspective of the configuration challenge studied in this work is that we may have too many parameters today. One might argue that it is perhaps better to reduce the parameters to avoid vulnerabilities or confusions, instead of adding new configurations for more features. Also, one might suggest that (in theory) we can implement every utility functionality in the file system itself to avoid tricky cross-component configuration dependencies. Essentially, these are trade-offs of the file system and configuration design that deserve more investigation from the community. We hope that by studying real-world configuration issues and releasing the CONFD prototype, our work can help identify problematic configuration parameters and further help with the reduction of such parameters to improve the configuration design in general.

7 Conclusion

We have presented a study on 78 real-world configuration issues and built an extensible framework called CONFD for addressing various configuration issues. Our experiments on Ext4 and XFS demonstrate that CONFD can help address configuration issues effectively by leveraging configuration dependencies. In the future, we would like to improve CONFD further and investigate other systems as discussed in §6. We hope that CONFD can facilitate follow-up research on addressing the increasing challenge of configurations in general.

Acknowledgments

We thank Ethan Miller, our shepherd, and the anonymous reviewers for their tremendous feedback. We also thank Runzhou Han and Wei Xu for their help on validating and reproducing a few bug cases. This work was supported in part by National Science Foundation (NSF) under grants CNS-1855565, CCF-1853714, CCF-1910747 and CNS-1943204. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] Sanjay Ghemawat et al. “The Google file system”. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. 2003.

[2] Vijayan Prabhakaran et al. “IRON File Systems”. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*. 2005.

[3] Haryadi S. Gunawi et al. “SQCK: A Declarative File System Checker”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2008.

[4] Lorenzo Keller et al. “ConfErr: A tool for assessing resilience to human configuration errors”. In: *Proceedings of the 38th IEEE International Conference on Dependable Systems and Networks (DSN)*. 2008.

[5] Shan Lu et al. “Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics”. In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2008.

[6] Huning Dai et al. “CONFU: Configuration Fuzzing Testing Framework for Software Vulnerability Detection”. In: *Int. J. Secur. Softw. Eng.)* 1.3 (2010).

[7] Ariel Rabkin et al. “Static extraction of program configuration options”. In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. 2011.

[8] Zuoning Yin et al. “An Empirical Study on Configuration Errors in Commercial and Open Source Systems”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*. 2011.

[9] Edmund Clarke et al. “Model Checking and the State Explosion Problem”. In: *Tools for Practical Software Verification*. Jan. 2012.

[10] Daniel Fryer et al. “Recon: Verifying File System Consistency at Runtime”. In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*. 2012.

[11] Christoph Albrecht et al. “Janus: Optimal Flash Provisioning for Cloud Storage Workloads”. In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (ATC)*. 2013.

[12] Lanyue Lu et al. “A Study of Linux File System Evolution”. In: *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. 2013.

[13] Tianyin Xu et al. “Do Not Blame Users for Misconfigurations”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*. 2013.

[14] Dongpu Jin et al. “Configurations Everywhere: Implications for Testing and Debugging in Practice”. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 2014.

[15] Thanumalayan Sankaranarayana Pillai et al. “All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2014.

[16] Mai Zheng et al. “Torturing Databases for Fun and Profit”. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2014.

[17] Changwoo Min et al. “Cross-Checking Semantic Correctness: The Case of Finding File System Bugs”. In: *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. 2015.

[18] James Bornholt et al. “Specifying and checking file system crash-consistency models”. In: *SIGPLAN Not.* 51.4 (2016).

[19] Scott Klemmer Tianyin Xu Vineet Pandey. “An HCI View of Configuration Problems”. In: *arXiv*. 2016.

[20] Aishwarya Ganesan et al. “Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions”. In: *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST)*. 2017.

[21] Aravind Machiry et al. “DR. Checker: A Soundy Analysis for Linux Kernel Drivers”. In: *Proceedings of the 26th USENIX Conference on Security Symposium (SEC)*. 2017.

[22] Salman Niazi et al. “HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases”. In: *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*. 2017.

[23] Jinrui Cao et al. “PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems”. In: *Proceedings of the 2018 International Conference on Supercomputing (ICS)*. 2018.

[24] Mikaela Cashman et al. “Navigating the Maze: The Impact of Configurability in Bioinformatics Software”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 2018.

[25] Om Rameshwar Gatla et al. “Towards Robust File System Checkers”. In: *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. 2018.

[26] Om Rameshwar Gatla et al. “Towards Robust File System Checkers”. In: *ACM Transactions on Storage (TOS)* 14.4 (2018).

[27] Kuei Sun et al. “Spiffy: Enabling File-System Aware Storage Applications”. In: *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. 2018.

[28] Shehbaz Jaffer et al. “Evaluating File System Reliability on Solid State Drives”. In: *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 2019.

[29] Seulbae Kim et al. “Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 2019.

[30] Wen Xu et al. “Fuzzing file systems via two-dimensional input space exploration”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019.

[31] Omry Yadan. *Hydra - A framework for elegantly configuring complex applications*. Github. 2019. URL: https://github.com/facebookresearch/hydra.

[32] Zhen Cao et al. “Carver: Finding Important Parameters for Storage System Tuning”. In: *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. 2020.

[33] Qingrong Chen et al. “Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2020.

[34] Runzhou Han et al. “Fingerprinting the Checker Policies of Parallel File Systems”. In: *IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*. 2020.

[35] Xudong Sun et al. “Testing Configuration Changes in Context to Prevent Production Failures”. In: *PProceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2020.

[36] Duo Zhang et al. “A Study of Persistent Memory Bugs in the Linux Kernel”. In: *Proceedings of the 14th ACM International Conference on Systems and Storage (SYSTOR)*. 2021.

[37] Runzhou Han et al. “A Study of Failure Recovery and Logging of High-Performance Parallel File Systems”. In: *ACM Transactions on Storage (TOS)* 18.2 (2022).

[38] *American Fuzzy Lop*. https://lcamtuf.coredump.cx/afl/.

[39] *Apache Common Configuraitons*. https://commons.apache.org/proper/commons-configuration/userguide/upgradeto2_0.html.

[40] *Apache Hadoop*. https://hadoop.apache.org/.

[41] *btrfs-balance*. https://man7.org/linux/man-pages/man8/btrfs-balance.8.html.

[42] *btrfs-check*. https://man7.org/linux/man-pages/man8/btrfs-check.8.html.

[43] *btrfs-scrub*. https://man7.org/linux/man-pages/man8/btrfs-scrub.8.html.

[44] *CFEngine*. https://github.com/cfengine/core.

[45] *chkdsk*. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/chkdsk.

[46] *defrag*. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/defrag.

[47] *Discussion between Ext4 developers and newbie on finding bugs on Ext4*. https://lore.kernel.org/linux-ext4/Yx9fUHiiZaKXeLUw@mit.edu/.

[48] *disk utility*. https://www.dssw.co.uk/reference/diskutil.html.

[49] *dump*. https://www.freebsd.org/cgi/man.cgi?query=dump&apropos=0&sektion=8&manpath=FreeBSD+13.1-RELEASE+and+Ports&arch=default&format=html.

[50] *e2fsck*. https://linux.die.net/man/8/e2fsck.

[51] *e2fsprogs-test*. https://sourceforge.net/projects/e2fsprogs/files/e2fsprogs-TEST/.

[52] *E2fsprogs: Ext2/3/4 Filesystem Utilities*. https://e2fsprogs.sourceforge.net/.

[53] *e4defrag*. https://man7.org/linux/man-pages/man8/e4defrag.8.html.

[54] *Ext4*. https://ext4.wiki.kernel.org/index.php/Main_Page.

[55] *format*. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/format.

[56] *fsck*. https://man.minix3.org/cgi-bin/man.cgi?query=fsck.

[57] *fsck_apfs*. https://www.manpagez.com/man/8/fsck_apfs/.

[58] *fsck_ufs*. https://www.freebsd.org/cgi/man.cgi?query=fsck_ufs.

[59] *growfs*. https://www.freebsd.org/cgi/man.cgi?growfs(8).

[60] *HotHardware: Windows 10 20H2 Update Reportedly Damages SSD File Systems If You Run ChkDsk*. https://hothardware.com/news/windows-10-20h2-update-damages-ssd-file-systems-chkdsk.

[61] *JavaScript Object Notation*. https://www.json.org/json-en.html.

[62] *Jenkins*. https://www.jenkins.io/.

[63] *Lustre*. https://www.lustre.org/.

[64] *Maintaining Linux man-pages*. https://www.kernel.org/doc/man-pages/maintaining.html.

[65] Wilcox Matthew. *DAX: Page cache bypass for filesystems on memory storage*. https://lwn.net/Articles/618064/.

[66] *mke2fs*. https://linux.die.net/man/8/mke2fs.

[67] *mkfs.btrfs*. https://man7.org/linux/man-pages/man8/mkfs.btrfs.8.html.

[68] *mkfs.xfs*. https://man7.org/linux/man-pages/man8/mkfs.xfs.8.html.

[69] *mount*. https://man7.org/linux/man-pages/man8/mount.8.html.

[70] *mount*. https://www.freebsd.org/cgi/man.cgi?query=mount.

[71] *mount_apfs*. https://www.manpagez.com/man/8/mount_apfs/.

[72] *mountvol*. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/mountvol.

[73] *MySQL NDB Cluster*. https://en.wikipedia.org/wiki/NDB_Cluster.

[74] *NDCTL*. https://github.com/pmem/ndctl.

[75] *newfs*. https://www.freebsd.org/cgi/man.cgi?newfs(8).

[76] *NTFS*. https://www.ntfs.com/index.html.

[77] *OpenStack*. https://www.openstack.org/.

[78] *OpenStack Swift*. https://docs.openstack.org/swift/latest/.

[79] *resize2fs*. https://linux.die.net/man/8/resize2fs.

[80] *restore*. https://www.freebsd.org/cgi/man.cgi?query=restore.

[81] *shrink*. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/shrink.

[82] *Soot - A framework for analyzing and transforming Java and Android*. http://soot-oss.github.io/soot/.

[83] *The First and Only Adaptive Computational Storage Platform*. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html.

[84] *The Linux man-pages Project*. https://www.kernel.org/doc/man-pages/maintaining.html.

[85] *The LLVM Compiler Infrastructure*. https://llvm.org/.

[86] *Trim*. https://en.wikipedia.org/wiki/Trim_(computing).

[87] *Windows 10 2004/20H2: Microsoft fixes chkdsk issue in update KB4592438*. https://borncity.com/win/2020/12/21/windows-10-2004-20h2-microsoft-fixes-chkdsk-issue-in-update-kb4592438/.

[88] *Windows 10 20H2: ChkDsk damages file system on SSDs with Update KB4592438 installed*. https://borncity.com/win/2020/12/18/windows-10-20h2-chkdsk-damages-file-system-on-ssds-with-update-kb4592438-installed/.

[89] *XFS*. https://xfs.wiki.kernel.org/.

[90] *xfs_admin*. https://man7.org/linux/man-pages/man8/xfs_admin.8.html.

[91] *xfs_fsr*. https://man7.org/linux/man-pages/man8/xfs_fsr.8.html.

[92] *xfs_growfs*. https://man7.org/linux/man-pages/man8/xfs_growfs.8.html.

[93] *xfs_repair*. https://man7.org/linux/man-pages/man8/xfs_repair.8.html.

[94] *xfsprogs*. https://www.linuxfromscratch.org/blfs/view/svn/postlfs/xfsprogs.html.

[95] *xfstests*. https://github.com/kdave/xfstests.

[96] *zfs-create*. https://www.freebsd.org/cgi/man.cgi?query=zfs-create.

[97] *zfs-destroy*. https://www.freebsd.org/cgi/man.cgi?query=zfs-destroy.

[98] *zfs-mount*. https://www.freebsd.org/cgi/man.cgi?query=zfs-mount.

[99] *zfs-rollback*. https://www.freebsd.org/cgi/man.cgi?query=zfs-rollback.

[100] *zfs-set*. https://www.freebsd.org/cgi/man.cgi?query=zfs-set.

HadaFS: A File System Bridging the Local and Shared Burst Buffer for Exascale Supercomputers

Xiaobin He[1], Bin Yang[2 1*], Jie Gao[1], Wei Xiao[1], Qi Chen[2], Shupeng Shi[1],
Dexun Chen[1], Weiguo Liu[4], Wei Xue[2 3 1], Zuo-ning Chen[5†]
[1]National Supercomputing Center in Wuxi, [2]Tsinghua University, Dept. of C.S,
[3]Tsinghua University, BNRist.,[4]Shandong University, [5]Chinese Academy of Engineering

Abstract

Current supercomputers introduce SSDs to form a Burst Buffer (BB) layer to meet the HPC application's growing I/O requirements. BBs can be divided into two types by deployment location. One is the local BB, which is known for its scalability and performance. The other is the shared BB, which has the advantage of data sharing and deployment costs. How to unify the advantages of the local BB and the shared BB is a key issue in the HPC community.

We propose a novel BB file system named HadaFS that provides the advantages of local BB deployments to shared BB deployments. First, HadaFS offers a new Localized Triage Architecture (LTA) to solve the problem of ultra-scale expansion and data sharing. Then, HadaFS proposes a full-path indexing approach with three metadata synchronization strategies to solve the problem of complex metadata management of traditional file systems and mismatch with the application I/O behaviors. Moreover, HadaFS integrates a data management tool named Hadash, which supports efficient data query in the BB and accelerates data migration between the BB and traditional HPC storage. HadaFS has been deployed on the Sunway New-generation Supercomputer (SNS), serving hundreds of applications and supporting a maximum of 600,000-client scaling.

1 Introduction

High Performance Computing (HPC) is experiencing an era of explosive growth in computing scale and data. In order to meet the growing I/O demands of HPC applications, researchers propose Burst Buffer (BB) [35] to build a data acceleration layer through new storage media such as SSDs to serve applications' I/O quickly. Since 2016, more and more supercomputers have introduced Burst Buffer, such as Frontier [44], Fugaku [18], LUMI [15], Summit [43], Tianhe-2 [63], etc. Depending on the deployment location of SSDs, BBs can be classified into two types [10]: 1) local BB, which means SSDs are deployed on each computing node as local disks;2) shared BB, which means SSDs are deployed on dedicated nodes that can be accessed by computing nodes, such as I/O forwarding nodes [5] to support shared data access.

Since each BB node in the local BB is dedicated to serving one computing node, the local BB can achieve good scalability, and its performance can grow linearly with the number of computing nodes. However, it still has some limitations: 1) The local BB is not suitable for scenarios such as N-1 I/O mode (all processes share one file) and workflow due to the difficulty of data sharing. 2) The local BB architecture results in significant resource waste due to the large variance in I/O load between HPC applications and the relatively low percentage of data-intensive applications [67]. 3) The deployment cost of the local BB will rise sharply in the future as supercomputers scale up rapidly.

In contrast, the shared BB has the advantage of data sharing and deployment costs compared to the local BB. But it is challenging to support ultra-scale supercomputers with hundreds of thousands of clients [71], and existing work has many limitations. For example, Qian et al. [48] proposed LPCC, a caching technique that integrates SSDs in the Lustre [8] clients to improve read/write performance. However, LPCC is inefficient for data sharing and metadata-intensive access because data stored on the Lustre clients' SSDs must be flushed to the Lustre server before being shared. Herold et al. [25] proposed BeeOND, which functions similarly to LPCC but inherits the scalability and cache sharing limitations of BeeGFS.

Currently, we have entered the era of exascale supercomputers, which leads to a sharp increase in concurrent I/O. At the same time, the I/O requirements of HPC applications vary widely. How to unify the advantages of local BB and shared BB to meet the various application requirements and reduce the cost of building BB is an urgent problem to solve. Moreover, both the local BB and the shared BB have the advantage of high performance compared with the traditional global

[*] First Author and Second Author contribute equally to this work.

[†] Zuo-ning Chen is the corresponding author, email: chenzuoning@vip.163.com.

file system (e.g., Lustre, and we will use "GFS" to represent "global file system" in this paper.) but have the disadvantage of small capacity. So, BBs must work in conjunction with the GFS to meet capacity requirements. But the existing BBs either run in a static data migration mode [16, 43, 48] or require applications to migrate data through computing nodes [24, 51, 52], which has low migration efficiency and leads to a waste of computing resources. Large-scale BB data management and migration is also a problem that needs to be solved.

To solve these problems, we propose a novel BB file system, HadaFS, building on the shared BB deployment, which combines the scalability and performance advantages of local BB with the data sharing and deployment costs advantages of shared BB. HadaFS proposes a new architecture named Localized Triage Architecture (LTA) to solve the problem of insufficient scalability of the shared BB. LTA constructs all HadaFS servers as a shared storage pool, flexibly controlling the concurrency scale between clients and servers to ensure convenient data sharing. Additionally, HadaFS proposes a runtime user-level interface to ensure that I/O requests can be processed on the nearest server, helping clients use the BB in a manner that approximates the local BB. To solve the performance problems caused by the strong POSIX consistency, HadaFS proposes a full-path indexing approach, using the K-V approach instead of the traditional directory tree, supporting three-category metadata management policies. What's more, HadaFS integrates a data management tool to help users manage data in the BB, and migrate data between the BB and the GFS quickly and efficiently.

HadaFS has been deployed on the Sunway New-generation Supercomputer (SNS) [36], serving hundreds of applications, supporting a maximum of 600,000-client scaling, with an I/O aggregation bandwidth of 3.1 TB/s. The main contributions of this paper include:

- This paper describes a novel BB file system named HadaFS and performs a comprehensive experimental study on the SNS to evaluate its effectiveness.
- This paper proposes the LTA architecture, which enables the application-oriented data layout, achieves scalability on par with node-local BBs, and reduces interference caused by a large number of connections on a single server.
- HadaFS proposes three metadata synchronization strategies to address the mismatch between traditional file systems' complex metadata management and HPC applications' various consistent semantics requirements.
- This paper proposes a localized data management method that enables all BB nodes to execute data management commands in parallel via a pipeline, enabling efficient data query and fast data migration between the BBs and the GFS.

2 Motivation and Background

2.1 Motivation

As I/O requirements for HPC applications continue to grow, BBs have been introduced to many cutting-edge supercomputers. However, the existing major types of BB technologies still have many limitations.

2.1.1 The contradiction between BBs' scalibility and application behaviors

With the barrier to exascale computing being broken, the I/O concurrency of cutting-edge supercomputers can reach hundreds of thousands, which stresses the scalability of BBs. At the same time, the increase in the proportion of data sharing applications such as AI and workflow has led to changes in I/O requirements, and high-speed sharing of large-scale data has become much more important [45].

Building a more flexible BB architecture to meet the new changes in supercomputer systems and application requirements has become a challenge for the design of exascale supercomputers. Currently, some cutting-edge supercomputers use different solutions. For example, Frontier is an exascale supercomputer and uses independent hardware to build the local BB and the shared BB, respectively [44]. But this method requires many acceleration devices (SSDs) and high construction and maintenance costs. Fugaku deploys the shared BB and uses software to provide storage services similar to the local BB and the shared BB with different name spaces [21]. But their implementation is static and is challenging to control performance contention during large-scale I/O concurrency. Summit deploys the local BB and supports data sharing through the software [43]. But this method requires data sharing through GFS storage, which is inefficient.

In summary, the above methods have obvious advantages and disadvantages. Since shared BB can also be deployed on computing or data forwarding nodes [21, 66], from the perspective of cost control, we believe the shared BB deployment is more suitable for future ultra-large-scale computing node systems. To this end, in this paper we investigate how, starting from a shared BB model, we can attain the benefits of the local BB model to better address the requirements of HPC applications at exascale and beyond.

2.1.2 Complex metadata management mismatches application behaviors

Traditional file systems are designed for generality, so their file management is implemented in the directory tree structure and strictly follows the POSIX protocol. However, in HPC, computing nodes are generally responsible for reading and writing data and rarely perform directory tree access [32]. So relaxation of POSIX has become a common choice for many file systems [6, 12, 43, 59] to improve the performance.

However, due to the wide variety of HPC applications, how to relax POSIX remains a huge challenge.

Table 1: Applications and their suitable consistency semantics

Consistency Semantics	Applications
Strong consistency	–
Commit consistency	FLASH-HDF5 [60]
Session consistency	NWChem [58], QMCPACK [29], VASP [55] LBANN [20], Chombo [4], VPIC-IO [64]
Eventual consistency	ENZO [9], pF3D-IO [31], HACC-IO [38]

Wang et al. [60] studied the requirements of some typical HPC applications and classified the consistency semantics of HPC file systems into strong consistency semantics, commit consistency semantics, session consistency semantics, and eventual consistency semantics, as shown in Table 1. The higher the degree of consistency a system supports, the more adaptable it is, but at the cost of higher overhead. And different HPC applications have different requirements for consistency. Therefore, it is a big challenge to choose consistency semantics flexibly to balance the application's requirements and exploit the BB performance.

2.1.3 Inefficiencies in data management

A recent study found that although most applications on Summit and Cori can use the BB to speed up I/O performance, the BB utilization is low, and it is necessary to develop flexible data management tools for users [7]. Besides, the BB is not a place for applications to persistent store data in most cases. On the one hand, the BB capacity is smaller than the GFS capacity, e.g., Summit's BB capacity is 7.4PB while its GFS capacity is 250PB [26], Fugaku's BB capacity is 16PB while its GFS capacity is 100PB [21]. On the other hand, some typical HPC applications require hundreds of terabytes of data to input or output, e.g., NICAM-LETKF [69] has nearly 300,000 files and over 400 terabytes, Tokmark [65] has 32,768 files and nearly 100 terabytes. So, the BB system needs to consider efficiently and conveniently migrating data between the BB and the GFS.

Data migration between the BB and the GFS can be divided into two types: transparent and non-transparent. In transparent data migration, software automatically migrates the BB data to the GFS in blocks or files [16, 43, 48], which may cause a large amount of unnecessary data migration. In non-transparent migration, data migration often needs computing nodes to participate, leading to the computing resource being idle during the data migration process [24, 51, 52] and wasting resources. Both of the above types support loading data from the GFS to the BB asynchronously statically in advance, which can satisfy the data readahead requirements. However, neither of the above two types could support users to dynamically manage the BB data migration during the application running, which is very unfavorable for efficient utilization of the BB.

2.2 Background

The SNS is built on Sunway's new-generation heterogeneous high-performance many-core processors and interconnection network chips and adopts a similar architecture to Sunway TaihuLight [13]. The supercomputer consists of a computing system, interconnection network system, software system, storage system, maintenance and diagnosis system, power supply system, and cooling system. Figure 1 shows the overall architecture.

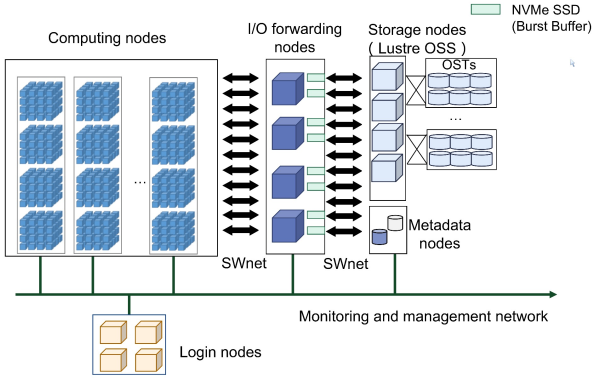


Figure 1: Architecture of the SNS

Each computing node contains one Sunway new-generation many-core processor *SW26010P*, which adopts a heterogeneous architecture similar to SW26010 and has 6 CGs (Core Groups [36]) with 390 computing cores. These components are interconnected through a ring network. The whole system is composed of more than 100,000 SW26010P processors, which are interconnected by a fat-tree network called *SWnet*.

The computing nodes are connected to the I/O forwarding nodes through the interconnection network, and the I/O forwarding nodes provide I/O request forwarding or storage. When providing I/O request forwarding, the SNS adopts a similar software architecture to TaihuLight (LWFS+Lustre [13]) and connects to the storage nodes through a separate storage network. When providing storage services, the SNS adopts a new software architecture and deploys a burst buffer file system, HadaFS, which is proposed in this paper. The I/O forwarding nodes serve as the HadaFS servers and use NVMe SSDs to handle users I/O requests.

3 Design and Implementation

3.1 Overview of HadaFS

Figure 2 shows the overall architecture of HadaFS, including the HadaFS client, HadaFS server, and data management tool. HadaFS serves as a shared burst buffer file system and can provide a global view for each client. The HadaFS client runs on the computing nodes and serves as a static/dynamic library that intercepts and redirects the POSIX I/O requests from applications to the HadaFS server, which means the lifecycle of the HadaFS client is entirely dependent on applications. Note that HadaFS does not support the move, rename, or link operation as recent studies have demonstrated that these functions are rarely or not used at all during parallel application running [32]. The HadaFS server runs on the dedicated burst buffer nodes where NVMe SSDs are deployed, providing global data and metadata storage services. Each file in HadaFS is associated with two types of servers. One type is the data storage server that stores the HadaFS file's data through the basic file system on NVMe SSDs, and the other type is the metadata storage server that stores the HadaFS file's metadata through a high-performance database (RocksDB [17]). The data management tool, named *Hadash*, runs on the user login nodes and is used to manage the data migration between the global file system and HadaFS. More details can be seen in Section 3.7.

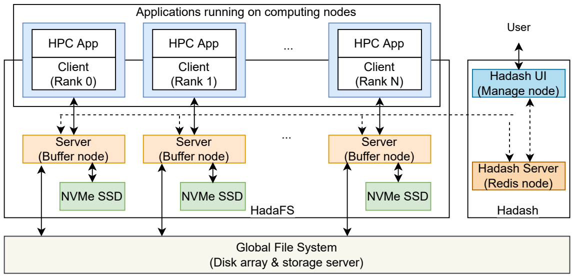


Figure 2: Architecture of HadaFS

3.2 Localized Triage Architecture

The traditional kernel file system handles the application's I/O requests by mounting the file system through the operating system, requiring to implement the full POSIX semantics and introducing the kernel's overhead of I/O requests stage-in and stage-out. An alternative method is to mount the file system through the application and bypass the kernel. Although this method can help clients avoid many rules that the kernel imposes on a file system and reduce the overhead of I/O requests stage-in and stage-out of the kernel, too many links are not conducive to large-scale expansion and may lead to service instability [71]. For example, for a computing node with 24 CPU cores, a file system client running in the kernel mode can be accessed by all processes running on the node only after mounting once. In contrast, each application process has to mount a client for a file system running in the user space. Obviously, both methods have certain limitations. HadaFS combines both advantages in a new approach named Localized Triage Architecture (LTA).

HadaFS follows the idea of bypassing the kernel and uses it by directly mounting the client into the application. In order to control the number of clients served by a single server at the same time and avoid the performance bottleneck caused by too many clients connected to a single server, HadaFS adopts the method of connecting only one server per client. For a HadaFS client, we call the HadaFS server connected to it the *bridge server*. The bridge server is responsible for handling all I/O requests generated by the client and writes data to the underlying file based on the offset and size of the I/O request initiated by the client. Each file corresponds to an independent file in the underlying file system(ext4) on the bridge server. When the client needs to access data on another server, it must be forwarded through the bridge server. Therefore, servers are a fully connected structure. Note that if the storage space of one bridge server is filled up, all the clients connected to it will automatically switch to another HadaFS server.

Considering that the number of clients (computing node processes) in an ultra-scale supercomputer will be much larger than the number of servers (storage nodes), it is a better way to perform the necessary I/O forwarding through the full connection of the server. In order to ensure that most I/O requests are processed on the bridge server and reduce the forwarding ratio of requests, HadaFS proposes an interface ($mount(mount\_point, Seq)$) to allow applications to control the selection of bridge servers when this is advantageous. *mount_point* stands for the mount point and is a prefix for a file path in HadaFS. *Seq* can be set flexibly according to changes in the application data sharing mode, network topology differences, and other factors to adapt to the application's data parallelism and system architecture parallelism. Currently, HadaFS supports three types of settings:

- *Seq* is set to the *MPI_RANK* of the application, which will connect the server for the client in a round-robin manner. This setting is suitable when the application is submitted to different computing nodes multiple times to ensure that each application process can connect precisely to the original bridge server, thus reducing the data forwarding during data access.

- *Seq* is set to the computing node ID, which can be used to match the topology between the specific computing node and the BB node, and helps ensure that the computing node can store data in the nearest BB node in the network.

- *Seq* is set according to the application's actual data distribution and sharing requirements, which means each client can specify any server it wants to connect to. So applications can improve the efficiency of data access by flexibly controlling the mapping between the clients and the servers.

LTA not only provides each computing node with a bridge server that runs as the local BB but also supports the sharing of all clients through the full interconnection between all bridge servers, combining the advantages of the local BB and the shared BB. Moreover, *mount_point* and *Seq* can be controlled by the environment variables, so HadaFS can support transparent mounting for users by loading the HadaFS library to read environment variables in advance before the application starts. However, to fully exploit the high performance of HadaFS, especially for read performance, we advise applications to change their code to specify the client-to-server mapping, which can help reduce the data forwarding. After HadaFS mounted, applications can perform I/O with the interface, which is exactly the same as POSIX file operations.

3.3 Namespace and metadata handling

In order to improve the scalability and performance, HadaFS abandons the idea of directory trees and employs a full-path indexing approach like CHFS [57] and Vesta [14]. For a file in HadaFS, its data is stored on the bridge server of the HadaFS client that generated the file, and its metadata storage location is determined by the path hash. Files' metadata are stored by key-value, and the file path is a globally unique ID (key). The HadaFS client performs compliance checking on the absolute path of files based on its specific prefix mount point instead of checking layer by layer in the form of a directory tree. When multiple files need to be accessed, the load can be distributed to various servers, thus significantly improving metadata performance [57].

The metadata of HadaFS is compatible with the metadata items under the stat structure in Linux, including *name*, *ino*, *owner*, *mode*, *timestamp*, etc. HadaFS divides its metadata information into four categories:

- The first category is maintained during the file creation, including *name*, *owner*, *mode*, etc.

- The second category is maintained during the file access, including *file size*, *modification time*, *access time*, etc.

- The third category is information that HadaFS does not need to maintain, such as *ino*, *stdev*, etc. Since HadaFS adopts the file path as the globally unique ID, this information has no meaning in HadaFS.

- The fourth category is an ordered list of the location information of the file segments. Each item in the list is sorted by offset, consisting of server name, fragment offset, size, writing time, and other information.

Two kinds of metadata databases are maintained on each HadaFS server, and their data structures are shown in Figure 3. One is the local metadata database (*LMDB*), which stores the first and fourth category metadata information of the file locally, and the file's local identification (LID) is the local path corresponding to the file. The other is the global metadata database (*GMDB*), which stores the first two and the fourth categories of metadata information. The metadata of a HadaFS file is stored in a unique GMDB on a HadaFS server located by hashing the full path of the file.

Both metadata databases are built based on the RocksDB [50], which is also used to maintain metadata by many other famous file systems, such as GekkoFS [59], MadFS [28], etc. Although RocksDB does not support multi-threaded shared writing, it doesn't constitute a bottleneck, and this is demonstrated by both production-run and test scenarios. The keys of the two metadata databases are composed of the user's UID, GID, and $PATH$. The GID and UID are used to control the range of string retrieval because HadaFS uses string prefix matching to retrieve files. For the N-N I/O mode, each client writes the independent file, and the metadata stored in the LMDB matches the category one and four metadata stored in the GMDB. For the N-1 I/O mode, multiple clients share the same file and may use different HadaFS bridge servers. At this time, GMDB is responsible for merging file metadata from multiple LMDBs.

During the file reading and writing, LMDB records the change of its metadata, maintains an ordered list of local data segment locations, and sends the data to the GMDB to which the HadaFS file belongs. GMDB is responsible for maintaining a global list of data segment locations for files to support the global sharing of data between HadaFS servers. More details can be seen in Section 3.6.2.

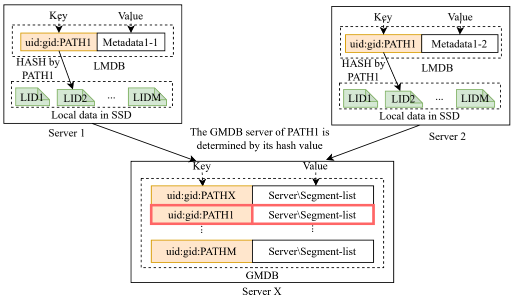


Figure 3: Two K-V tables on the HadaFS server

3.4 HadaFS I/O control and data flow

Here, we discuss the control and data flow details of HadaFS. Figure 4 shows an example with three HadaFS clients and three HadaFS servers:

- Client A performs an I/O request to create a file $F1$ and write 100-MB data. The data will be written directly to client A's bridge server, X.

- Server X writes the metadata information and location information of $F1$ to the LMDB.

- Based on the file path of $F1$, the metadata of $F1$ is calculated to be stored on server Y. Then, server X writes the metadata information and location information of $F1$ to the GMDB on server Y.

- Client C performs an I/O request to read a file $F1$.

- Client C's bridge server, Z, receives the I/O request and gets the metadata and location information of F1 from server Y based on the path of $F1$.

- Server Z reads data from server X and forwards it to client C.

Note that ensuring local writes and global readability of data streams is advantageous, especially for scenarios where the application needs to output checkpoints frequently. Besides, read-intensive applications can also achieve high performance through the mount interface, which can control the mapping relationship between the client and the bridge server to reduce the probability of forwarding as much as possible and improve read performance. In summary, approaches proposed by HadaFS not only help constrain the number of clients undertaken by each server and reduce performance jitter but also lay the foundation for the storage system to support the application's parallelism fully.

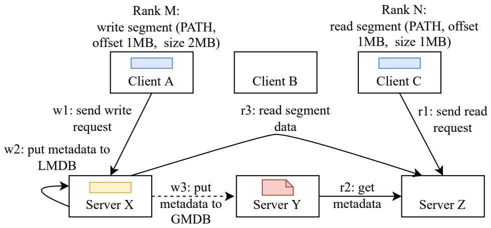


Figure 4: An example of HadaFS I/O control and data flow

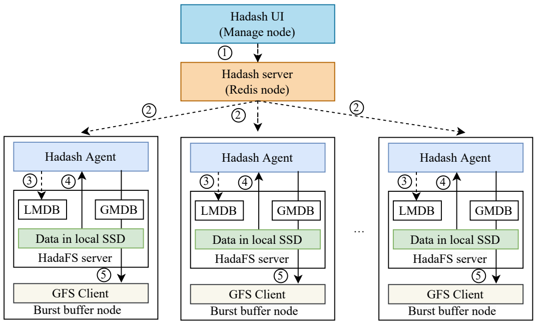


Figure 5: The stage-out flow of Hadash

3.5 Data management tool

We seek to overcome some disadvantages of existing BB approaches, such as LPCC [48] and Datawarp [24]. LPCC may result in the migration of large amounts of temporary data. Datawarp requires the application to specify the migration between the BB and GFS in their source code or job submission scripts, which is usually a static migration approach and requires computing nodes to participate in the migration. HadaFS provides a data management tool named *Hadash* to support users in retrieving and managing files in the directory tree view, which is divided into two categories according to functions: metadata information query and data migration.

The metadata information query mainly provides commands such as *ls*, *du*, *find*, *grep*, etc. Among them, *ls* and *find* support file information query with directory tree view. Hadash obtains information from the metadata database for these query-type operations and presents them in commands commonly used in the Linux shell. The other commands involve data migration, such as *rm*, *get*, *put*, etc. Hadash sends the commands to the data management modules on the HadaFS servers through a specific Redis [49] pipeline. Then, the data management module on each HadaFS server uses LMDB for local data location and executes these commands in parallel.

Figure 5 shows an example of the data migration flow from HadaFS to the GFS. Firstly, the user sends a data management command to the Hadash server via Hadash UI. Secondly, the Hadash server receives and forwards the command to all Hadash agents on the BB nodes. Thirdly, the Hadash agents parse the command, obtain the list of files specified by the command from LMDB, and then read these files from the local SSDs. Finally, the Hadash agents write these files to the GFS. When all files have finished writing, the Hadash agents will return success via another Redis pipeline, and then Hadash will tell the user that the stage-out has been completed. If the file to be migrated is continuously appended, Hadash will also continuously copy the newly written data. This behavior is the same as that of Linux's default data copying tool, *cp*. It is worth mentioning that Hadash uses a prefix-matching approach to present a virtual directory tree in data management, and the prefix-matching approach could be executed locally through the LMDB, thus reducing the impact on the GMDB.

Hadash uses a distributed management method to support data localization management. During the data management process, there is no need to know all the data views on the BB nodes, so its performance can grow linearly with the number of BB nodes (the main bottleneck is the GFS client).

3.6 Optimizations on HadaFS

3.6.1 Consistency semantics and metadata optimization

HadaFS adopts the idea of relaxed consistency semantics as other file systems [59]. HadaFS does not support cache data on the client and the server. It relies on the cache mechanism of the basic file system (ext4) to increase performance, and its consistency semantics mainly depend on metadata synchronization. Thus, HadaFS proposes three metadata synchronization strategies for different application scenarios to avoid the over-designing of traditional file systems.

The first strategy (called *mode1*) is to update all metadata asynchronously (corresponding to eventual consistency semantics). All operations are executed locally on the bridge servers first during file opening, deletion, reading, and writing, and metadata will be updated asynchronously from the LMDB to the GMDB later, which is the highest performance metadata update mode. This strategy is equivalent to providing node-local storage for computing nodes and is suitable for scenarios without data dependencies.

The second strategy (called *mode2*) is to update part of the metadata synchronously and part of the metadata asynchronously (corresponding to session consistency semantics and commit consistency semantics). The first metadata category mentioned above (such as *name*, *owner*, etc.) will be updated synchronously when the file is created. The second metadata category (such as $file\ size$, $modify\ time$, etc.) will be updated asynchronously during file reading and writing or be updated synchronously by the flush operation. The second strategy is the default way to use and beneficial to improving file reading and writing performance.

The third strategy (called *mode3*) is to synchronize the metadata in all open, read, and write operations during the file access processes (slightly weaker than strong consistency semantics since HadaFS does not support overlapping writes). All servers need to get the file location first to ensure the synchronization of the first metadata category, and all operations such as open, write, read and flush need to synchronize the second metadata category.

HadaFS has special data location rules and does not use a distributed lock mechanism, so it isn't easy to ensure data consistency by HadaFS itself. If the application uses the N-N I/O mode (also known as File Per Process), then there is no data conflict because there is no file sharing. However, for the N-1 write scenario, since HadaFS requires that the data be written to the bridge server locally, it cannot support the overlap write in the N-1 Mode. In addition, atomic write is only supported under the third metadata synchronization strategy. To ensure data consistency, users must at least understand the file-sharing mode of the application, which could be obtained through Darshan [11], Beacon [67], etc.

3.6.2 Optimization on the shared file

The LTA architecture is suitable for N-N I/O mode, which can realize multi-file parallelism and help fully utilize the performance of the BBs. For N-1 I/O mode, HadaFS proposes a management method similar to ADIOS BP file layout [40] to improve the performance, where each client writes its own data in an independent file (corresponding to BP's process group), and the GMDB maintains the file's metadata(corresponding to BP's group index). In this way, a shared file can be stored on multiple servers, and the reading and writing of the shared file can be converted into concurrent reading and writing. However, since the HadaFS server stores data in file format, the data layout of each process is completely unknown, so the amount of fragment information managed by the GMDB may be high in extreme scenarios, which affects system performance. For example, suppose there are 100,000 processes concurrently writing a shared file, and each process writes 6 times consecutively. In a completely random case (no fragment information to merge), it may produce 600,000 file fragments.

To this end, HadaFS uses a list sorted by offset to store segment location information and merges location information for adjacent segments of the same bridge server to improve the performance of the segment management. The average time complexity of segment insertion and retrieval during write and read is $O(logN)$, where N is the number of segments in the file. All three metadata synchronization strategies support N-1 mode, and the metadata of a file will only be stored in one GMDB. Each GMDB uses one thread to access the RocksDB, and the peak IOPS of a GMDB is the peak IOPS for accessing a single file when using the third metadata strategy.

3.6.3 Interference avoidance

As we all know, there are many jobs running simultaneously on supercomputers. These jobs tend to compete for shared resources, resulting in I/O interference. I/O interference is a serious problem known to modern supercomputer users [19, 30, 33]. Many studies have also proved that dynamically mapping the client to the server is also helpful in improving application performance [27, 72]. Since the shared BBs support data sharing for many applications, the clients belonging to different jobs may share the same server, resulting in resource competition and performance degradation. Thanks to the flexible design of HadaFS, users can dynamically formulate the connection relationship from the HadaFS clients to the HadaFS servers, which can effectively help isolate the BB resources for different applications to solve the I/O interference between jobs. Section 4.4.3 demonstrates the effectiveness of HadaFS in avoiding interference with five real-world applications.

Moreover, we have also noticed that the flexible design of HadaFS has also led to high requirements for users who want to fully utilize HadaFS, as mentioned in Section 3.6.1 and Section 3.2. In order to reduce the burden on users, the HadaFS team is developing an automatic server assignment tool based on the monitoring tool [67] and adaptive I/O optimization framework [68]. This tool can automatically assign underlying BB resources, set the mount environment variables, and select the metadata synchronization strategy for applications, helping users isolate the underlying BB resources and improve the performance of their applications.

3.7 HadaFS on the SNS

HadaFS has been deployed on the SNS for over a year and supports hundreds of applications, including the 2021 Gordon Bell Award finalist application (Tokamak Plasma Simulation) [65] that scales to 480,000 processes (32,768 I/O processes) via HadaFS with an I/O aggregation bandwidth of 700 GB/s. Figure 6 shows the deployment of HadaFS. There are two HadaFS server on every I/O forwarding node, and each HadaFS server uses an NVMe SSD to support the storage of the HadaFS file's data (with ext4) and metadata (with LMDB and GMDB).

As we all know, the overhead of achieving fault tolerance can be significant. Therefore, HPC storage systems often transfer high availability to the application layer for implementation to pursue higher performance. Most HPC applications generally use periodic write checkpoints [6] to reduce the cost of restoring applications after failures occur. So, HadaFS is positioned as a temporary high-performance BB system, similar to the BB system on Frontier [44], Summit [43], etc. The Sunway new generation supercomputer doesn't adopt erasure code or data redundancy to handle node failures. If a BB node fails, HadaFS can be available after the BB node recovery as long as the SSD is not damaged. Moreover, in order to reduce the cost of recovering data in case of failure, HadaFS supports applications to periodically back up key data to the GFS. There have been 15 BB node failures but no SSD corruptions for over a year of deployment.

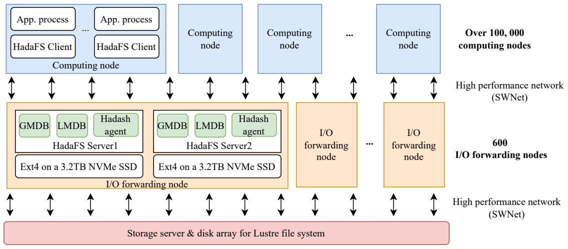


Figure 6: The deployment of HadaFS

4 Evaluation

We carry out the evaluation on the SNS to test the performance of HadaFS. The SNS contains more than 100,000 computing nodes, and each node can start up to 6 MPI processes and 6 HadaFS clients. That means the whole machine can support more than 600,000 MPI processes and 600,000 HadaFS clients. It has a total of 600 I/O forwarding nodes, and each I/O forwarding node is configured with two 3.2TB NVMe SSDs (Each NVMe SSD corresponds to a HadaFS server). All nodes are interconnected using the SWnet network, and HadaFS uses the SWnet-based RDMA protocol to transfer data. We compare the performance of HadaFS with BeeGFS (a popular parallel file system that many supercomputers have used to manage the BB [3,25,41]) and GFS (the traditional parallel file system used by the SNS based on LWFS [67] and Lustre [8]). To ensure the evaluation fairness, we slightly modify BeeGFS (version: 7.2.5) to make it better suited to the SNS, including using the SWnet to provide the high-speed RDMA communication and increasing the number of processes for managing services to achieve high-speed mounting performance. And, BeeGFS is configured with the same number of storage servers and metadata servers as HadaFS. For GFS, it uses a similar forwarding architecture as Sunway TaihuLight [13], consisting of 132 OSSs and 4 MDSs.

4.1 Metadata performance evalutaion

In order to improve the metadata performance, HadaFS proposes three metadata management strategies. Here, we use *mode1*, *mode2*, and *mode3* to represent these three strategies as mentioned in Section 3.6.1.

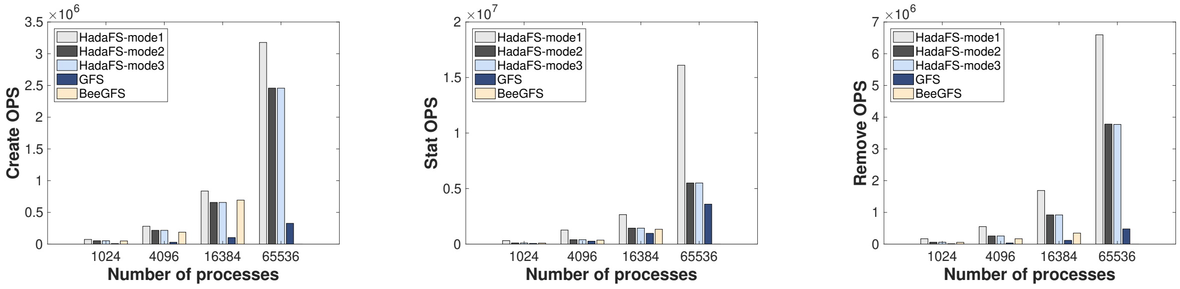
We first use MDTest [2] (A benchmark for metadata performance evaluation) to compare the metadata performance differences of HadaFS, GFS, and BeeGFS with parallel scales of 1024, 4096, 16384, and 65536 processes. 4, 16, 64, and 256 I/O forwarding nodes are used for HadaFS and BeeGFS, each running a data server and a metadata server on a common SSD. Note that the number of the GFS metadata servers in this experiment is 4 due to the limited metadata servers of the Lustre file system.

Figure 7(a), 7(b), and 7(c) show the OPS comparison of *Create*, *Stat*, and *Remove*, respectively. Mode1 has the highest performance. Mode2 has comparable performance to mode3 because there is no read/write operation in the MDTest setting. BeeGFS metadata performance is similar to HadaFS's mode2 and mode3 for 1024 processes. When the number of test processes increases, both HadaFS and BeeGFS obtain the higher performance, but the performance of BeeGFS is slightly slower than HadaFS. Besides, BeeGFS can not scale up to 65,536 processes. The main reason is that BeeGFS needs to mount 16384 clients to support 65,536 processes on the SNS, but it cannot mount successfully at such a large scale due to the limitation of centralized management service (It isn't easy to successfully mount clients in batches after exceeding 10,000 nodes). Unsurprisingly, the traditional file system GFS has the lowest performance due to the performance overhead caused by data forwarding software LWFS and the limited metadata servers of Lustre.

4.2 Data performance evaluation

Here, we use IOR [53] (A benchmark for data performance evaluation) to compare the I/O bandwidth differences between HadaFS, GFS, and BeeGFS with parallel scales of 1024, 4096, 16384, and 65536 processes. The request size is set to 8 KB for random read/write and 1 MB for sequential read/write. 4, 16, 64, and 256 I/O forwarding nodes are used for HadaFS and BeeGFS, each running a data server and a metadata server on a common SSD. Specifically, for GFS, the data server is the Lustre OSS, and the metadata server is the Lustre MDS. All 132 OSSs (located on the storage nodes) and 4 MDS are used in the experiment.

Figure 8 shows the results. For HadaFS, mode1 has the highest performance, followed by mode2, and finally mode3. HadaFS does not show a significant performance advantage at smaller scales, but as the scale reaches 65,536 processes, HadaFS performs much better than other file systems. For read operations, HadaFS can approach the theoretical performance limit of SSDs. For write operations, random writes are not conducive to the performance of HadaFS due to the inability to utilize the kernel caching mechanism. For BeeGFS, it can perform close to mode1 and mode2 sometimes but still cannot scale to 65,536 processes. Expectantly, GFS has the lowest performance again due to the forwarding overhead (see



(a) create OPS

(b) stat OPS

(c) remove OPS

Figure 7: Metadata performance comparison

Section 4.1) and the storage medium (OSTs are constructed by HDDs).

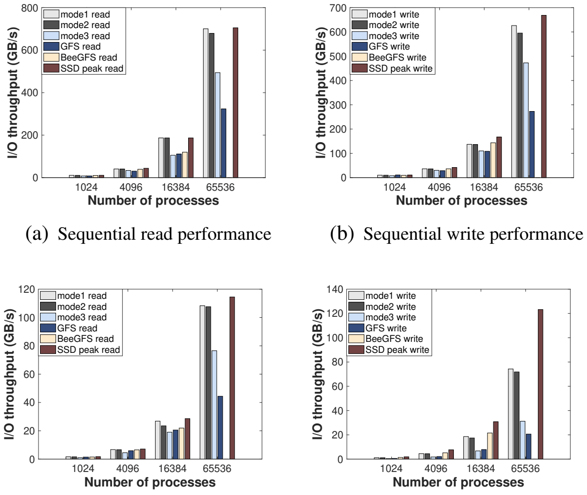
In addition, we also scale HadaFS to 600,000 clients and 1200 servers, and Figure 9 shows the experiment results. Since mode2 is a default metadata management strategy of HadaFS, all tests are done based on mode2, and the request size is set to 1 MB. Comparing the theoretical performance of 1200 NVMe SSDs (3.4 TB/s and 3.1 TB/s for read and write, respectively), the bandwidth utilization of the SSDs is close to 90% under ultra-large-scale concurrent data access.

4.3 Data migration evaluation

In order to manage data migration between the BB system and the GFS quickly and efficiently, HadaFS provides a data management tool named Hadash. Here, we evaluate Hadash in terms of its I/O throughput and its ability to migrate small files compared with Datawarp [24] simulated by Slurm-LUA [52] (Datawarp and LUA are the only two BB plugins supported by Slurm). And like BeeGFS, we construct a new LUA script based on a BB-LUA-example provided by Slurm to fit the experimental environment to ensure fairness. HadaFS is configured with 256 data servers and 256 metadata servers, and Datawrap is configured with 4096 processes for data migration.

First, we use 4096 files for the data stage-in and stage-out experiments, and the total data volume of these files ranges from 256 MB to 64 TB. Figure 11 shows the results of the experiment. When the total volume of the files to be migrated is relatively small (less than 64 GB for stage-in and less than 16 GB for stag-out), Hadash obtains a slightly worse performance than Datawarp. This is because when the total volume is small, the size of the individual files is also small, resulting in the command distribution and result acquisition mechanism based on the Redis pipeline occupying a larger proportion of the time. However, as the total volume and the individual file size get larger, the I/O throughput of Hadash stabilizes around 100 GB/s (for stage-in) and 140 GB/s (for stage-out), which is much higher than Datawarp.

Additionally, we found that the stage-out performance is significantly better than the stage-in performance. The write performance of the GFS and the read performance of the BB determines the stage-out performance, while the read performance of the GFS and the write performance of the BB determines the stage-in performance. In our test, the GFS (Lustre) client has a write cache, so the write performance of the GFS is higher than the read performance, and the read performance of the BB is also higher than the write performance, which leads to the higher stage-out performance.



(a) Sequential read performance

(b) Sequential write performance

(c) Random read performance

(d) Random write performance

Figure 8: I/O throughput comparison

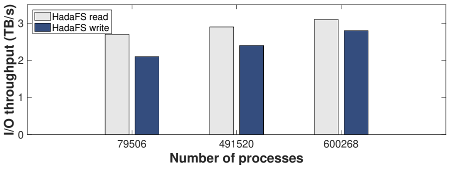


Figure 9: Ultra-scale performance

We also evaluated Hadash's ability to handle large amounts of small files. Figure 12 shows the result of the experiment using different numbers of 4-KB small files for the data stage-in and stage-out. For stage-in, Hadash outperforms Datawarp significantly when the number of small files exceeds 10,000 while Datawarp's performance varies less. For stage-out, Hadash outperforms Datawarp significantly when the number of small files exceeds 100,000.

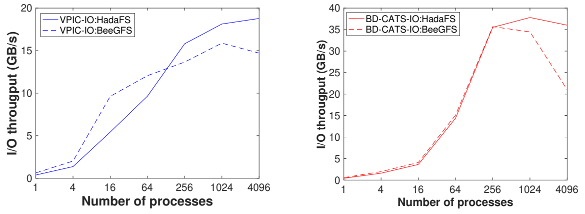
Again, the performance of stage-out is better. One of the reasons is as stated above, and another reason is as follows. In the stage-in flow, Hadash needs to read all files in a single directory from the GFS, and this process takes longer as the number of files in a single directory increases. In contrast, Hadash does not need to read any files in the directory from the GFS in the stage-out flow and only needs to create files.

4.4 Evaluation with real-world applications

4.4.1 Performance evaluation on the shared files

For the shared file access pattern, HadaFS adopts the idea of BP files similar to ADIOS [39] and further improves the shared file access performance by merging adjacent segments through ordered lists. This subsection compares the performance differences between HadaFS and BeeGFS on shared file access using several applications. Both HadaFS and BeeGFS are configured with 16 servers, each running on a common SSD. Figure 10 shows the results.

First, we use VPIC-IO [64] (provides scalable writing HDF5 data by VPIC) to evaluate the performance of HadaFS when writing shared files. Applications' parallelism scales from 1 to 4096, and each process writes about 1.1-GB data to a shared file containing 8 variables. Figure 10(a) shows the results. BeeGFS performs better than HadaFS when the application's parallelism is less than 64. This is because BeeGFS clients can use the kernel's cache, and the striping technique used by BeeGFS can ensure a low probability of conflict at small scales. However, as the parallelism gets larger, HadaFS outperforms BeeGFS significantly due to its good scalability.



(a) Write performance

(b) Read performance

Figure 10: Performance evaluation on the shared file

Then, we use BD-CATS-IO [46] (provides scalable reading HDF5 data by the VPIC) to evaluate the performance of HadaFS when reading shared files. Applications' parallelism also scales from 1 to 4096, and each process reads about 1.1-GB data from a shared HDF5 file. Figure 10(b) shows the results. BeeGFS and HadaFS have almost the same performance when the parallelism of applications is less than 256. Similar to the write performance, when the application scale gets larger, the performance of HadaFS will be significantly better than that of BeeGFS, as HadaFS uses the LTA architecture to better isolate I/O conflicts between different clients.

4.4.2 Performance evaluation on the mount policy

Compared to the traditional fully connected mount approach, HadaFScannot guarantee that the data demanded by the client is always on its bridged server, so we first evaluate the performance impact of I/O forwarding on HadaFS. We distribute files evenly and regularly on the server according to the RANK number in advance and then accurately control the forwarding of generated data between servers through the mount interface.

We use one process to evaluate the latency variation of different block sizes due to the I/O forwarding, and Figure 13(a) shows the results. The solid line in the figure represents the client's direct access latency to its bridge server, while the dashed line represents the I/O forwarding latency. I/O forwarding does cause an increase in latency. The larger the block size, the smaller the proportional increase in latency. For 8-KB and 1-MB block sizes, the latency increases by 34.4% and 17.7%, respectively.

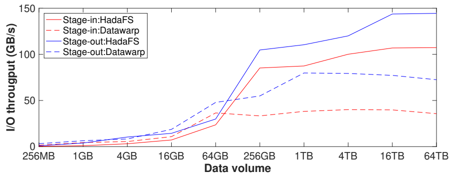


Figure 11: Data migration throughput comparison

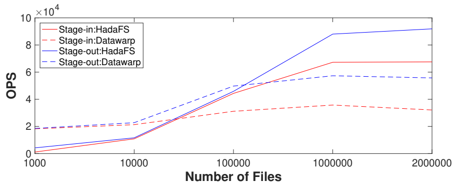
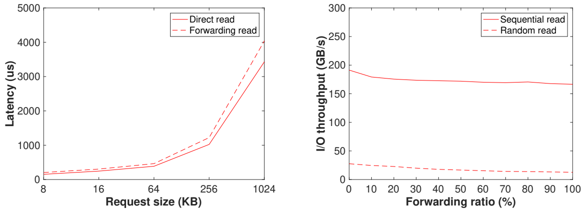


Figure 12: Number of files migrated per second comparison

We then evaluate the impact of the data forwarding ratio on the bandwidth of HadaFS. The higher the data forwarding ratio, the more data needs to be forwarded through the bridge server. In the experiment, HadaFS is configured with 16,384 clients and 64 servers, and the I/O request size is set to 1 MB for sequential read and 8 KB for random read. Figure 13(b) shows the results. As the I/O forwarding ratio increases, the throughput of HadaFS decreases, with a maximum loss of 18% for sequential read and 54% for random read. This is because the smaller the block size, the larger the forwarding overhead, and the larger the throughput loss.

However, note that HadaFS provides a runtime mount interface to control the mapping relationships flexibly, which can significantly reduce I/O forwarding. Let's take NEMO (a state-of-the-art modeling framework for research activities and forecasting services in the ocean and climate sciences) [70] and its post-processing as an example to illustrate the advantages of the runtime mount interface. NEMO uses N-N I/O mode (also known as File Per Process) to read and write NetCDF files and is configured with 65,536 processes. And its post-processing is configured with 512, 1024, and 2048 processes. It is worth mentioning that in real application scenarios, the parallelism of the post-processing is significantly smaller than the parallelism of the module application. All 250,000 files (the total volume is more than 5 TB) output by NEMO are stored in 16 HadaFS servers.

Figure 14 shows the results. In the default configuration, the individual post-processing processes often need to access data that is not on the bridge server. So, the forwarding rate is high(up to 93%), and the performance is poor. After re-mapping the client-to-server connections with the mount interface, the forwarding rate can be greatly reduced, and performance can be significantly improved, up to 30% or more. This demonstrates that the flexible mount interface of HadaFS can significantly improve the performance of applications that need to share data.



(a) Forwarding latency (b) Forwarding throughput

Figure 13: Forwarding evaluation of HadaFS

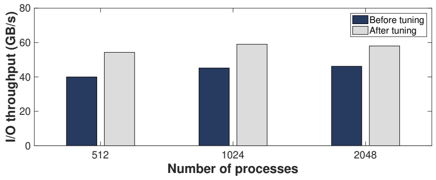


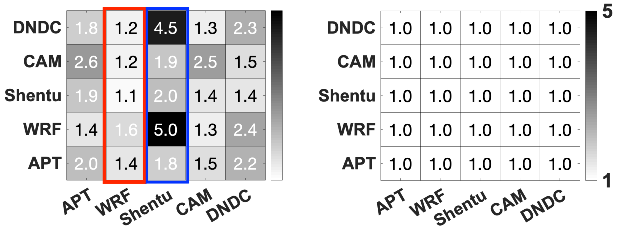
Figure 14: Performance improvement with mount interface

4.4.3 Performance evaluation on the interference

In this subsection, we evaluate HadaFS as a shared file system with 5 real-world applications, including APT (a particle dynamics simulation application) [62], WRF (a regional numerical weather prediction system) [1], Shentu (an extreme-scale graph engine) [34], CAM (a standalone global atmospheric model deriving from the CESM project for climate simulation/projection) [54], and DNDC (a biogeochemistry application for agroecosystems) [22].

First, we simulate the common I/O interference caused by sharing resources between jobs in HPC by co-running two applications on the same HadaFS server, and each application runs with 512 processes. Figure 15(a) shows the results. Each block's darkness reflects the application's slowdown factor at the row header by the application at the column header. As we can see, since different applications have different I/O behaviors, they share HadaFS with each other resulting in varying levels of performance slowdown. For example, WRF is a traditional serial I/O application that uses only one I/O process to access files through the NetCDF library, and its I/O load is very low (I/O bandwidth less than 200 MB/s). When WRF shares HadaFS server with other applications, it has less impact on them, as marked by the red box. On the contrary, Shentu is an I/O intensive application with N-N I/O mode, so its I/O bandwidth is very high (up to 2.5 GB/s). When Shentu shares HadaFS server with other applications, it has a high impact on them (up to 5x performance slowdown for other applications), as marked by the blue box.

HadaFS supports a runtime user-level mount interface and can assign the service resources according to the group name mentioned in Section 3.6.3. So in the production environment, HadaFS can flexibly change the mapping relationship from HadaFS clients to HadaFS servers through the mount interface to avoid I/O interference. Figure 15(b) shows the performance of avoiding sharing HadaFS server with other applications through the mount interface. This experiment demonstrates that the flexible mount approach provided by HadaFS can be beneficial for applications to avoid I/O interference.



(a) Before tuning (b) After tuning

Figure 15: Impact of HadaFS's interference avoidance on pairwise application co-run slowdown

4.4.4 Evaluation with large scale applications

This subsection shows the usage of HadaFS in five real-world large-scale applications, including NEMO [70], TK (Tokamak Plasma Simulation, 2021 Gorden Bell Prize finalist) [65], DiDA (an AI-enabled large-scale parallel atmospheric data-assimilation system) [23], Jstack (a debugging tool for the SNS) [47], and SWLBM (an efficient and scalable LBM) [37].

Figure 16 shows that applications can achieve significant I/O improvement (at least 7x) and reduce their runtimes and I/O ratios after using HadaFS, proving the effectiveness of HadaFS in improving the large-scale applications' performance. Details are as follows. TK runs with 32,768 I/O processes and 960 HadaFS servers, and the total runtime is more than 48 hours. With HadaFS, the I/O percentage of the total runtime dropped from 9.4% to 1.5%. NEMO runs with 480,000 I/O processes and 1200 HadaFS servers, and the total runtime of a model-year simulation is about 114 hours. With HadaFS, the I/O percentage of the total runtime dropped from

1.3% to 0.13%. DIDA runs with 65,536 I/O processes and 1200 HadaFS servers, and the total runtime is more than 2 hours. With HadaFS, the I/O percentage of the total runtime dropped from 4.1% to 0.5%. Jstack runs with 100,000 I/O processes and 256 HadaFS servers, and the total runtime is about 100 minutes every day. With HadaFS, the I/O percentage of the total runtime dropped from 5.0% to 0.46%. SunwayLBM runs with 18,000 I/O processes and 256 HadaFS servers, and the average runtime is about 7 days. With HadaFS, the I/O percentage of the total runtime dropped from 14.2% to 2.6%.

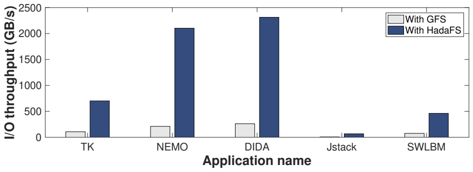


Figure 16: Performance improvement for large-scale real-world applications

5 Related work

Burst buffer on Top computers Many supercomputers deploy Burst Buffers to accelerate applications' I/O performance. Summit [26] adopts the local BBs technology and deploys SSDs on each computing node. To support data staging and migrate applications data to the GFS, Summit uses two technologies. One is Spectral, which provides the block-level data cache for applications. The other is SymphonyFS, which provides the file-level data cache for applications [43]. Fugaku [18] deploys the shared SSDs on the dedicated BB nodes (also used as the computing nodes but have two more cores), with each BB node serving a portion of the computing nodes, and provides users with three namespaces for different BB usage through LILO [21]. Applications can select the corresponding namespace according to the shared access requirements of the data (intra-node, intra-application, or inter-applications). As the world's first exascale supercomputer, Frontier [44] builds both the local BB and the shared BB. The local BB provides a burst data cache within the node, while the shared BB provides a shared data cache. In summary, the above situation shows that BB technology is moving towards integrating the local BB and the shared BB to support HPC applications' various requirements. For supercomputers with more than 100,000 nodes, local BB needs to deploy NVMe SSDs on each computing node, which will undoubtedly increase the cost. HadaFS combines the advantages of local BB and shared BB through the LTA architecture and the application-controllable mount interface, which can be deployed on ultra-scale supercomputers with more than 100000 nodes at a relatively low cost.

Researches for Burst Buffer Research on BBs has recently become a hot topic and can be divided into three technical routes. The first one is to improve the traditional distributed file systems and add new functions to support BBs, such as Lustre LPCC [48], which can cache data in client SSDs for transparent data caching. However, this mechanism inherits the scalability problems of traditional distributed file systems and is difficult to scale to ultra-scale. Similarly, there is BeeOND, which is based on BeeGFS [3]. The second one is to add data-sharing mechanisms based on the local BBs, such as Unifyfs [42], Burstfs [61], CHFS [57], Gfarm/BB [56], etc. These file systems run on the user layer and will be created when the job is submitted and destroyed when the job completes. In order to take advantage of BB's performance, these file systems also use a consistent relaxation protocol similar to HadaFS but does not consider data staging. The third one is to build a full-featured persistent BB storage system, e.g., DAOS [39]. DAOS [39] is an object storage system developed based on SPDK/PMDK. It is organized in an object-centric manner, supports transaction and multiple consistency management methods, and supports POSIX semantics based on object storage. Compared with the above works, HadaFS is built based on Shared BB, which has more advantages in scalability and has passed the verification of ultra-large-scale deployment of more than 100,000 nodes. In addition, HadaFS can provide applications with flexible and controllable POSIX consistency semantics.

6 Conclusion

We present a Burst Buffer file system named HadaFS, bridging the local BB and the shared BB based on the shared BB deployment. HadaFS can support ultra-scale deployments and balance the performance and the overhead with the novel architecture LTA and hierarchical metadata management mechanism. Besides, HadaFS integrates an internal data management tool named Hadash, which can provide a global data view and efficient data migration for users. HadaFS has been deployed on the SNS (over 100,000 computing nodes) and supports hundreds of applications. Especially, HadaFS supports several ultra-scale applications, providing stable and high-performance I/O services for these applications in preparation for the ACM Gordon Bell bid. Moreover, We demonstrate the high performance, high scalability, and low cost of HadaFS through a comprehensive experimental study.

Acknowledgement

We appreciate the thorough and constructive comments/suggestions from all reviewers. We thank our shepherd, Rob Ross, for his guidance during the revision process. This work is partially supported by the National Key R&D Program of China (Grant No. 2020YFB0204800), Marine S&T Fund of Shandong Province for Laoshan Laboratory (LSKJ202202100), National Natural Science Foundation of China (Grant No. U2242210), and the Major Key Project of PCL (No. PCL2022A05).

References

[1] A description of the advanced research WRF version 3. http://www2.mmm.ucar.edu/wrf/users/.

[2] Mdtest hpc benchmark, 2010. https://sourceforge.net/projects/mdtest/.

[3] David Abramson, Chao Jin, Justin Luong, and Jake Carroll. A beegfs-based caching file system for data-intensive parallel computing. In *Asian Conference on Supercomputing Frontiers*, pages 3–22. Springer, Cham, 2020.

[4] Mark F. Adams, Phillip Colella, Daniel T. Graves, Jeffrey N. Johnson, Hans Johansen, Noel Keen, Terry J. Ligocki, Daniel F. Martin, Peter McCorquodale, David Modiano, Peter O. Schwartz, T. D. Sternberg, and Brian van Straalen. Chombo software package for amr applications design document. 2014.

[5] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and Ponnuswamy Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[6] John Bent, Garth 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 Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12. IEEE, 2009.

[7] Jean Luca Bez, Ahmad Karimi, Arnab Paul, Bing Xie, Suren Byna, Philip Carns, Sarp Oral, Feiyi Wang, and Jesse Hanley. Access patterns and performance behaviors of multi-layer supercomputer i/o subsystems under production load. pages 43–55, 06 2022.

[8] Peter Braam. The lustre storage architecture. *arXiv preprint arXiv:1903.01955*, 2019.

[9] Corey Brummel-Smith, Greg L. Bryan, Iryna S. Butsky, Lauren Corlies, et al. Enzo: An adaptive mesh refinement code for astrophysics. *The Astrophysical Journal Supplement Series*, 211, 2019.

[10] Lei Cao, Bradley W. Settlemyer, and John Bent. To share or not to share: comparing burst buffer architectures. In *SpringSim*, 2017.

[11] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale i/o workloads. In *International Conference on Cluster Computing and Workshops*, pages 1–10, New Orleans, 2009. IEEE.

[12] 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 (ALS 2000)*, 2000.

[13] Qi Chen, Kang Chen, Zuo-Ning Chen, Wei Xue, Xu Ji, and Bin Yang. Lessons learned from optimizing the sunway storage system for higher application i/o performance. *Journal of Computer Science and Technology*, 35(1):47–60, 2020.

[14] Peter F Corbett and Dror G Feitelson. The vesta parallel file system. *ACM Transactions on Computer Systems (TOCS)*, 14(3):225–264, 1996.

[15] Cray. Lumi supercomputer, 2022. https://www.lumi-supercomputer.eu/.

[16] Bin Dong, Surendra Byna, Kesheng Wu, Prabhat, Hans Johansen, Jeffrey N. Johnson, and Noel Keen. Data elevator: Low-contention data movement in hierarchical storage system. *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 152–161, 2016.

[17] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *TOS*, 17(4):1–32, 2021.

[18] Jack Dongarra. Report on the fujitsu fugaku system. *University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06*, 2020.

[19] Matthieu Dorier, Gabriel Antoniu, Robert Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[20] Brian C. Van Essen, Hyojin Kim, Roger A. Pearce, Kofi Boakye, and Barry Y. Chen. Lbann: livermore big artificial neural network hpc toolkit. *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, 2015.

[21] Fujitsu. File system and power management enhanced for supercomputer fugaku, 2021. https://www.fujitsu.com/.

[22] Donna L Giltrap, Changsheng Li, and Surinder Saggar. DNDC: A process-based model of greenhouse gas fluxes from agricultural soils. *Agriculture, Ecosystems & Environment*, 2010.

[23] Thomas M Hamill. Ensemble-based atmospheric data assimilation. *Predictability of weather and climate*, 124:156, 2006.

[24] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J Wright. Architecture and design of cray datawarp. *Cray User Group CUG*, 2016.

[25] Frank Herold, Sven Breuner, and Jan Heichler. An introduction to beegfs, 2014.

[26] Jonathan Hines. Stepping up to summit. *Computing in science & engineering*, 20(2):78–82, 2018.

[27] Xu Ji, Bin Yang, Tianyu Zhang, Xiaosong Ma, Xiupeng Zhu, et al. Automatic, application-aware i/o forwarding resource allocation. In *17th USENIX Conference on File and Storage Technologies*, pages 265–279, 2019.

[28] chen Kang, Wu Yongwei, and zheng Weiming. Madfs: a high performance burst buffer file system. *Big Data*, 7(3):150, 2021.

[29] Jeongnim Kim, Andrew D. Baczewski, Todd D. Beaudet, Anouar Benali, et al. Qmcpack: an open source ab initio quantum monte carlo package for the electronic structure of atoms, molecules and solids. *Journal of Physics: Condensed Matter*, 30, 2018.

[30] Youngjae Kim, Scott Atchley, and Galen M. Shipman. LADS: Optimizing data transfers using layout-aware data scheduling. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[31] Steven Langer, Abhinav Bhatele, and Charles H. Still. pf3d simulations of laser-plasma interactions in national ignition facility experiments. *Computing in Science & Engineering*, 16:42–50, 2014.

[32] Paul Hermann Lensing, Toni Cortes, Jim Hughes, and André Brinkmann. File system scalability with highly decentralized metadata on independent storage devices. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 366–375. IEEE, 2016.

[33] Yan Li, Xiaoyuan Lu, Ethan L. Miller, and Darrell D. E. Long. ASCAR: Automating contention management for high-performance storage systems. In *IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2015.

[34] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, et al. Shentu: processing multi-trillion edge graphs on millions of cores in seconds. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 706–716. IEEE, 2018.

[35] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2012.

[36] Yong Liu, Xin Liu, Fang Li, Haohuan Fu, Yuling Yang, Jiawei Song, Pengpeng Zhao, Zhen Wang, Dajia Peng, Huarong Chen, et al. Closing the" quantum supremacy" gap: achieving real-time simulation of a random quantum circuit using a new sunway supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2021.

[37] Zhao Liu, XueSen Chu, Xiaojing Lv, Hongsong Meng, Shupeng Shi, Wenji Han, Jingheng Xu, Haohuan Fu, and Guangwen Yang. Sunwaylb: Enabling extreme-scale lattice boltzmann method based computing fluid dynamics simulations on sunway taihulight. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 557–566. IEEE, 2019.

[38] LLNL. Hacc i/o benchmark summary, 2017. https://asc.llnl.gov/sites/asc/files/2020-06/HACC_IO_Summary_v1.0.pdf.

[39] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. Daos and friends: a proposal for an exascale storage system. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 585–596. IEEE, 2016.

[40] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Input/output apis and data organization for high performance scientific computing. In *2008 3rd Petascale Data Storage Workshop*, pages 1–6. IEEE, 2008.

[41] Satoshi Matsuoka. Being "bytes-oriented" in hpc leads to an open big data/ai ecosystem and further advances into the post-moore era. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 5–5. IEEE Computer Society, 2017.

[42] Adam Moody, Danielle Sikich, Ned Bass, Michael J. Brim, and others. Unifyfs: A distributed burst buffer file system - 0.1.0, 10 2017.

[43] Sarp Oral, Sudharshan S Vazhkudai, Feiyi Wang, Christopher Zimmer, Christopher Brumgard, Jesse Hanley, George Markomanolis, Ross Miller, Dustin Leverman, Scott Atchley, et al. End-to-end i/o portfolio for the summit supercomputing ecosystem. In *Proceedings of the International Conference for High Performance*

Computing, Networking, Storage and Analysis, pages 1–14, 2019.

[44] ORNL. Frontier exascale system, 2022. https://www.olcf.ornl.gov/frontier/.

[45] Tirthak Patel, Suren Byna, Glenn K Lockwood, Nicholas J Wright, Philip Carns, Robert Ross, and Devesh Tiwari. Uncovering access, reuse, and sharing characteristics of {I/O-Intensive} files on {Large-Scale} production {HPC} systems. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 91–101, 2020.

[46] Md Mostofa Ali Patwary, Suren Byna, Nadathur Rajagopalan Satish, Narayanan Sundaram, Zarija Lukić, Vadim Roytershteyn, Michael J Anderson, Yushu Yao, Pradeep Dubey, et al. Bd-cats: big data clustering at trillion particle scale. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[47] Dajia Peng, Yunlong Feng, Yong Liu, Xin Liu, Wei Xue, Dexun Chen, Jiawei Song, and Zuoning Chen. Jdebug: A fast, non-intrusive and scalable fault locating tool for ten-million-scale parallel applications. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[48] Yingjin Qian, Xi Li, Shuichi Ihara, Andreas Dilger, Carlos Thomaz, Shilong Wang, Wen Cheng, Chunyan Li, Lingfang Zeng, Fang Wang, et al. Lpcc: hierarchical persistent client caching for lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[49] WG Redis. Redis, 2016. http://redis.io/topics/faqAccessedNovember.

[50] RocksDB. A persistent key-value store for fast storage environments, 2022. http://rocksdb.org/.

[51] RSE-Cambridge. The data accelerator, 2022. https://www.hpc.cam.ac.uk/research/data-acc.

[52] SchedMD. Slurm workload manager, 2022. https://slurm.schedmd.com/overview.html.

[53] Hongzhang Shan and John Shalf. Using ior to analyze the i/o performance for hpc platforms. Technical report, Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US), 2007.

[54] RD Smith and PR Gent. Reference manual for the Parallel Ocean Program (POP), ocean component of the Community Climate System Model (CCSM2. 0 and 3.0). Technical report, Technical Report LA-UR-02-2484, Los Alamos National Laboratory, Los Alamos., (2002).

[55] Guangyu Sun, Jenő Kürti, Péter Rajczy, Miklós Kertész, Jürgen Hafner, and Georg Kresse. Performance of the vienna ab initio simulation package (vasp) in chemical applications. *Journal of Molecular Structure-theochem*, 624:37–45, 2003.

[56] Osamu Tatebe, Shukuko Moriwake, and Yoshihiro Oyama. Gfarm/bb — gfarm file system for node-local burst buffer. *Journal of Computer Science and Technology*, 35:61–71, 2020.

[57] Osamu Tatebe, Kazuki Obata, Kohei Hiraga, and Hiroki Ohtsuji. Chfs: Parallel consistent hashing file system for node-local persistent memory. *International Conference on High Performance Computing in Asia-Pacific Region*, 2022.

[58] Marat Valiev, Eric J. Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P. Straatsma, Hubertus Van Dam, D. Wang, Jarek Nieplocha, Edoardo Aprá, Theresa L. Windus, and Wibe A. de Jong. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Comput. Phys. Commun.*, 181:1477–1489, 2010.

[59] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. Gekkofs-a temporary distributed file system for hpc applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 319–324. IEEE, 2018.

[60] Chen Wang, Kathryn Mohror, and Marc Snir. File system semantics requirements of hpc applications. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 19–30, 2021.

[61] Teng Wang, W Yu, K Sato, A Moody, and K Mohror. Burstfs: A distributed burst buffer file system for scientific applications. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016.

[62] Yulei Wang, Jian Liu, Hong Qin, Zhi Yu, and Yicun Yao. The accurate particle tracer code. *Computer Physics Communications*, 2017.

[63] Junjie Wu, Yong Liu, Baida Zhang, Xianmin Jin, Yang Wang, Huiquan Wang, and Xuejun Yang. A benchmark test of boson sampling on tianhe-2 supercomputer. *National Science Review*, 5(5):715–720, 2018.

[64] Kesheng Wu, Surendra Byna, and Bin Dong. Vpic io utilities. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2018.

[65] Jianyuan Xiao, Junshi Chen, Jiangshan Zheng, Hong An, Shenghong Huang, et al. Symplectic structure-preserving particle-in-cell whole-volume simulation of tokamak plasmas to 111.3 trillion particles and 25.7 billion grids. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2021.

[66] Weixia Xu, Yutong Lu, Qiong Li, Enqiang Zhou, Zhenlong Song, Yong Dong, Wei Zhang, Dengping Wei, Xiaoming Zhang, Haitao Chen, et al. Hybrid hierarchy storage system in milkyway-2 supercomputer. *Frontiers of Computer Science*, 8(3):367–377, 2014.

[67] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, et al. End-to-end {I/O} monitoring on a leading supercomputer. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 379–394, 2019.

[68] Bin Yang, Yanliang Zou, Weiguo Liu, and Wei Xue. An end-to-end and adaptive i/o optimization tool for modern hpc storage systems. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1294–1304. IEEE, 2022.

[69] Hisashi Yashiro, Koji Terasaki, Yuta Kawai, Shuhei Kudo, Takemasa Miyoshi, Toshiyuki Imamura, Kazuo Minami, Masuo Nakano, Chihiro Kodama, Masaki Satoh, et al. The nicam 3.5 km-1024 ensemble simulation: Performance optimization and scalability of nicam-letkf on supercomputer fugaku. In *EGU General Assembly Conference Abstracts*, pages EGU21–4771, 2021.

[70] Y. Ye, Z. Song, S. Zhou, Y. Liu, Q. Shu, B. Wang, W. Liu, F. Qiao, and L. Wang. swnemo_v4.0: an ocean model based on nemo4 for the new-generation sunway supercomputer. *Geoscientific Model Development*, 15(14):5739–5756, 2022.

[71] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Rob Ross, and Gabriel Antoniu. On the root causes of cross-application i/o interference in hpc storage systems. pages 750–759, 05 2016.

[72] Hao Yu, Ramendra K Sahoo, C Howson, George Almasi, José G Castanos, Manish Gupta, José E Moreira, Jeffrey J Parker, TE Engelsiepen, Robert B Ross, et al. High performance file i/o for the blue gene/l supercomputer. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 187–196. IEEE, 2006.

Fisc: A Large-scale Cloud-native-oriented File System

Qiang Li[◇], Lulu Chen[†◇], Xiaoliang Wang[‡], Shuo Huang[◇], Qiao Xiang[⋆],
Yuanyuan Dong[◇], Wenhui Yao[◇], Minfei Huang[◇], Puyuan Yang[◇], Shanyang Liu[◇],
Zhaosheng Zhu[◇], Huayong Wang[◇], Haonan Qiu[◇], Derui Liu[◇], Shaozong Liu[◇], Yujie Zhou[◇],
Yaohui Wu[◇], Zhiwu Wu[◇], Shang Gao[◇], Chao Han[◇], Zicheng Luo[◇], Yuchao Shao[◇],
Gexiao Tian[◇], Zhongjie Wu[◇], Zheng Cao[◇], Jinbo Wu[◇], Jiwu Shu[⋆], Jie Wu[†], Jiesheng Wu[◇],
[◇]Alibaba Group, [†]Fudan University, [‡]Nanjing University, [⋆]Xiamen University

Abstract

Despite the progress of cloud-native technologies, existing distributed file systems are ill-suited for multi-tenant cloud-native applications for two reasons, First, their clients are typically heavyweight, resulting in a low level of resource multiplexing among containers. Second, their architecture is based on network gateway and falls short in providing efficient, highly-available and scalable I/O services for cloud-native applications. In this paper, we propose Fisc, a large-scale, cloud-native-oriented distributed file system. Fisc introduces three key designs: (1) a lightweight file system client to improve the multiplexing of resources with a two-layer resource aggregation, (2) a storage-aware distributed gateway to improve the performance, availability and scalability of I/O services, and (3) a DPU-based virtio-Fisc device offloading key functions to hardware. Fisc has been deployed in production for over three years and now serves cloud applications running over 3 million cores. Results show that Fisc client only consumes 69% CPU resources compared to the traditional file system client. The production environment shows that the average latency of online searching tasks is less than 500 μs when they access their files in Fisc.

1 Introduction

Many applications, such as data analytics [1], machine learning [2], and transactional workflows [3, 4] are deployed in public clouds. The emerging cloud-native technologies are shifting virtualization in clouds from virtual machines (VM) to containers and pushing up the abstraction provided to tenants from resources (*e.g.*, CPU and memory) to services (*e.g.*, database and object storage service). As such, cloud service providers (CSPs) must rethink their fundamental services to provide efficient, flexible support to cloud-native applications.

Specifically, file system (FS) is one such fundamental service, with which applications can store and access their data [5–7]. Tenants typically employ FS in the cloud in one of two modes. They either purchase cloud storage (*e.g.*, SSD) and deploy their own FS, or directly use the FS service provided by CSPs. As CSPs gradually switch from server- centric to resource-disaggregated architectures, tenants increasingly use the second approach for its elasticity, flexibility, on-demand charging, and ease of use [6, 8, 9].

File systems need to be redesigned to support cloud-native applications. Existing distributed file systems (*e.g.*, [5–7]) are ill-suited for multi-tenant cloud-native applications for two reasons. First, clients in these systems have a low level of resource multiplexing among containers. That hinders CSPs from achieving high efficiency of resources and makes it difficult for each computation server to support a large number of containers for cloud-native applications. Specifically, these clients typically adopt a heavyweight design to provide many functionalities, including interfaces for interacting with applications, storage protocols for data persistence and failure handling, network-related functions for communications with data nodes and metadata masters, and security-related functions for authorization checking. As such, each client needs to reserve many exclusive resources, and a server can host only a small number of containers concurrently, resulting in inefficient use of resources.

Second, a centralized network gateway employed for file system service in the cloud cannot satisfy the requirement of cloud-native applications for performance, availability, and load balancing. A network gateway is a component that connects clients in the virtual domain of users to backend proxies in the physical domain of CSPs. This network-gateway-based architecture has a series of limitations, including (1) a suboptimal, ms-level latency to pass through the gateway, (2) the incapability of data locality optimization and fast failure handling due to the unawareness of file semantics and storage protocols, (3) the incompatibility with high-performance network stack like RDMA without intrusive changes to clients, and (4) the load balancing gap between network connections and files. Besides, to match the throughput of a large-scale file system of thousands of nodes, it would take non-negligible costs for CSPs. Luna and Solar [10] propose storage network stacks for Alibaba's EBS service. However, they only focus on achieving high performance within the physical domain of CSPs. They cannot provide high performance for the whole path from the file clients in the virtual domain of users to the storage clusters in the physical domain of CSPs.

Fisc: a cloud-native-oriented file system. In this paper, we design Fisc, a cloud-native-oriented distributed file system service to provide cloud-native applications with high-performance, high-availability storage services at low cost. Fisc consists of two key components: lightweight clients and a storage-aware distributed gateway (SaDGW).

First, with a two-layer aggregation, Fisc moves user-unaware functionalities (*e.g.*, network stacks and storage protocols) out of clients in the containers and offloads them to the Data Processing Units (DPU) of computation servers and the backend storage nodes of CSPs to aggregate their resources, respectively. As a result, the resources used for these functionalities can be fully multiplexed, lowering the amortized cost. Meanwhile, as each client consumes substantially fewer resources, a computation server can host a large number of containers for cloud-native applications.

Second, Fisc introduces SaDGW to provide a direct highway with a high-performance network stack [10] between the computation and storage servers. Specifically, we leverage the file system semantics on the highway path to build a storage-aware routing mechanism to route clients' file requests from the frontend virtual domain of tenants to the backend physical domain of CSPs with a granularity of files instead of network flows. We design a series of mechanisms, such as storage-aware failure handling and locality-aware read optimization, to improve the availability of Fisc. We have also employed a file-based fine-grained scheduling mechanism to balance loads of proxies at storage nodes.

Implementing Fisc with a software-hardware co-design. Realizing lightweight clients and SaDGW completely in software is inefficient. As such, we leverage the emerging DPUs to implement part of the functionalities of clients and the core functionalities of SaDGW. We adopt a virtio-Fisc device in DPU to offload the network stacks and storage protocols and provide secure and high-efficient passthrough from the users' virtual domain containers to the file system of CSP's physical domain. We also leverage the fast path in DPU to accelerate the I/O processing, further improving the performance of Fisc.

Production deployment. Fisc has been deployed in production DCN for three years and serves applications running on over 3 million cores in Alibaba. For large-scale development, it presents an abstracted virtual RPC (vRPC) based on SaDGW and virtio-Fisc devices, which is easy to use and can be adopted by other cloud-native services like Function as a Service (FaaS). Compared to the on-premise Pangu client, the CPU and memory consumption of the Fisc client is reduced by 69% and 20%, respectively. The availability is improved by an order of magnitude (*e.g.*, failure recovery from a second-level to a 100ms-level). For the online-search query service, its average and P999 latency in Fisc are <500 μs and <60 ms, respectively. Its average latency jitter is less than 5%.

2 Background and Motivation

2.1 File Systems

File system (FS) is a fundamental service for users to store and access their data. Large-scale distributed file systems like Tectonic [7], Colossus [5], and Pangu [6] have been developed by different companies in their datacenters. Generally, they consist of three components, *masters, data servers,* and *clients*. The *masters* manage data servers and maintain the metadata of the whole system (*e.g.*, the file namespace and the mapping from file chunks to data servers). The *data servers* are storage nodes responsible for managing file chunks and storing their data on storage media (*e.g.*, HDDs and SSD). The *clients* interact with the masters for metadata and the data servers for data. Notice that clients in representative large-scale file systems (*e.g.*, Tectonic [7] and Colossus [5]) are heavyweight. They provide complex functions, including not only storage protocols for data persistence and failure handling but communication with masters and data servers, as well as security-related functions such as authorization.

Pangu [11] is a large-scale distributed storage system in Alibaba and provides append-only file semantics like HDFS [12]. It works as a unified storage core of Alibaba Cloud. Multiple businesses (*e.g.*, Elastic Block Service [10, 13], Object Storage Service [14], Network Attached Storage [15], and MaxCompute [16]) are built on top of Pangu. They adopt the Pangu clients for persistent, append-only file storage, employ a key/value-like index mapping to update data, and use a garbage collection mechanism to compress historical data.

2.2 Cloud Native

With the development of cloud-native technology (*e.g.*, microservice, container, and serverless computing), more and more tenants are deploying their applications into the public cloud and directly using the services provided by CSPs (*e.g.*, database and object storage service). In 2020, Alibaba also migrated all its core businesses, such as Taobao and Tmall, to cloud-native containers. Cloud-native technologies substantially simplify the development and operation of tenants and demonstrate two characteristics. First, with fine-grained containers being used instead of VMs, the number of containers in a computation server can exceed 1000 [17, 18], *i.e.*, ∼10 times more than that of VMs. Second, cloud-native technologies push up the abstraction provided to tenants from VMs to services. The implementation of services is transparent to tenants but must provide high performance under heterogeneous workloads. To this end, bare-metal DPUs are increasingly used to accelerate cloud-native applications. For example, AWS adopts Nitro [19] and Alibaba adopts X-Dragon [20,21]. These bare-metal DPUs utilize the virtio technology for I/O virtualization and can provide high-performance support to a broad range of cloud services.

2.3 Motivation

Cloud-native applications bring new challenges for CSPs to provide file system service.

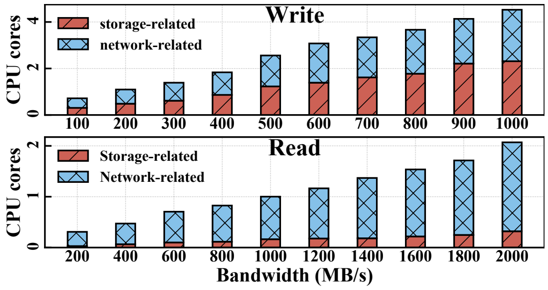


Figure 1: The CPU consumption of an HDFS client under different I/O bandwidths.

Isolated file system clients cause low resource utilization. In traditional file systems [5, 7], a client is responsible for multiple tasks, including storage protocols of reliability and consistency, failure handling, network-related functions, and authorization-related functions. As such, applications usually pre-allocate I/O threads and reserve memory resources and network connections for file system clients. Because the resources of FS clients in containers are isolated from each other, the resource utilization of CSPs is low. As a result, achieving a high density of over 1,000 containers in a computation server is difficult. Take the resource consumption of an HDFS client in a Hadoop-2.10.2 cluster of three Intel(R) Xeon(R) Gold 5218 servers as an example. Figure 1 plots the CPU consumption of the HDFS client under different read and write bandwidths. Even if the client writes files at a bandwidth of 200 MB/s, it consumes 1.1 CPU cores. Consider a typical scenario where a container is allocated two cores. It means over 50% CPU resources are spent on I/O.

We make a key observation that many common functions (*e.g.*, storage protocols and network stacks) of different FS clients can be aggregated to achieve more efficient resource sharing. With this aggregation, we can provide a lightweight file system interface for different tenants, and simplify the maintenance and upgrade of FS clients.

Network gateway becomes the bottleneck. FS clients of cloud-native applications are in the virtual domain of users, while the file system resides in the physical domain of CSPs. For security reasons, clients cannot directly access the file system but have to use a network gateway (*i.e.*, network load balancer) to access the data. However, this network gateway cannot satisfy the requirements of cloud-native applications on file services in terms of performance, availability, load balance, and cost.

- *Performance.* Performance-critical cloud-native applications (*e.g.*, interactive applications [22]) require a 100μs-level storage access latency. Although file systems such as Pangu are equipped with high-performance SSD and RDMA in the backend cluster [6], which provides a 100μs-level latency, an I/O request needs to go through multiple hops in a network-gateway-based architecture, resulting in a second-level or ms-level latency [23, 24].
- *Availability.* Cloud-native applications often require a ms-level recovery latency [25] in the case of storage system failures (*e.g.*, network jitters and server breaking down). However, with a network gateway, file systems can only support second-level failure handling [26, 27] due to the gap between files and network connections. Specifically, the network-connection-based Service Level Agreement (SLA) is substantially different from the file-based SLA of file systems. As such, it is hard to leverage storage protocols in a network-gateway-based architecture to improve the availability of file systems.
- *Load balance.* The network gateway distributes the load to different proxies based on the number of network connections. That may lead to a significant load imbalance of files among the proxies due to the semantics gap between files and connections. For example, the load among proxies can be as much as ten-fold different in the NAS service in Alibaba Cloud [15]. In addition, the gateway may direct a read request to a storage server with no requested data. The server must forward the request to another storage server that has the data, which will amplify the traffic.
- *Cost.* A large-scale file system requires a large amount of hardware dedicated to the network gateway in order to match the total throughput of its storage cluster, which typically consists of thousands of storage nodes. Given a cluster of 10,000 storage nodes, each of which is equipped with a 25×2 Gbps NIC, its total throughput is 500 Tbps. If the throughput of a network gateway machine is 100 Gbps, we need 5,000 gateway machines to match the total throughput of the cluster, which introduces a non-negligible cost for CSPs.

3 Overview of Fisc

In this section, we give an overview of Fisc, including its design rationale, architecture and basic workflow.

3.1 Design Rationale

Aggregating the resources of FS clients. Resource aggregation is the nature of cloud computing, which can improve resource utilization and provide elastic, efficient, and on-demand cloud service. In contrast to the traditional resource-intensive FS clients, we aggregate functions like storage protocol and network-related functions by offloading them to the CSP's domain (*e.g.*, the DPUs at computation and storage servers). Meanwhile, this aggregation allows CSP to provide a reservation-only interface with a lightweight client for cloud-native applications. As such, it allows a computation server to host a large number of application containers concurrently.

Storage-aware distributed gateway. Instead of using a centralized network gateway, we resort to a distributed storage-aware gateway to set up direct highways between each computation server and its corresponding remote storage nodes. This design allows us to adopt high-performance network protocols connecting the virtual and physical domains. It also leverages storage semantics on the highways to improve the availability and locality of file access requests and guarantee the load balance among storage nodes.

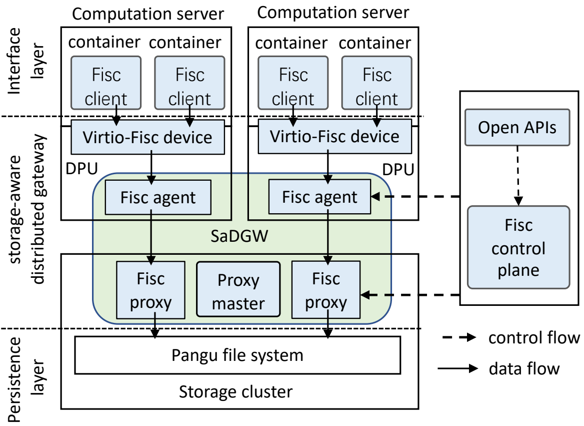


Figure 2: The architecture of Fisc.

Software and hardware co-design. To improve the efficiency and performance of the file system service, we leverage the emerging DPUs deployed in physical servers. Through careful hardware and software co-design, we can implement secure, efficient passthrough from the users' containers in the virtual domain to the file system of CSP's physical domain. Moreover, we can also introduce a fast path in DPU to accelerate the I/O processing.

3.2 Architecture

As shown in Figure 2, Fisc consists of a control plane and a data plane. The control plane provides open APIs for tenants to create Fisc FS instances, mount the Fisc FS to their VM/containers, and allocate virtio devices to accelerate the passthrough from the virtual domain to the physical domain.

Fisc's data plane consists of three layers: interface layer, storage-aware distributed gateway, and persistence layer. The lightweight Fisc client is placed in the frontend, which provides FS service interfaces for applications. The distributed storage-aware distributed gateway (SaDGW) is in the middle layer, composed of Fisc agents in the DPU of each computation server, Fisc proxies in each storage node, and a group of Fisc proxy masters in the storage cluster. The Fisc proxy masters are responsible for managing Fisc proxies and Fisc agents. The backend persistence layer is Pangu, which is responsible for processing the requests and persisting the data in storage media.

Lightweight Fisc client. The aggregation of client resources occurs at the Fisc agent in the DPU of each computation server and the Fisc proxy in each storage server. We dissect the functions of FS clients and make careful aggregation tradeoffs to decide where these functions should be aggregated (*i.e.*, Fisc agents or proxies). We also design mechanisms to simplify the implementation of Fisc clients and maintain compatibility across different versions of their software libraries.

SaDGW. This gateway gives full play to the 100μs-level high-speed SSD and RDMA technologies via direct and high-performance network connections between Fisc agents and Fisc proxies. Based on the file granularity routing in each Fisc Agent, it leverages the storage semantics on the route to eliminate the gap between network and file to achieve a P999 ms-level SLA. Moreover, it implements a locality-aware read mechanism that avoids the read traffic amplification and doubles the read throughput.

HW and SW co-design on DPU. Fisc provides a virtio-Fisc device to build up secure and efficient passthrough from virtual containers to the physical storage cluster. Based on the device, a co-designed FPGA cache is presented as a fast path to further improve Fisc's performance. With regard to the scarce resource of DPU, optimizations for CPU, memory, and network are proposed.

With these three modules, we further provide a vRPC (virtual RPC) abstraction for storage service, which can be easily adopted by cloud-native services. Besides, Fisc adopts an end-to-end (E2E) QoS mechanism for different priority applications like online search and offline training. With proxy master scheduling, Fisc builds up file-granularity load balancing among Fisc Proxies, which avoids the imbalance caused by traditional network connection-based scheduling.

3.3 Workflow of Fisc

In the control plane, when a tenant calls the open APIs to create a Fisc instance, Fisc control plane maps the instance to the backend Pangu file system, and pushes the information of the tenant and the mount point to the Fisc proxy masters deployed in the Pangu storage cluster. The Fisc proxy master pushes the proxy mapping (*i.e.*, the mapping between the mount point and the Fisc proxies) to the Fisc agent whenever the tenant attaches the mount point to a VM/container. In the end, the control plane attaches a virtio-Fisc device to the corresponding VM/container.

The workflow of the data plane mainly involves SaDGW with a fine-granularity route table. Given a meta operation request of files of the mount point, it arrives at the Fisc agent through the virtio-Fisc device. The Fisc agent randomly chooses a Fisc proxy according to the mapping between the mount point and Fisc proxies. If it is an open operation for a file, a route entry associated with the opened file will be constructed with its file handle and the Fisc proxy location. Afterwards, the subsequent read/write requests of the file will be routed according to the route entry. More details of storage-aware routing optimizations are in §4.2.

4 Design and Implementation

4.1 Lightweight Fisc Client

We adopt a lightweight design for Fisc clients by offloading most of their functions to Fisc agents in the DPU of computations servers and Fisc proxies on the storage nodes. Through this two-layer function aggregation, Fisc achieves a high level of resource multiplexing. In addition, we also introduce a unified RPC-based method to simplify the implementation of Fisc clients and a mechanism similar to Protocol Buffers (PB) to maintain compatibility across their different versions.

4.1.1 Function Offloading and Aggregation Tradeoff

Typical heavyweight FS clients [5–7] provide four types of functions: (1) file interfaces and structures (*e.g.*, APIs and file handlers), (2) storage-related protocols (*e.g.*, replication reliability, data consistency, and failure handling), (3) security and authentication (*e.g.*, authorization checking) and (4) network-related protocols (*e.g.*, RPC with data nodes or metadata nodes). We make a *key observation* that in cloud-native applications, users are only interested in the first type of functions and the implementations of other functions are transparent to users. Therefore, we can move the latter three functions out of Fisc clients and aggregate them to achieve a high level of multiplexing on resources. However, the locations where they are aggregated (*i.e.*, Fisc agents or Fisc proxy) have a great impact on the effects of multiplexing. We elaborate on our offloading designs of different functions.

Offloading network-related functions to Fisc agent. We offload the network-related functions of conventional clients to the Fisc agent in the DPU of the computation server. This is motivated by the recent success of DPU-based high-performance network stacks (*e.g.*, Luna/Solar [10] and Nitro SRD [28]) in the physical domain of CSPs. In particular, a Fisc agent extends Luna/Solar network stack and aggregates multiple network connections of Fisc clients on the same computation server. This substantially reduces the CPU and memory resources each client needs to reserve for network-related operations.

Offloading security-related functions to Fisc agent. We adopt an early-checking design to perform security checks (*e.g.*, authentication and authorization) in Fisc agent when it receives requests from Fisc clients. Different from the methods with network gateway, which deal with the malicious traffic at their proxies, this design prevents malicious traffic from consuming the resources of backend storage clusters.

Offloading storage-protocol functions to Fisc proxy. We choose to offload storage-protocol functions to Fisc proxies in the storage clusters, instead of Fisc clients, for three reasons. First, the DPU in the computation server has limited resources. After spending resources on network-related functions, security-related functions and bare-metal virtualization of virtio-Fisc device, the DPU does not have sufficient resources to implement complex storage protocols. Second, offloading these functions to the storage clusters helps move the storage traffic between the computation servers and the storage clusters in the backend network within storage clusters, saving the scarce network resources in the compute-storage disaggregated architecture. Third, it allows us to adopt storage-oriented optimization and hardware-assisted accelerations in the storage clusters to improve the overall system performance and reduce costs.

4.1.2 Simplification and Compatibility

Implementing an FS client and maintaining its compatibility across different versions of its software library is challenging because a typical FS client has a large number of APIs (*e.g.*, the HDFS client has more than 100 APIs [29]). We introduce a unified RPC-based method to simplify the implementation of Fisc clients and a mechanism similar to protocol buffers for compatibility maintenance.

Simplifying client implementation using RPC. We implement the APIs in the Fisc client using RPC stubs. When the application invokes an API, the Fisc client passes the parameters of the API to its corresponding RPC stub. The stub encodes these parameters, the file handle, and the tenant information into an RPC request. This request is sent to the Fisc agent in the DPU with a virtio-Fisc device (§4.3.1). The Fisc agent checks the authorization of tenants and looks up the file handle in its route table (§4.2.1) to forward the RPC request to a corresponding Fisc proxy. Upon receiving the request, the Fisc proxy resolves it and invokes the corresponding RPC service of the API, which completes the API and encodes its return value in an RPC response. The response is returned to the Fisc client along the opposite path of the RPC request and resolved by the client. This design makes it easier to implement and add APIs in Fisc clients.

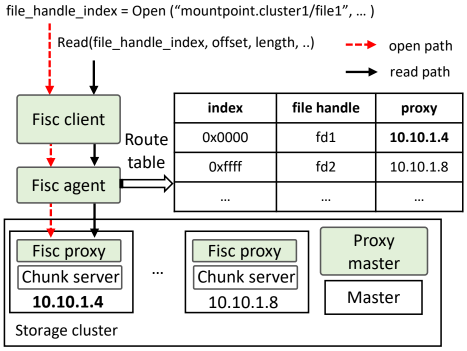
Maintaining compatibility using a PB-based mechanism. Building on top of the RPC-based API implementation, we introduce a PB-based mechanism to maintain the compatibility of Fisc clients across different versions. Directly applying the PB protocol [30] would introduce extra data center tax of (de)serialization [31], wasting the limited resources in DPU. To this end, we categorize Fisc APIs into data-related ones (*e.g.*, read and append) and meta-related ones (*e.g.*, create, delete, open and close). Although the former has fewer APIs, it is more frequently used than the latter. Thus, for data-related APIs, we adopt several carefully designed, efficient data structures to maintain their compatibility. For the meta APIs, we use the PB protocol as it is. In this way, we can achieve a balance between performance and compatibility.

4.2 Storage-aware Distributed Gateway

SaDGW is a distributed gateway that sets up direct connections, referred to as "direct highways" in the paper, between the Fisc agents and the Fisc proxy. As such, Fisc can adopt high-performance network stacks on these direct highways, and further leverage storage semantics to build a file-granularity storage-aware routing. It improves the availability through storage-aware failure handling and improves the read throughput through locality optimizations.

4.2.1 Direct Highway Between Agents and Proxies

Direct highway. With the help of DPUs, Fisc builds direct highways between Fisc agents and Fisc proxies, where no network gateways are needed. Considering a storage cluster with thousands of nodes, this would be a significant cost saving. On the highways, we adopt high-performance network stacks of Luna/Solar [10], which is transparent to cloud-native applications, instead of the TCP/IP stack. Raw data structures [32] are adopted to eliminate the overhead of (de)serialization between Fisc agents and proxies.



index	file handle	proxy
0x0000	fd1	**10.10.1.4**
0xffff	fd2	10.10.1.8
…	…	…

Figure 3: The routing process of Fisc.

File granularity route table. SaDGW manages highways through a centralized control mechanism (§3.3). As shown in Figure 3, Fisc agent adopts a file-granularity route table for routing file requests to Fisc proxies, which records the file handle information and the location of the Fisc proxy serving the file. For the route table, one entry is inserted once a file is opened for the first time. When Fisc agent receives a file open request, it randomly chooses a Fisc proxy from the proxy mapping. An entry is constructed when a response of successful file open is returned. The entry includes the returned file handle, the location of the chosen proxy, and the SLA-related attributes mentioned below. Afterwards, when an I/O request arrives at the Fisc agent, it looks up a proxy in the route table with the file handle of the request and then transmits the request to the proxy. Due to the scarce memory in DPU, Fisc uses an LRU policy to control the size of the route table.

4.2.2 Storage-aware Failure Handling

Enhanced route entry. Based on the file granularity route table, Fisc further leverages storage semantics to improve its availability. For failure handling of storage protocols, three main factors are considered: *retry timeout*, *retry destination*, and *highway quality*. 1) The *retry timeout* means the maximum number of times the Fisc agent retries the failed requests, which is related to the request timeout set by users and highway quality; 2) The *retry destination* denotes the proxy location in the entry, which will be replaced by a new proxy if retry timeout occurs; and 3) The *highway quality* is measured by the average latency to estimate the network quality to the proxy. Therefore, we enhance the route table to support failure handling. Besides the file handle and proxy location, each route entry is extended with three items: *retry times*, *retry timeout*, and *avg-latency*, which record when the agent reset the connection, the condition under which the agent gives up, and the average latency of requests, respectively. We make use of these items to implement storage-aware failure handling.

Failure handling. Fisc leverages several mechanisms in the Fisc agent to conduct failure handling.

- *Retry.* When detecting a failed request, the Fisc agent retries the request several times until it receives a successful response or it exceeds the timeout defined by users. Since users usually set a relatively large timeout for their requests, the Fisc agent initially sets a small empirical timeout (*i.e.*, ten times the average latency) to detect failed requests. When such a request is found, the agent doubles the timeout to execute the retry. This mechanism deals with temporary failures (*e.g.*, network jitters and burst proxy load).
- *Blacklist.* Upon detecting consecutive failures of requests or an abnormally large average latency to a Fisc proxy, the Fisc agent puts this Fisc proxy into the blacklist. A background thread periodically pings these proxies and will remove the successful pinged proxy from the blacklist. The metadata requests in the metadata path will exclude the proxies in the blacklist when choosing Fisc proxies. The data operations in the data path will involve the following reopen mechanism.
- *Reopen.* If the destination Fisc proxy of a request is in the blacklist, Fisc agent will select a new Fisc proxy to reopen the file and update the route entry. Otherwise, for a failed request, it adopts a threshold of retry times to make sure that there is still time left after the retry. In the remaining time, it reopens the file by retrying the request to a new Fisc proxy. This operation provides the opportunity to complete the request with the new proxy and avoid request failure.

These mechanisms are transparent to cloud-native applications. It provides flexibility for CSPs to upgrade the failure handling policy and helps keep the Fisc client lightweight.

4.2.3 Locality-aware Read

For a read operation, its request is first sent to a Fisc proxy and then sent to the Pangu chunkserver where the data to read is located by the proxy. The read response with the data to read is returned along the opposite path: from the chunkserver to the proxy and then to the client. It results in a two-time amplification of the read traffic, which consumes extra bandwidth and reduces the read throughput of the whole cluster by half. Considering that each storage node is deployed with a Fisc proxy process and chunkserver process, we design the locality-aware read by letting Fisc agent record the locations of file chunks and sending read requests to the proxy, where the concurrently deployed chunkserver holds their required chunks.

Predicted locations in a range table. When an open or read response returns to Fisc agents, the location information of the file chunks is piggybacked, as shown in Figure 4. The proxy returns the chunk information that would be read in the near future by the read prediction mechanism of Pangu. The number of the predicted chunks is empirically set to 16. Then the location information is encoded as range and location pairs and inserted into a range table. Each entry of the range table corresponds to a file, and the total number of range pairs in an entry is limited to 64 due to the scarce resource of DPU. For the 64 MB chunk size, its spanned range is 4 GB, which covers a large range space of files. The index of a file's corresponding range table entry is stored as a read hint attribute in a route table entry.

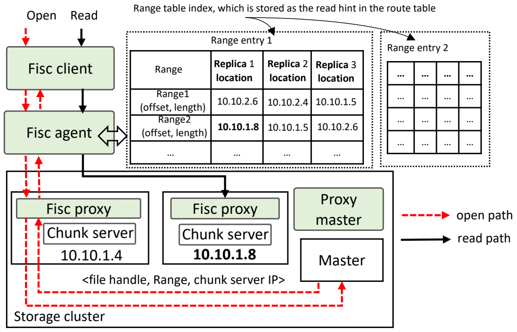


Figure 4: The design of locality-aware read.

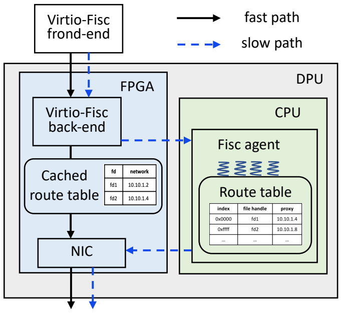


Figure 5: The design of fast path.

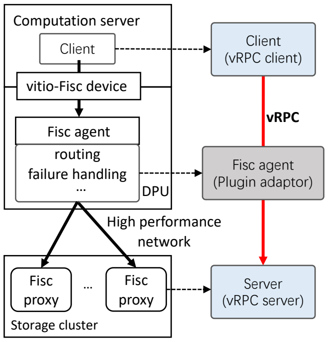


Figure 6: The abstracted vRPC.

Shared memory instead of cross-node communication. When a request arrives at a Fisc agent, the Fisc agent looks up its route entry and finds the read hint, which is an index of the range table. With the index, the Fisc agent then looks up the range table and finds the matching range and location pair. If a pair hits, the read request will be sent to the location in the pair. In some cases, the range for a read, with the offset adding the length to read, is larger than the range of a hit pair. In this case, due to the limitation of CPU resources, we do not divide a read request into multiple ones to avoid complex processing in DPU, such as segmentation, combination and failure handing. When the Fisc proxy of the location receives the read request, it calls the Pangu client to complete the request. As the Pangu client finds that the chunkserver and the proxy is located in the same physical node, it uses shared-memory communication instead of the network. As a result, the data to read is only transmitted once through the network, and increase the total read-throughput of a storage cluster.

4.3 SW/HW Co-design with DPU

Fisc adopts X-Dragon DPU [33] to build a novel virtio-Fisc device to accelerate its secure passthrough from the virtual domain of users to the physical domain of CSPs. To meet the requirements of cloud-native applications, a fast path is applied in DPU, and many optimizations are adopted to mitigate the impact of the scarce resources of DPU.

4.3.1 DPU-based Virtio-Fisc Device

The virtio-Fisc device is a PCIe device following the virtio standard. It consists of two parts, the frontend in VMs/containers and the backend in DPUs. Fisc client puts requests in the virtio hardware queues through the frontend, and Fisc agents running on the processor of DPU process the requests of the hardware queues. Agents send the requests to the Fisc proxies, and put the returned responses into the virtio hardware queues, which are consumed by the frontend. Two generations of virtio-Fisc devices are adopted in Fisc:

Virtio-Fisc devices based on virtio-block. We adopt virtio-block device for its compatibility with major operating systems and can be used by most VMs/containers without modification. With virtio-block interface, the front-end is the same as the standard virtio-block device, and a lightweight communication library is implemented with block read and write operations for Fisc client. However, the backend is quite different, and requests in hardware queues are processed with Fisc agents instead of the traditional virtio-block software, as mentioned in [20]. In fact, it only makes use of the virtio-block interface and works as a virtio-Fisc device.

Virtio-Fisc devices based on customized design. We design a novel vitio-Fisc device to eliminate the limitation of virtio-block. For example, the depth of a virtio-block queue is limited to 128 in most operating systems. Though it is enough for virtio-block but not for nonblocking requests of Fisc. The novel virtio device is more like a NIC device. We further leverage its interface to the RPC level, which can be suitable for cloud-native services like FaaS. It makes use of virtio queues to transmit commands for RPC requests and receive responses. We equip the device driver in our released OS in Alibaba.

4.3.2 Fast Path

We adopt a cache of route table in FPGA of DPU, which generates a fast path to speed up the processing of file requests. As shown in Figure 5, the mapping between the file handle and network connection is cached in the FPGA. With the cache, when a request with its file handle comes to the customized virtio-Fisc device in the FPGA, the FPGA resolves the file handle from the request and looks up the table. If it hits, the request will be directly packed as network packets and transmitted to the network connection. Otherwise, the request will be sent to Fisc agents via the slow path. Entries of the cache are controlled and updated by Fisc agent in software to relieve the complexity of the FPGA implementation of the cache. And to control network transmission bandwidth, the transmission window of each connection is also set and updated in cache entries by the Fisc agent.

4.3.3 Resource Optimizations

With regard to the scarce resources of DPU, optimizations for its CPU, memory and network are applied.

CPU optimization. We leverage two methods to optimize the CPU usage in DPU. 1) *Batch operation.* Fisc gathers multiple requests into one to share the processing of virtio protocols between Fisc clients and agents. 2) *Manual PB (De)Serialization.* Fisc adopts manual PB (De)Serialization methods. They are customized for particular data types of

Fisc and are more efficient than compiler-generated ones. According to our experiment with the manual methods, the IOPS can be improved by about 1.5% for 4 KB requests with one processor core of a DPU.

Memory optimization. The route table and range table occupy most memory in Fisc agent. To save memory, Fisc compresses the memory space of their entries. As the number of storage nodes is less than 1 million, we adopt 20 bits to represent the location IP instead of 32 bits, i.e. 4 bytes, in general. For locations of 3 replicas for a chunk, it consumes 8 bytes in total. A file with 64 predicted chunks for locality-aware read occupies 512 bytes. Taking file handle and tenant information into account, the total size of the memory space for a file in Fisc agent is no more than 1 KB. Thus, 1 GB of memory can hold up to 1 million files.

To further save the memory, we pass the range table to Fisc client as a hint. In this way, there is no need to store a large number of locations for locality-read in the range table in DPU. Instead, they can be stored in Fisc client. Fisc client is aware of the range of chunks and can find the location corresponding to its read request. When Fisc client sends a request, it is accompanied by the hint. Then Fisc agent first checks the file handle and tenant information with the route table in DPU. For a passed request, Fisc agent sends the request to the Fisc proxy according to the hint. For security, the location hint passed to Fisc client is encoded with an index and has no meaning to users. To avoid applications changing the hint maliciously, the index is checked in Fisc agents and Fisc proxies. If the check fails, the locality-read mechanism for the tenants will be forbidden for a period of time in Fisc agent. It then falls back to using the route table of fewer locations in DPU. Thus, with the hint, Fisc can save a lot of memory and support more range table entries in DPU.

Network optimization. SaDGW carefully deals with the number of connections for the direct highway. First, it adopts the shared-connection mechanism [13] to reduce the connections between Fisc agents and Fisc proxies. Second, it recycles the resources of network connections by periodically tearing down idle connections. Third, for the narrow inter-region Tbps-level network bandwidth compared to that of intra-region, Fisc agents only connect parts of proxies in different regions where there can be thousands of storage nodes. In this way, it is sufficient for inter-region network throughput and reduces the number of connections.

4.4 Large-scale Deployment

Fisc carefully deals with ease of use, load balance, and QoS to support applications running over 3 million CPU cores.

4.4.1 vRPC

As shown in Figure 6, we abstract a vRPC service from Fisc. It is similar to the traditional RPC mechanism of RPC client and RPC server. Clients placed in containers call an RPC request by vRPC stub in Fisc client, and the request is processed by the vRPC Server in Fisc proxy. For a cloud-native service, developers only need to concern the RPC stub for clients in containers and its RPC service for servers in the backend clusters. The implementation details of vRPC such as virtio device and SaDGW, are transparent to both clients and servers, which is different from the traditional RPC as follows. First, it provides a secure passthrough from the virtual domain to the physical domain with an efficient hardware-assisted virtio device. Second, its RPC request can be retried in Fisc agent, which is transparent to vRPC client, and high-performance network stacks can also be transparently adopted. Third, it gives an opportunity to adopt an adapter for a service, which can be integrated into Fisc agent to improve the availability of the service by its developers. vRPC can not only support Fisc FS but also other cloud-native services.

4.4.2 Load Balance

Fisc introduces two mechanisms for the load balance among thousands of Fisc proxies in storage clusters.

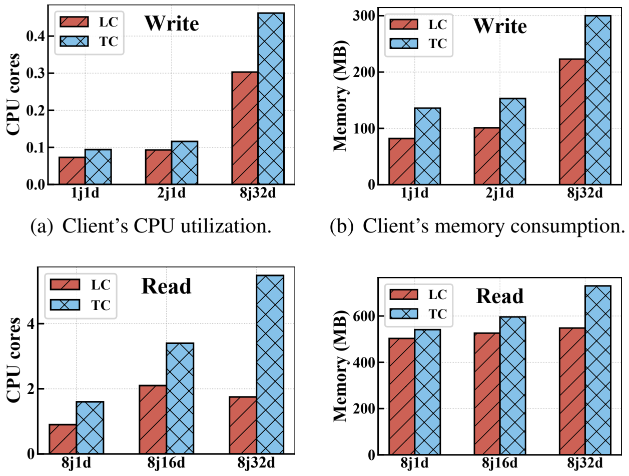
File granularity schedule. Traditional load balance relies on connection-based scheduling of network gateway, which focuses on balancing the number of network connections among proxies. However, a gap exists between the number of network connections and that of files. This means the number of connections may be balanced, but that of files in each proxy may be significantly different. To tackle this problem, Fisc eliminates the gap and presents a file-granularity schedule for load balance. Fisc agent forwards each file to a random Fisc proxy according to the hash value of its file name and other information such as access time. In this way, the files are evenly distributed among Fisc proxies. With locality-aware read optimization, as the chunks of files are evenly distributed to each data server by Pangu, it leads to an even balance of read requests to Fisc proxies, which are currently deployed with data servers in storage nodes.

The centralized re-scheduling. As Fisc proxy masters periodically collect the load of each proxy, they can schedule and migrate part of the files from a high-load Fisc proxy to a low-load one. The migration is transparent to the applications of tenants, and Fisc agent accordingly reopens these files after the migration. Meanwhile, Fisc proxy masters also push the load information to Fisc agents. After receiving the information, Fisc agents reduce the hash weight of high-load Fisc proxies and improve that of low-load ones. In this way, Fisc implements a centralized re-scheduling.

4.4.3 E2E QoS

Fisc supports hybrid file access for online real-time applications and offline batch-processing applications, the demands of which are represented by high priority and low priority.

Hardware-based QoS. Vitio-Fisc devices, NICs, and networks adopt hardware-based QoS mechanism. Virtio-Fisc devices and NICs make use of their hardware queues of high and low priorities. We set DSCP values in the IP packet header through the networking library to leverage the priority queues of network switches.



(a) Client's CPU utilization. (b) Client's memory consumption.

(c) Client's CPU utilization. (d) Client's memory consumption.

Figure 7: The resource consumption of LC and TC when writing/reading data to/from the storage cluster.

Software-based QoS. Fisc client, Fisc agent and Fisc proxy adopt a software-based QoS mechanism, utilizing a hybrid thread model of exclusive threads for high-priority and low-priority requests, respectively. The reason for the hybrid thread model is to avoid head-of-line (HOL) blocking problems. To conserve the scarce CPU resource of DPU, a large offline request is not divided into separate smaller ones to avoid the complex request combination and failure dealing. As a result, if high and low priority requests are in the same thread, there may be a HOL blocking problem between them. And the other reason is the lack of cache isolation capability of NIC like CAT for Intel CPU [34]. The buffer cache of NIC may be full-filled with low-priority packets if a DPDK-based polling thread [35] stops polling network packets. Therefore, if high and low priority network packets are processed in the same thread, the low-priority queue of NIC should keep polling. Otherwise, the buffer cache may be full-filled and eventually affect high priority traffic. However, the non-stop polling for low-priority requests in a thread makes it hard to guarantee the high-priority requests.

Besides Fisc modules, the backend Pangu also adopts software-based QoS for NVMe SSDs, as current SSDs lack of hardware-level QoS mechanism. Therefore, Fisc enables end-to-end priority classification.

5 Evaluations

We evaluate Fisc through extensive experiments in a testbed and demonstrate its performance in a production environment. We focus on the following measurements:

- The efficiency of Fisc lightweight client (§5.2).
- The performance of I/O requests in Fisc (§5.3).
- The availability of I/O requests in Fisc (§5.4).
- The impact of QoS scheme on multi-applications (§5.5).
- The effectiveness of load balancing in Fisc (§5.6).

5.1 Testbed Setup

Our testbed is a disaggregated cluster consisting of one computation server and a storage cluster of 43 commodity storage servers. The computation server is equipped with a DPU. The storage cluster is equipped with the Pangu storage system [6]. We use FIO [36] to generate different I/O workloads in the computation server and record the CPU and memory consumption of the client. The number of threads to issue I/O requests is denoted by $numjobs$. The number of inflight I/O is denoted by $iodepth$. For simplicity, we use njmd in figures to represent the workload of $numjobs = n$, $iodepth = m$.

5.2 Lightweight Client

We first compare the resource consumption of Fisc client, denoted as LC, with that of a traditional FS client, which integrates the storage-related protocols (*e.g.*, three replicas) and network-related stacks (*e.g.*, RPC and TCP/IP) and is denoted as TC.

Microbenchmark. To test the resource utilization of clients with different data sizes, we write data to the file system with a granularity of 4 KB and read the file with a granularity of 128 KB. Figure 7 shows that LC has substantially lower CPU utilization and memory consumption than TC for both write and read operations. For example, when writing data with 8j32d (*i.e.*, $numjobs = 8$ and $iodepth = 32$), LC and TC each consumes 0.3 and 0.46 CPU cores, respectively. In another experiment where we let one FIO job write data to the storage cluster at a fixed rate of 1.75 GB/s, LC consumes less CPU and memory resources than TC by 69% and 20%, respectively.

Production environment. We also evaluate Fisc in a production system, which consists of thousands of servers and provides Swift service, a distributed streaming service similar to Kafka [37]. Figure 8 shows the bandwidth and CPU and memory consumption of Swift in one month when writing data to the remote storage cluster. Swift initially uses TC and switches to LC on day 18. After the switch, LC maintains the same high bandwidth performance as TC does, but consumes 16% and 57% less CPU and memory, respectively, than TC. Specifically, when we only offload erasure coding to Fisc proxies, the CPU and memory consumption of containers is reduced by 9% and 40%, respectively.

These results in the testbed and production environment demonstrate the efficiency and efficacy of the Fisc lightweight client in supporting cloud-native applications with high performance while consuming substantially fewer resources.

5.3 Latency

To evaluate the latency of I/O requests in Fisc, we first compare Fisc with a network-gateway-based load balancing solution [38], denoted as LB. We then validate the effectiveness of locality-aware read in the testbed.

Microbenchmark. We first start different FIO tasks on the computation server and measure the end-to-end latency of I/O

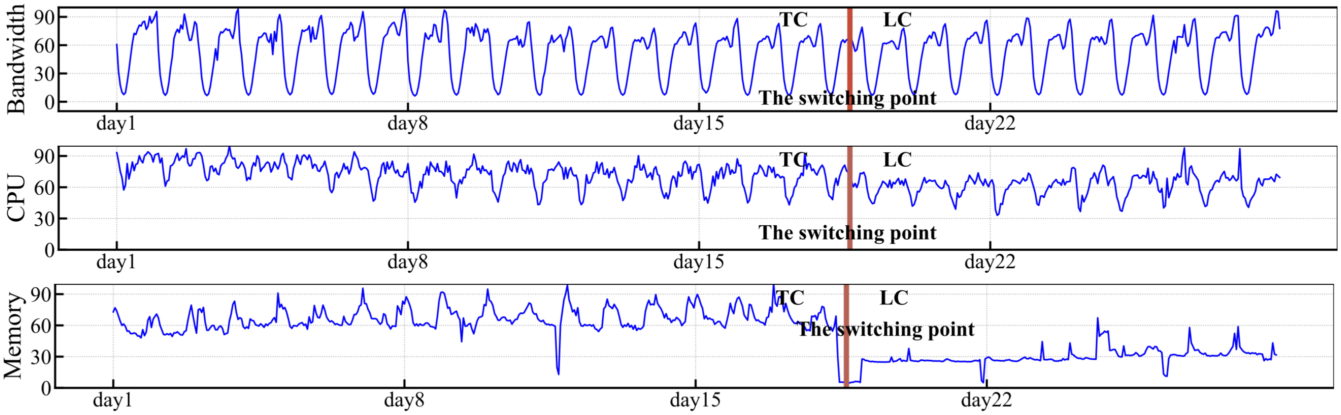
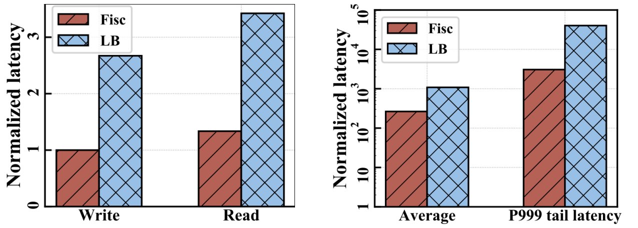


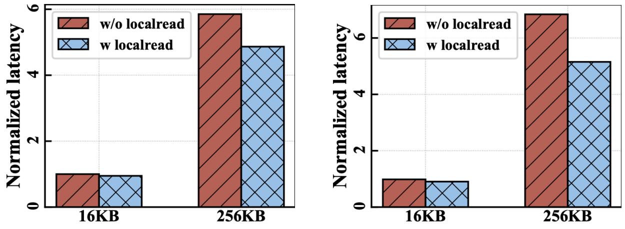
Figure 8: The bandwidth, CPU utilization, and memory consumption in a month. Results are given in the range of [0, 100].



(a) The write and read latency of a single job with a data size of 8 KB.

(b) The write latency with 64 jobs, a data size of 8 KB, and a fixed bandwidth of 100 MB/s.

Figure 9: The comparison of latency between Fisc and LB.



(a) The latency of sequential read.

(b) The latency of random read.

Figure 10: The effectivenss of locality-aware read.

requests. Figure 9(a) shows that the write and read latency of Fisc is 63% and 61% lower than those of LB when launching I/O requests with a data size of 8 KB. In the next experiment, we let Fisc and LB write files with 64 jobs and a data size of 8 KB at a fixed bandwidth of 100 MB/s. Figure 9(b) shows that Fisc reduces the average and P999 tail latency compared to LB by 76% and 92%, respectively. This latency improvement results from two optimizations: (1) the SaDGW provides one-hop communication instead of the two-hop communication in a centralized network gateway; (2) the SaDGW transparently adopts the high-performance networking stack to replace the inefficient TCP/IP stack.

To verify the benefits of locality-aware read, we further compare the latency of FIO tasks in sequential read and random read scenarios with a data size of 16 KB and 256 KB. As shown in Figure 10, the latency of read requests reduces in both scenarios (*e.g.*, by 25% when randomly reading files with a data size of 256 KB). It shows that the locality information in the range table is effective in helping route the read requests directly to the target storage server, reducing the end-to-end latency.

Production environment. We plot the average write latency of an online search workload over 30 days. As shown in Figure 11, the average latency is stable at ∼500 μs even when the workloads reach as high as millions-level IOPS. This result demonstrates that Fisc provides a low-latency file system service for cloud-native applications. In contrast, this latency becomes several milliseconds when the file system service is provided through LB.

5.4 Availability

We use the P999 tail latency as a key metric to measure the effectiveness of Fisc's storage-aware failure handling mechanisms in guaranteeing the availability of file system services.

Microbenchmark. To verify the impact of proxy failure on tail latency, we randomly kill some proxy processes in the storage cluster of 80 storage servers and record the tail latency for all I/O requests. As shown in Figure 13, we kill one proxy at t_1 and then kill five proxy processes at t_2. We observe that the tail latency increases to <40 ms for a short time and quickly returns to the previous level. This result shows that proxy failures in the storage cluster have a limited effect on the tail latency. It is because Fisc can retry the failed I/O requests with its storage-aware failure handling methods. As a result, such failures have a limited impact on applications.

Production environment. Figure 12 illustrates the P999 tail latency of online searching tasks over the same 30-day period. Most of the time, it stays under 30 ms. We analyze the spikes in the figure. The spikes t_1, t_2, t_3, and t_4 happen due to the upgrade of FS, at which we launch/stop some proxies in the storage cluster. Other spikes are caused by network jitters and storage node breakdowns. However, after each spike, the P999 tail latency quickly returns to a low level with the help of Fisc's storage-aware failure handling mechanisms.

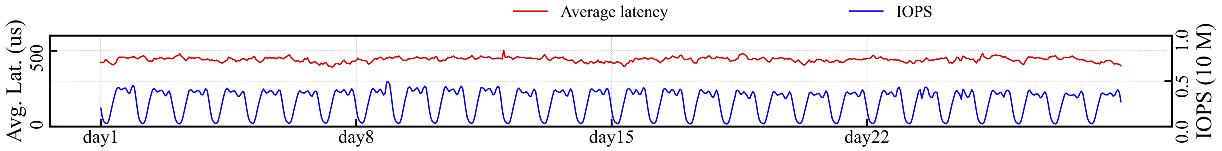


Figure 11: The average write latency and IOPS in one month.

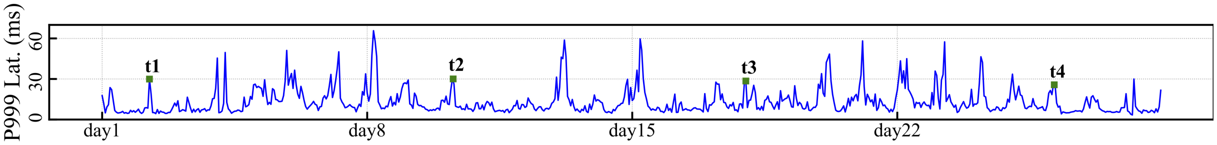


Figure 12: The P999 tail latency of write in one month in production environment.

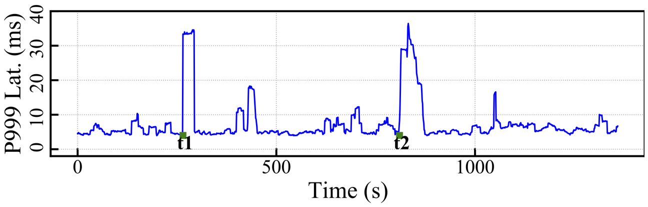


Figure 13: The P999 tail latency of write in micro-benchmark.

5.5 QoS

We demonstrate Fisc's ability to guarantee the QoS across different applications by measuring the latency of online searching tasks and the throughput of offline AI training tasks in a production environment. Both tasks are deployed in the same computation cluster and share the same storage cluster. Figure 14 shows that the latency of online search tasks stays stable and is barely affected by the fluctuated offline AI training tasks. It is because Fisc assigns a high priority to latency-critical tasks like online search and guarantees the corresponding QoS with an E2E QoS mechanism.

5.6 Load Balancing

To study the load-balancing capability of Fisc, We randomly choose six storage servers from our storage cluster and measure their normalized read IOPS over seven days. We compute the coefficient of variation of these nodes as a measurement of Fisc's load-balancing capability [39]. As shown in Figure 15, the coefficient of variation of read IOPS is < 5%. This result indicates that the read requests are evenly distributed among Fisc proxies and proves that Fisc achieves a similar quality of load balancing as Maglev, Google's in-house load balancer [39], whose coefficient of variation is 6-7%. This efficacy is due to Fisc's fine-grained storage-aware load-balancing strategy. Specifically, Fisc agents forward I/O requests with a granularity of files. In contrast, network-based load balancing methods forward I/O requests with a granularity of network connections, causing unbalanced numbers of files forwarded to different storage nodes.

6 Discussion

Not just migration. The two-layer aggregation of Fisc offloads network-related functions and storage protocols to Fisc agent in DPU and Fisc proxy in the storage node. The question is whether Fisc merely transfers the resource consumption from containers of users to DPUs and back-end storage clusters of CSPs but does not reduce the total amount of consumed resources. The answer is that Fisc not just migrates resources spatially but can significantly reduce resource consumption, because it *"aggregates"* the resource for storage protocols and network stacks processing in terms of tenants, applications and workloads. For example, one application in containers usually pre-allocate I/O threads and reserve memory and network connections, which cannot be shared with the applications in other containers. However, in Fisc, these resources are *"migrated"* and *"aggregated"* in Fisc agents and proxies, and they are efficiently shared by multiple applications to achieve high resource utilization. Furthermore, with function offloading, Fisc can leverage modern hardware-assisted acceleration for these storage protocols and advanced network-related stacks to improve their efficiency. For example, the Erasure-coding and CRC operation can be accelerated by hardware in the storage cluster.

With the development of cloud-native applications, more cloud-native services should aggregate their service-related resources among containers. Based on the traditional aggregation of VM resources, it will further improve the resource efficiency of CSPs.

Ecosystem service. The ecosystem is vital for cloud-native applications. Fisc extends its ecosystem in two aspects: compatibility with HDFS ecosystem and virtio-Fisc devices for different operating systems. For the former one, Fisc Client adopts a Java Native Interface (JNI) method to use its lightweight client of C language, and many optimizations have been introduced for the semantics compatibility between HDFS and Pangu. For the latter issue, we abstract virtio-Fisc devices to more general virtio-RPC devices, which are suitable for more cloud-native services. And we develop the virtio-Fisc driver in our released OS and will submit it to the open source community.

Resource in DPU. The resources in DPU are scarce, and Fisc also has to share these resources with other virtualization services, such as virtual networking and block services. Therefore, Fisc adopts a variety of optimization technologies to economize resource utilization, as mentioned in §4.3.3. With the development of DPUs such as Intel IPU [40] and

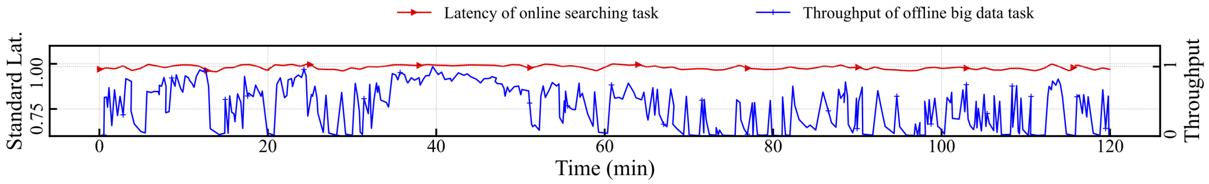


Figure 14: The latency of online tasks with background offline tasks in one month.

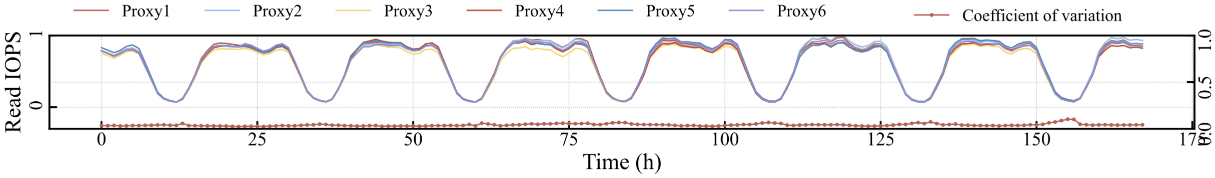


Figure 15: The load distribution of read IOPS of six storage nodes in one week.

Nvidia DPU [41], the processing capability of embedded processors of DPU has been greatly improved. Meanwhile, more hardware acceleration functions like compression have been integrated. These new features help Fisc agents adopt more complex policies to deal with failure handling and locality-aware read mechanisms. It is noteworthy that careful resource optimizations are still needed with the increase of throughput from 25 Gbps to 100 Gbps or 200 Gbps.

7 Related Work

Infrastructure support for cloud-native applications. Many studies [4, 42–51] have investigated how to provide efficient infrastructure support for emerging cloud-native applications (*e.g.*, microservice, container and serverless computing), including state management [43–45], runtime [46], data storage [47], fault tolerance [4] and performance optimization [48–51]. Some work [52,53] also looked into designing efficient service interfaces for cloud-native applications. For example, LogBook [52] provides logging interfaces for stateful serverless applications and uses a metalog to address log ordering, read consistency, and fault tolerance. Fluid [53] provides a unified data abstraction for cloud-native deep learning applications. In this paper, we design Fisc, a large-scale file system that provides high-performance file system services for cloud-native applications.

High-performance distributed file systems. Many distributed file systems have been designed and deployed (*e.g.*, pNFS [54], NAS [15], Facebook Tectonic [7], Google Colossos [5], and Alibaba Pangu [6]) to provide high-performance storage services for applications. However, they are ill-suited for cloud-native applications because they use heavyweight clients and a centralized network gateway. To this end, some studies (*e.g.*, OFC [55], FaaSCache [56], FLASHCUBE [57], and Pocket [58]) proposed adding cache to the persistence layer to improve the performance. However, they still suffer from a low level of resource multiplexing. In contrast, Fisc proposes the design of a lightweight client and storage-aware gateway, and resorts to a software-hardware co-design to provide high-performance file system services for cloud-native applications.

Bare-metal DPUs in clouds. The cloud computing community is increasingly developing and deploying bare-metal DPUs in clouds (*e.g.*, Nitro [19], BM-Hive [20], ELI [59], Splinter [60], and Bluebird [61]). Some studies also use DPUs to accelerate file system services (*e.g.*, LineFS [62], Gimbal [63], and Leapio [64]). However, they are not designed to provide cross-domain file system services between tenants and CSPs. In contrast, Fisc leverages the X-Dragon DPU in the computation server and introduces a new virtio device to provide secure, high-performance cross-domain file system services.

8 Conclusion

The trend of cloud-native brings new challenges and opportunities for CSPs to revisit their file system services. In this paper, we present Fisc, a large-scale cloud-native-oriented file system, which adopts a two-layer aggregation mechanism to multiplex resources of file clients among containers and a distributed storage-aware gateway to improve performance, availability and load balance of I/O requests. Fisc also adopts virtio-Fisc device with DPU for high performance and secure passthrough from users' virtual domain to CSPs' physical domain. Fisc has been deployed in a production DCN for over three years and provides large-scale file system service for cloud-native applications.

Acknowledgements

We are extremely grateful to our shepherd, Liuba Shrira, and the anonymous FAST'23 reviewers for their invaluable feedback. We also thank Yuxin Wang, Ridi Wen, and Haohao Song for their help during the preparation of the paper. Lulu Chen and Jie Wu are supported in part by the National Key R&D Program of China 2021YFC3300600 and an Alibaba Innovative Research Award. Qiao Xiang is supported in part by an Alibaba Innovative Research Award, NSFC Award 62172345, Open Research Projects of Zhejiang Lab 2022QA0AB05, and NSF-Fujian-China 2022J01004. Xiaoliang Wang is supported by NSFC Award 62172204.

References

[1] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *NSDI'19*, pages 193–206. USENIX Association, 2019.

[2] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards Demystifying Serverless Machine Learning Training. In *SIGMOD'21*, pages 857–871. ACM, 2021.

[3] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service Workflows. In *ATC'21*, pages 957–971. USENIX Association, 2021.

[4] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-Tolerant and Transactional Stateful Serverless Workflows. In *OSDI'20*, pages 1187–1204. USENIX Association, 2020.

[5] Google. A peek into Google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system, 2022.

[6] Pangu. The High Performance Distributed File System by Alibaba Cloud. https://www.alibabacloud.com/blog/, 2022.

[7] 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 *FAST'21*, pages 217–231. USENIX Association, 2021.

[8] Amazon. AWS Elastic File System. https://docs.aws.amazon.com/efs/latest/ug/whatisefs.html, 2022.

[9] Microsoft. Azure Data Lake Storage Gen2. https://learn.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-introduction, 2022.

[10] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *SIGCOMM'22*, pages 753–766. ACM, 2022.

[11] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Wang, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, Huayong Wang, Shanyang Liu, Lulu Chen, Zhiwu Wu, Haonan Qiu, Derui Liu, Gexiao Tian, Chao Han, Shaozong Liu, Yaohui Wu, Zicheng Luo, Yuchao Shao, Junping Wu, Zheng Cao, Zhongjie Wu, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. Deployed System: More Than Capacity, Performance-oriented Evolution of Pangu in Alibaba. In *FAST'23*. USENIX Association, 2023.

[12] Hdfs. Hadoop HDFS. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2022.

[13] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When Cloud Storage Meets RDMA. In *NSDI'21*, pages 519–533. USENIX Association, 2021.

[14] Alibaba cloud. Object Storage Service. https://www.alibabacloud.com/help/en/object-storage-service, 2022.

[15] Alibaba cloud. Apsara File Storage NAS. https://www.aliyun.com/product/nas, 2022.

[16] Alibaba. Maxcompute. https://www.alibabacloud.com/product/maxcompute, 2022.

[17] Alibaba cloud. The exploration of cloud-native. https://developer.aliyun.com/article/721889, 2021.

[18] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *ATC'22*, pages 53–68. USENIX Association, 2022.

[19] Amazon. AWS nitro system. https://aws.amazon.com/cn/ec2/nitro/, 2022.

[20] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. High-density Multi-tenant Bare-metal Cloud. In *ASPLOS'20*, pages 483–495. ACM, 2020.

[21] Xiantao Zhang, Xiao Zheng, and Justin Song. High-density Multi-tenant Bare-metal Cloud with Memory Expansion SoC and Power Management. In *HotChips'20*, pages 1–18. IEEE, 2020.

[22] Yuyu Luo, Chengliang Chai, Xuedi Qin, Nan Tang, and Guoliang Li. Visclean: Interactive cleaning for progressive visualization. *Proceedings of the VLDB Endowment*, 13(12):2821–2824, 2020.

[23] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Bluesky: A Cloud-backed File System for the Enterprise. In *FAST'12*, pages 1–14. USENIX Association, 2012.

[24] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Transactions on Storage*, 9(4):1–33, 2013.

[25] Yilong Li, Seo Jin Park, and John Ousterhout. MilliSort and MilliQuery:Large-Scale Data-Intensive Computing in Milliseconds. In *NSDI'21*, pages 593–611. USENIX Association, 2021.

[26] Amazon. AWS storage gateway. https://aws.amazon.com/cn/blogs/storage/deploy-a-highly-available-aws-storage-gateway-on-a-vmware-vsphere-cluster/, 2022.

[27] Digitalocean. DigitalOcean. https://www.digitalocean.com/community/tutorials/how-to-create-a-high-availability-setup-with-heartbeat-and-reserved-ips-on-ubuntu-14-04, 2022.

[28] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A Cloud-optimized Transport Protocol for Elastic and Scalable HPC. *Micro*, 40(6):67–73, 2020.

[29] Apache. HDFS APIs. https://github.com/apache/hadoop/blob/trunk/hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/fs/FileSystem.java, 2022.

[30] Google. Protocol Buffers. https://developers.google.com/protocol-buffers, 2022.

[31] Svilen Nikolaev Kanev. *Efficiency in Warehouse-scale Computers: A Datacenter Tax Study*. PhD thesis, Harvard University, pages 1–24, 2017.

[32] John Biddiscombe, Anton Bikineev, Thomas Heller, and Hartmut Kaiser. Zero Copy Serialization Using RMA in the HPX Distributed Task-based Runtime. In *Proceedings of the International Conference on WWW/Internet 2017 and Applied Computing*, pages 1–8. IADIS, 2017.

[33] Shuangchen Li, Dimin Niu, Yuhao Wang, Wei Han, Zhe Zhang, Tianchan Guan, Yijin Guan, Heng Liu, Linyong Huang, Zhaoyang Du, et al. Hyperscale FPGA-as-a-Service Architecture for Large-scale Distributed Graph Neural Network. In *ISCA'22*, pages 946–961. IEEE, 2022.

[34] Intel. Introduction to Cache Allocation Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, 2022.

[35] DPDK. DPDK Poll Mode Driver. https://doc.dpdk.org/guides/prog_guide/poll_mode_drv.html, 2022.

[36] Flexible i/o tester. Flexible I/O tester. https://fio.readthedocs.io/en/latest/, 2022.

[37] Apache. Kafka. https://kafka.apache.org/intro, 2022.

[38] What is ALB. Server Load Balancer. https://www.alibabacloud.com/help/en/server-load-balancer/latest/what-is-application-load-balancer, 2022.

[39] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI'16*, pages 523–535. USENIX Association, 2016.

[40] Intel. Intel® Infrastructure Processing Unit (Intel® IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html, 2022.

[41] Nvidia. NVIDIA BlueField Data Processing Units). https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2022.

[42] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *ATC'19*, pages 475–488. USENIX Association, 2019.

[43] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help Rather than Recycle: Alleviating Cold Startup in Serverless Computing through Inter-Function Container Sharing. In *ATC'22*, pages 69–84. USENIX Association, 2022.

[44] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-As-A-Service. *arXiv preprint arXiv:2001.04592*, pages 1–15, 2020.

[45] Zhe Wang, Teng Ma, Linghe Kong, Zhenzao Wen, Jingxuan Li, Zhuo Song, Yang Lu, Guihai Chen, and Wei Cao. Zero Overhead Monitoring for Cloud-native Infrastructure using RDMA. In *ATC'22*, pages 639–654. USENIX Association, 2022.

[46] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless Computing on Heterogeneous Computers. In *ASPLOS'22*, pages 797–813. ACM, 2022.

[47] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. FaaSFlow: Enable Efficient Workflow Execution for Function-as-a-Service. In *ASPLOS'22*, page 782–796. ACM, 2022.

[48] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *OSDI'22*, pages 303–320. USENIX Association, 2022.

[49] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: A Native Serverless System for Low-latency, High-Throughput Inference. In *ASPLOS'22*, pages 768–781. ACM, 2022.

[50] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Ice-Breaker: Warming Serverless Functions Better with Heterogeneity. In *ASPLOS'22*, page 753–767. ACM, 2022.

[51] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *NSDI'21*, pages 653–669. USENIX Association, 2021.

[52] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *SOSP'21*, pages 691–707. ACM, 2021.

[53] Rong Gu, Kai Zhang, Zhihao Xu, Yang Che, Bin Fan, Haojun Hou, Haipeng Dai, Li Yi, Yu Ding, Guihai Chen, et al. Fluid: Dataset Abstraction and Elastic Acceleration for Cloud-native Deep Learning Training Jobs. In *ICDE'22*, pages 2182–2195. IEEE, 2022.

[54] Dave Hitz, James Lau, and Michael A Malcolm. File System Design for an NFS File Server Appliance. In *WTEC'94*, pages 1–23. USENIX Association, 1994.

[55] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. OFC: An Opportunistic Caching System for FaaS Platforms. In *EuroSys'21*, pages 228–244. ACM, 2021.

[56] Alexander Fuerst and Prateek Sharma. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *ASPLOS'21*, pages 386–400. ACM, 2021.

[57] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes. In *PLOS'21*, pages 38–45. ACM, 2021.

[58] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *OSDI'18*, pages 427–444. USENIX Association, 2018.

[59] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-Metal Performance for I/O Virtualization. *SIGPLAN*, 47(4):411–422, 2012.

[60] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal Extensions for Multi-tenant Low-latency Storage. In *OSDI'18*, pages 627–643. USENIX Association, 2018.

[61] Manikandan Arumugam, Deepak Bansal, Navdeep Bhatia, James Boerner, Simon Capper, Changhoon Kim, Sarah McClure, Neeraj Motwani, Ranga Narasimhan, Urvish Panchal, Tommaso Pimpo, Ariff Premji, Pranjal Shrivastava, and Rishabh Tewari. Bluebird: High-performance SDN for Bare-metal Cloud Services. In *NSDI'22*, pages 355–370. USENIX Association, 2022.

[62] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient Smart-NIC Offload of a Distributed File System with Pipeline Parallelism. In *SOSP'21*, pages 756–771. ACM, 2021.

[63] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: Enabling Multi-tenant Storage Disaggregation on SmartNIC JBOFs. In *SIGCOMM'21*, pages 106–122. ACM, 2021.

[64] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. Leapio: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *ASPLOS'20*, pages 591–605. ACM, 2020.

TENET: Memory Safe and Fault Tolerant Persistent Transactional Memory

R. Madhava Krishnan Diyu Zhou[*] Wook-Hee Kim[†] Sudarsun Kannan[‡] Sanidhya Kashyap[*] Changwoo Min
Virginia Tech *EPFL*[*] *Konkuk University*[†] *Rutgers University*[‡]

Abstract

Byte-addressable non-volatile memory (NVM) allows programs to directly access storage using memory interface without going through the expensive conventional storage stack. However, direct access to NVM makes the NVM data vulnerable to software bugs and hardware errors. This issue is critical because, unlike DRAM, corrupted data can persist forever, even after the system restart. Albeit the plethora of research on NVM programs and systems, there is little focus on protecting NVM data from software bugs and hardware errors.

In this paper, we propose TENET, a new NVM programming framework, which guarantees memory safety and fault tolerance to protect NVM data against software bugs and hardware errors. TENET provides the popular persistent transactional memory (PTM) programming model. TENET leverages the concurrency guarantees (*i.e.*, ACID properties) of PTM to provide performant and cost-efficient memory safety and fault tolerance. Our evaluations show that TENET offers an enhanced protection scope at a modest performance overhead and storage cost as compared to other PTMs with partial or no memory safety and fault tolerance support.

1 Introduction

Byte-addressable non-volatile memory (NVM) opens a new paradigm in designing storage stack. NVM provides byte-addressability and low-access latency like DRAM and it offers data persistence like storage. A program can directly map (`mmap`) an NVM region to its address space and access it using `load`/`store` instructions without storage stack overhead (referred to as *direct persistence*). Several works leverage NVM in the core storage stack, including file systems [34, 51, 88, 89, 96], key-value stores [52, 55, 57, 61, 65, 66, 86], and persistent transactional memory (PTM) [47, 56, 77, 87]. Although the first commercial NVM product, Intel Optane DCPMM, was discontinued recently [13],industry continues to explore various forms of direct persistence [42]. In particular, the emerging Compute Express Link (CXL) [12, 31] opens new opportunities for byte-level persistence based on NAND flash [16, 21], NRAM [39], battery-backed DRAM [41, 78], and PRAM [22]. Also, many software-based solutions [53, 67, 91], which exploit direct persistence (DRAM along with in-rack battery), are being widely deployed in data centers [1, 11, 18, 45, 50].

However, the direct persistence of NVM opens several challenges in protecting data from software bugs (*e.g.*, "memory scribbles") and media errors. NVM data can be permanently corrupted due to a single memory scribble, which roots from a spatial safety violation (*e.g.*, buffer overflow) or a temporal safety violation (*e.g.*, use-after-free) in a program. Previous studies [26, 32, 35, 36, 59, 69–71, 73, 79, 81, 83, 84, 92, 95] have shown that such memory safety violations are prevalent in programs (*e.g.*, 70% of CVEs [5, 17, 25]). Since NVM is mapped to the same address space as DRAM, memory safety violations in NVM and DRAM can corrupt NVM data. Besides these software bugs, dense NVMs have a higher random raw bit error rate (RBER) than DRAMs, with RBER closer to NAND flash [85, 93]. Hence, NVM (*e.g.*, Intel Optane) adopts stronger ECC for error correction. Unfortunately, certain hardware errors can still escape the error correction, leading to Uncorrectable Media Errors (UME) in NVM [4, 7].

PTMs [47, 56, 77, 87] are one of the most popular NVM programming models because of their ability to exploit direct persistence. A few recent PTM systems, such as SafePM [27] and Pangolin [94], attempt to provide NVM data protection by extending `libpmemobj` [47]. A desirable PTM system that offers NVM data protection should (1) offer extensive data protection: protect against both NVM media errors and software memory safety violations in both DRAM and NVM, and (2) incur lower performance overhead and storage costs.

Unfortunately, existing works fail to meet the above criteria. SafePM [27] provides NVM memory safety by instrumenting every NVM access. It does not protect against media errors and memory safety violations in DRAM. The memory instrumentation and the associated metadata incur high performance overhead and storage cost. Pangolin offers data protection with checksum and parity while `libpmemobj` provides fault tolerance by simply replicating the NVM data to a backup NVM region. However, both systems are still vulnerable to memory safety violations, incur high NVM storage cost, and suffer from high performance overhead. As further explained in §2.2, in summary, prior approaches compromise the protection coverage [27, 44, 94] while also incurring high storage cost and high performance overhead [27, 47, 94].

This paper proposes TENET, a principled PTM-based approach that offers an enhanced memory safety and fault tolerance guarantees at a significantly lower performance overhead and storage costs than prior works. Leveraging off-the-shelf hardware features and the concurrency properties of PTM, TENET reduces performance overhead and storage costs without compromising its protection coverage. We realize TENET's memory-safe design principles using the state-of-the-art and highly scalable PTM framework, TimeStone [56] that does not provide NVM data protection. In particular, key techniques of TENET are as follows:

- **Hardware-enforced memory domain separation.** Instead of instrumenting every memory access to check for memory safety violations, TENET exploits an existing hardware feature: Intel Memory Protection Keys (MPK) [49, 74], to separate the address space into NVM domains and a DRAM domain. Only the trustworthy Tenet library can write to the NVM domains. Thus, outside the Tenet library, Tenet offloads NVM data protection against memory scribbles to hardware. This enables data protection for most memory access with almost zero overhead.

- **On-first-read and on-commit memory safety enforcement.** Enforcing memory safety at every NVM access in the Tenet library incurs high overhead. Instead, leveraging PTM semantics, Tenet enforces the temporal safety violation only at the first reference of an NVM object and the spatial safety violation only at the commit of a persistent transaction. This, in tandem with the memory domain separation technique, prevents the corrupted data from reaching NVM with very low runtime overhead.
- **Asynchronous hybrid NVM-SSD replication.** Protecting against NVM media errors fundamentally requires creating redundancy. Tenet asynchronously replicates the NVM data to SSD off the critical path to tolerate any number of NVM media errors. It thus offers low storage cost fault tolerance without hindering performance.
- We design Tenet using the above approaches, which to the best of our knowledge is the first high-performance PTM with memory safety and fault tolerance guarantees.
- We evaluate two different versions of Tenet– (1) memory safety only (Tenet-MS) and (2) memory safety and fault tolerance (Tenet) with key data structures and real-world workloads. Our results indicate that Tenet offers enhanced protection at a modest performance overhead and storage cost as compared to state-of-the-art systems.

2 Background and Motivation

This section first introduces NVM media errors (§2.1) and memory safety violation in NVM programs (§2.2), followed by discussing the prior PTM works that address the media errors and memory safety violations (§2.3).

2.1 NVM Media Errors

Figure 1 shows the classification of potential errors in NVM. These errors can be classified into hardware errors and software errors. Hardware errors can be further classified into media errors (MEs) and silent data corruptions (SDCs). Media errors are caused by faults in the NVM media such as exceeding the write endurance, power spikes, soft media faults etc that directly corrupt data in the NVM media [85, 93]. SDCs are caused by faults that occur outside NVM media, which indirectly causes data corruption. Examples of SDCs are buggy NVM firmware, faults in CPUs, memory controllers, or other hardware components [28, 54]. Handling SDCs is a separate research area and it is out of scope of this paper.

Hardware media error (ME) correction. Commercially available NVMs implement error-correction code (ECC) in hardware to detect and correct media errors. For example, Intel Optane DCPMM uses hardware parity to detect any-bit errors, and it can correct up to two 2-bit errors [10]. The NVM hardware transparently fixes such correctable media errors (CMEs). However, uncorrectable media errors (UMEs) will be reported for software intervention as detailed below.

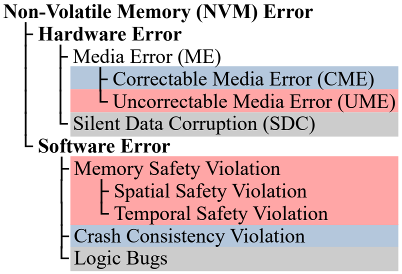

Non-Volatile Memory (NVM) Error
Hardware Error
Media Error (ME)
Correctable Media Error (CME)
Uncorrectable Media Error (UME)
Silent Data Corruption (SDC)
Software Error
Memory Safety Violation
Spatial Safety Violation
Temporal Safety Violation
Crash Consistency Violation
Logic Bugs

Figure 1: Classification of errors in NVM. Tenet handles *UME*, *Spatial and Temporal Safety Violation* bugs (red). Tenet relies on the hardware ECC to fix *CME* and the underlying PTM to handle *Crash Consistency Violations* such as atomicity and persistence ordering (blue). *Silent Data Corruption* in the hardware (*e.g.*, CPU faults) and *logical bugs* in the application are out of scope (grey).

Reporting uncorrectable media error (UME) to software. The OS receives the reports of UMEs; and it can pass it to the application. Specifically, when a CPU accesses an NVM page affected by UMEs, the NVM hardware sends a poison bit along with the relevant data to the CPU. Upon encountering the poison bit, the CPU raises a memory check exception (MCEs) for the OS to handle. Currently, Linux handles the MCE by adding the corrupted page to the bad block list and sends a `SIGBUS` signal to the application [4, 9]. Then the OS leaves the responsibility to the application for fixing UMEs during the recovery phase [7]. *We note that, although the NVM is byte-addressable, UMEs are reported to the software at the page granularity due to the blast radius effect [3].*

2.2 Memory Safety in NVM Programs

We categorize software "scribbles", which corrupt NVM data, as spatial and temporal memory safety violations (Figure 1). *Spatial safety violations* happen when memory is accessed beyond its allocated range. Buffer overflows and array out-of-bound accesses are classical examples. *Temporal safety violations* happen due to dangling pointers; *i.e.*, when accessing an already freed (*use-after-free*) or accessing a reallocated address range (*use-after-realloc*). These memory safety bugs are even more dangerous in NVM than DRAM because the NVM data will be corrupted forever and a simple system restart would not fix these issues. *Note that memory safety bugs on either DRAM or NVM region of an application can cause NVM data corruption since the NVM region is mapped directly to application's address space*.

2.3 Prior NVM Data Protection Approaches

Memory safety in NVM programs. Prior works – Pangolin [94], SafePM [27], and Corundum [44] – include mechanisms to protect NVM data from memory safety violations. Pangolin extends `libpmemobj` [47] and uses per-object checksum to detect spatial safety violations. SafePM adds Address-

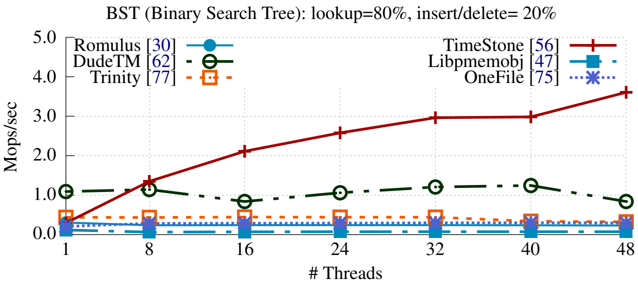


Figure 2: Performance of TimeStone against other PTMs. None of the PTMs are memory safe or fault tolerant against UME.

Sanitizer [80] to libpmemobj transaction to detect spatial and temporal safety violations on NVM data. Corundum is a Rust-based NVM programming library and leverages Rust's type system to statically enforce spatial and temporal memory safety. However, they have some critical limitations. *First, none of these approaches prevent NVM data corruption due to memory safety violations on "DRAM data"*. Suppose that the buggy code inside a transaction causes a buffer overflow on DRAM data; such spatial safety violations on DRAM can scribble arbitrary memory location, including NVM data. *Moreover, none of them guarantee to protect NVM data from temporal safety violations.* Pangolin does not check temporal safety violations. Meanwhile, SafePM does not detect use-after-realloc bugs. Even with Corundum, the developers still have the responsibility to guarantee type and memory safety for the "unsafe" Rust code, both can result in spatial and temporal safety violations.

Both Pangolin and SafePM suffer from high performance overhead and introduce additional performance bottlenecks. Pangolin calculates and verifies checksums on the critical path, imposing high performance overhead. Furthermore, it verifies checksum only for write transactions (*i.e.*, read transactions are unprotected). SafePM instruments every NVM access to check for memory safety violation, which is costly. SafePM further introduces extra UNDO logging overhead over the already existing expensive logging in the libpmemobj to guarantee crash consistency for its memory safe metadata.

Fault tolerance against UME. To protect against UME, libpmemobj supports replicating data on NVM. However, it replicates data on the write critical path, leading to high performance overhead. Furthermore, storing the replicated data on NVM wastes the precious NVM space, doubling (2×) storage cost. Pangolin uses parity for fault tolerance; however, *parity calculation on the critical path causes high performance overhead and it unnecessarily serializes the transactions which affects the write scalability*. Further, Pangolin can recover up to one page within a parity region; a data loss will happen if UME occurs on more than a page. SafePM and Corundum do not provide any fault tolerance against UME.

2.4 Prior PTMs for NVM

Libpmemobj [47] has been the de-facto PTM. However, it suffers from high performance overhead and poor scalability. Thus, several new PTMs focus on addressing its limitations [30,40,62,68,75–77,87]. Figure 2 shows that none of the existing PTMs, except TimeStone [56], scale beyond 8 cores even for a read-intensive workload. Further, TimeStone performs up to 8× better than the existing PTMs. Based on this observation, we chose TimeStone [24,56] as the transaction abstraction for TENET. Moreover, designing memory safety and fault tolerance techniques for such a high performance PTM is challenging as even a small bottleneck can compromise its original scalability and performance. We introduce the relevant design aspects of TimeStone below.

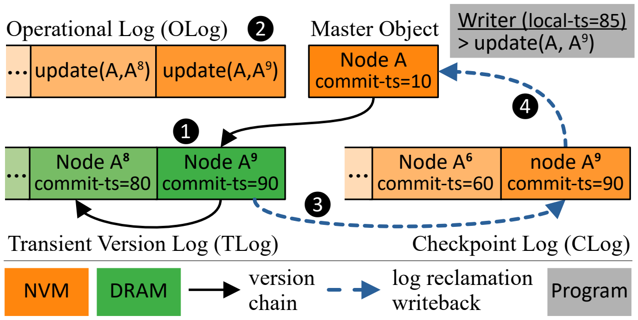


Figure 3: An illustrative example of updating Node A to its 9th version (A^9) in TimeStone.

Multi-version concurrency control. TimeStone follows multi-version concurrency control (MVCC). With MVCC, TimeStone supports non-blocking reads and concurrent disjoint writes, achieving high concurrency. For each object created by the application (*e.g.*, B-tree node), TimeStone allocates a *master object* on NVM (see Figure 3). On updating a master object, TimeStone creates a new version (❶) on DRAM, chaining multiple version objects from new to old object's age. TimeStone dereferences the right version object during the dereference phase with the help of timestamps. Each version object gets assigned a timestamp when it is committed (commit-ts). Also, each transaction gets a timestamp (local-ts), which denotes the transactions' start time. TimeStone traverses the version chain and chooses the most recent version of an object based on these timestamps (*i.e.*, commit-ts<=local-ts). This guarantees a consistent snapshot of NVM data for all transactions at any given time.

Operational log based immediate durability. TimeStone uses a DRAM-NVM hybrid logging technique, named TOC logging for efficient crash consistency. The TOC logging consists of Transient Version Log (TLog) on DRAM, Operational log (OLog) and Checkpoint log (CLog) on NVM, as illustrated in Figure 3. TimeStone creates a new version on TLog (❶), and logs the performed operation to the OLog (❷) for immediate durability. An operational log entry is typically much smaller than the conventional undo/redo logging, which duplicates the data, thus making crash consistency efficient.

Asynchronous log reclamation and replay based recovery. As more versions are created, TLog eventually becomes full, triggering log reclamation. When `TLog` is reclaimed, the latest version of an object on `TLog` ($\texttt{A}^9$ over $\texttt{A}^8$) is checkpointed to the `CLog` (❸). Similarly, when `CLog` becomes full, the latest checkpoint ($\texttt{A}^9$ over $\texttt{A}^6$) is written back to the master object (❹). To recover from a crash, TimeStone first applies the checkpoints in `CLog` to the respective master objects and reverts them to a consistent snapshot. Then the `OLog` is executed to recreate all the updates that are lost on `TLog`.

3 Overview of Tenet

3.1 Threat Model and Assumptions

Tenet aims to protect against spatial and temporal memory safety violations in buggy application code. Furthermore, Tenet considers the possibility of a memory safety violation on DRAM data corrupting NVM. Tenet also aims to guarantee fault tolerance for NVM data against the uncorrectable media errors (UMEs). PTMs in general and TimeStone in particular cannot guarantee **ACID** properties for the application code that is outside the transaction or when the PTMs' APIs are misapplied. This applies to Tenet as well, *i.e.*, it cannot guarantee memory safety and fault-tolerance for the code outside the transaction. Tenet is not designed to handle SDC that occur outside the NVM media. Protection against the adversarial attacks (*e.g.*, control-flow attacks) is out-of-scope. However, the protection techniques and mechanisms against SDC and control-flow attacks can be orthogonally deployed to Tenet. In Tenet, application code is distrusted while Tenet library code and OS kernel are considered as a trusted computing base (TCB).

3.2 Design Goals

- **Protect NVM data from memory safety violations.** Tenet should detect all spatial and temporal safety bugs not only from NVM but also from DRAM. Any memory safety bugs either in DRAM or NVM code should not corrupt NVM data.
- **Protect NVM data against UMEs.** Tenet should provide a robust fault tolerance mechanism to recover and restore NVM data from UMEs transparently.
- **Low performance and storage overhead.** Tenet aims to be a *practical* system that offers an enhanced protection scope and strong fault tolerance at a minimal performance and storage overhead.

3.3 Design Overview

Tenet re-purposes the multi-versioning and transactional semantics of TimeStone to achieve its design goals. Below we introduce Tenet's main techniques as illustrated in Figure 4.

(1) Separation of NVM protection domain from DRAM. A memory safety bug (*e.g.*, out-of-bound write) either in DRAM or NVM can result in NVM data corruption. Enforcing full memory safety in every single memory access incurs prohibitive runtime overhead as prior studies show [27, 94].

To prevent unauthorized NVM writes without checking every single memory access, Tenet grants the write permission to the NVM region only for the TCB *i.e.*, the Tenet library code. In other words, the application code has read-only permission for the NVM, and consequently, it only writes on DRAM. When the application commits its transaction, writer thread gets write permission to execute the Tenet library code which propagates the updates on DRAM to the NVM.

Tenet completely segregates DRAM and NVM regions so that all new version and master objects are created on DRAM (referred to as *DRAM Objects*). Therefore, Tenet application code does not require write access to NVM, as it writes only to the DRAM region. If a buggy application code tries to write to the NVM region, it will receive an exception (`SIGSEGV`) from Tenet and will be terminated. Tenet exploits Intel Memory Protection Keys (MPK) [49, 74] to efficiently switch NVM permissions for each thread.

(2) On-commit spatial safety enforcement. As applications can always write to the `TLog` (*i.e.*, DRAM), it is vulnerable to arbitrary memory scribble. A corrupted DRAM object can be eventually propagated to `CLog` (❻ in Figure 4) and the master object (❽), consequently corrupting the NVM data. We propose *on-commit spatial safety enforcement* to prevent corrupted DRAM objects from reaching NVM. Tenet adds eight byte canary values at the start and at the and of a DRAM object during its creation (❷). Specifically, Tenet assigns a random value to `C0`, and the hash of `C0` and its location (`xor(C0,&C1)`) to `C1`. When an application commits the transaction, Tenet inspects the integrity of canary values of all DRAM objects in that transaction (③ and ④). If the canaries are compromised (*i.e.*, `C0 != xor(C0,&C1)`), then Tenet aborts the transaction and gracefully terminates without propagating the corrupted objects to NVM.

Our on-commit spatial safety enforcement is efficient with minimal performance overhead. Unlike the prior approaches [27, 69, 79, 95], our technique avoids reading additional metadata, and it checks the integrity only once during the transaction commit. Note that NVM objects do not have canary values and thus no NVM space overhead.

(3) On-first-dereference temporal safety enforcement. Even after an NVM (master) object is freed (and then reallocated), a program still can reference it via dangling pointers which can corrupt the NVM data in unintended ways.

We propose *on-first-dereference temporal safety enforcement* to efficiently enforce temporal safety of NVM objects with a minimal runtime overhead. Tenet uses a tag-based approach, which essentially checks if a pointer points to the right object by comparing tags associated with the pointer and the pointed object. When Tenet creates an NVM (master) object, it assigns a 2-byte random integer as a tag of the object (*e.g.*, `0xCAFE` for `Node A` in Figure 4). We encode this 2-byte tag in the upper 16-bit of a pointer, which is unused in the x86 architecture. When the object is freed, its associated tag on the header set to zero for detecting use-after-free. When the object is dereferenced *first time* in a Tenet transaction (①), Tenet checks whether the encoded tag in the pointer matches

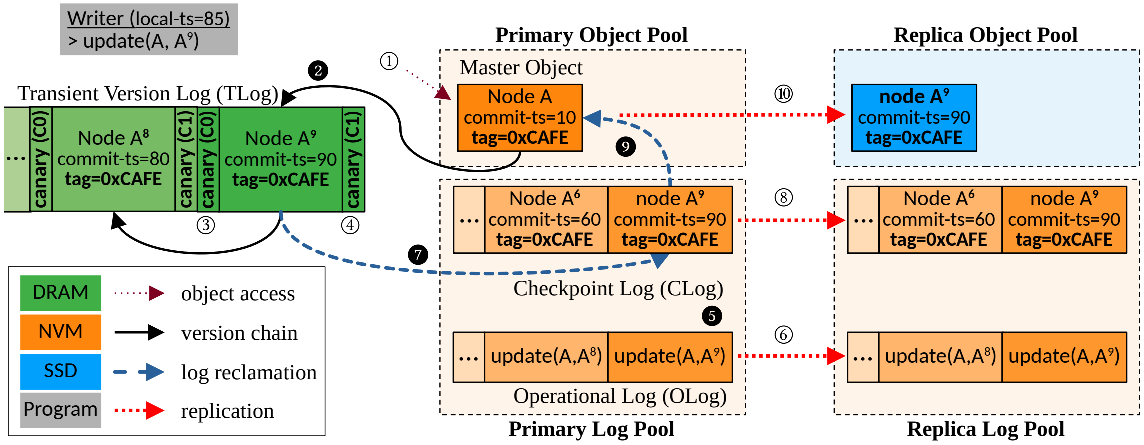


Figure 4: Overall architecture of TENET with an example of updating Node A to its 9th version (A^9). ⓝ denotes the newly added memory safety checks and replication to the TimeStone transaction. Note that the application has read/write access to DRAM and read-only permission for NVM. When accessing Node A, TENET validates its temporal safety by comparing the tags, 0xCAFE (①). If the tags do not match, the transaction is aborted. Otherwise, the writer proceeds to traverse the Node A's version chain, makes a copy of the latest version (A^8) in its TLog and updates it to A^9 (❷). Upon commit, Node A^9 is validated for spatial safety by checking the canary values (③ and ④). The transaction is aborted if the validation fails. Otherwise, the writer commits the transaction by updating its OLog (❺) for durability and it also synchronously updates the replica OLog for fault tolerance (⑥). When reclaiming the TLog, Node A^9 is once again validated for spatial safety before checkpointing it to the CLog (❼) followed by synchronously updating the replica CLog (⑧). Similarly, when the CLog is full, TENET writes back the latest checkpoint (Node A^9) to the original master object Node A (❾). The updated Node A is then *asynchronously* replicated to the disk (⑩).

with the tag in the pointed object. If the tags do not match, it means the pointer points to the already-freed/-reallocated object (*i.e.*, dangling pointer), which violates temporal safety. In this case, TENET aborts the transaction immediately.

Our approach is efficient and imposes minimal performance overhead because it checks the temporal safety of each object only once in a transaction. Also, accessing the inlined tags is cache-friendly, which, unlike prior approaches [27,71,79,95], requires no additional metadata lookup.

(4) Off-critical path NVM replication to SSD. TENET replicates all NVM data; in the case of a UME, corrupted NVM pages can be restored using the replica. The main challenge in designing a replication scheme is minimizing the performance overhead and storage cost. While replication to another NVM region can be performance efficient, it incurs 2× higher capacity cost. Instead, we propose a hybrid NVM-SSD replication technique; TENET *asynchronously* replicates the master objects to SSD (⑩) and *synchronously* replicates the transaction logs (CLog and OLog) to NVM (⑥, ⑧). Master objects, are application data structures, which can be large and also potentially occupy the entire NVM space. Hence, TENET replicates master objects to the SSD off the critical path to reduce both storage cost and performance overhead. Although the replication is asynchronous, TENET guarantees *loss-less* NVM data recovery by prudently leveraging the transaction logs and grace period semantics. Meanwhile, transaction logs are small and finite, so TENET replicates them to NVM to reduce performance overhead. Further, TENET is also capable of recovering from multiple simultaneous UMEs occurring in one or multiple NVM pages. We explain this design, its correctness and recovery guarantees in §4.4 and §4.5.

3.4 Putting It All Together For TimeStone

TENET makes the NVM read-only for all except the TENET's library code. So the NVM objects in TimeStone do not need spatial safety checks as they are read-only objects. TENET enforces temporal safety checks for all NVM objects (using pointer tags) during the object dereferencing to detect dangling pointers. On the contrary, DRAM objects are vulnerable to application scribbles (due to write permission) hence TENET enforces on-commit spatial safety checks using the canary bits. DRAM objects do not need separate temporal safety checks as they are managed internally by TENET; *i.e.*, as DRAM objects are accessed via the respective NVM object, enforcing temporal safety for NVM objects indirectly guarantees it for DRAM objects. We discuss the correctness of these techniques in §4.3. TimeStone can not handle UMEs, so TENET proposes to replicate master objects and transaction logs to SSD and NVM respectively; in the event of a UME, NVM data can be restored using the NVM/SSD backup. In a nutshell, we optimally apply TENET's memory safety techniques to the vulnerable parts of TimeStone and organically redesigned it to guarantee full memory safety. If TENET was to be used for other PTMs, then its techniques can well be applied, albeit it may require some engineering effort. We discuss this further in §6. Refer to Figure 4 for a summary on lifecycle of a TENET transaction.

4 TENET Design

In this section, we first describe TENET transaction design (§4.1) followed by the design of memory safety (§4.2-§4.3), fault tolerance replication (§4.4), and recovery (§4.5).

4.1 TENET Transaction

Below we explain how TimeStone transaction is redesigned using TENET to enforce memory safety and fault tolerance.

4.1.1 NVM Object Dereference

Object dereferencing in TimeStone (§2.4) only traverses the version chain and returns the correct version, whereas in TENET, object dereferencing is a two-step process.

(1) Temporal safety validation. TENET validates the master object pointer for temporal safety (§4.3.2) to detect dangling pointers; transaction aborts if the validation fails (§4.1.4).

(2) Version chain traversal. If the object passes the validation, then TENET dereferences the correct DRAM object or directly the master object if the version chain does not exist.

4.1.2 Updating an Object

In TENET, a writer updates a master object by creating a new DRAM object as done in the TimeStone. However, TimeStone allows its users (application) to allocate and write to the NVM when creating new master objects. Thus, a buggy application can easily corrupt the NVM region. In TENET, this is restricted to prevent direct NVM writes; so the application allocates and writes to a new master object (*shadow master object*) on the DRAM and then during the commit phase TENET library creates a corresponding NVM copy only if the writes pass the spatial safety violation checks.

4.1.3 Committing a Transaction

In TimeStone, the commit procedure updates the `OLog` to guarantee durability and then makes all the updates atomically visible. TENET's commit procedure happens in three phases:

(1) Spatial safety validation. All the new versions and shadow master objects created in a transaction are validated for spatial safety violations (§4.3.1). Upon successful validation, TENET allocates and updates the persistent master object from the corresponding shadow master object.

(2) Transaction durability and replication. Updating `OLog` guarantees durability, and replicating it ensures fault tolerance (§4.4.1). Also, TENET adds all the newly created master objects in **(1)** to the replica buffer to trigger async disk writes using background workers (§4.4.2).

(3) Publishing the updates atomically. TENET makes the updates atomically visible by adding the new versions to their respective version chain, and this procedure is exactly the same as TimeStone. Additionally, TENET frees all the shadow master objects, if any, and exits the critical section.

4.1.4 Aborting a Transaction

Common abort procedure. TENET rolls back any used log space, lock status, and reclaims all the shadow master objects and also its NVM counterpart if one exists. This is common for all three abort cases described below.

Abort due to lock conflict. During the object update (§4.1.2), if the writer fails to acquire a lock, it aborts the transaction. This is a benign abort *i.e.*, no memory safety violations, so TENET performs the common abort procedure and retries the transaction after the backoff period.

Abort due to memory safety violation. All ongoing transactions are aborted if a transaction aborts due to spatial safety or temporal safety violation. TENET executes the common abort procedure and returns an exception.

Abort due to a UME. The OS notifies a UME by sending a `SIGBUS` signal. TENET's signal handler catches the signal, returns a UME exception to notify the application, and gracefully terminates the process. TENET fixes the affected NVM region during the recovery process (§4.5).

4.2 Unauthorized NVM Write Prevention

TENET already prevents application code from directly writing to the NVM by using DRAM objects for the updates. However, a buffer overflow on DRAM can corrupt the NVM data as NVM is directly mapped to the applications' address space. TENET employs Memory Protection Keys (MPK), a hardware feature available in the Intel systems [33, 34, 43, 74, 82] to detect NVM writes out of TENET library code.

Using MPK to enforce read-only NVM access. With MPK, a page can be assigned to one of the 16 available protection domains. The assigned protection domain is encoded in the page table entry. A thread's access permission to the protection domains is controlled at the per-thread level via a user-accessible register, `PKRU`. A thread can switch its access permissions to the protection domains by writing to the `PKRU` register, which only costs 20 CPU cycles. In TENET, each NVM pool is assigned a unique protection key during pool creation. Only the TCB (*i.e.*, TENET library code) is allowed to write to the NVM pool. Thus, a thread grants itself read-write permissions to the corresponding NVM pool during the library code execution and revokes it before exiting the library. As a result, if the application writes to NVM (*e.g.*, due to buffer overflow), MMU prevents the access and OS sends a `SIGSEGV` signal. Thus, any spatial safety violations due to a buggy write is contained within the DRAM region.

4.3 Enforcing Memory Safety

In this section, we explain the spatial (§4.3.1) and temporal safety design (§4.3.2). In §4.3.3, we explain the array interface as an example, and how the interface provides memory safety.

4.3.1 On-commit Spatial Safety Design

TENET enforces spatial safety for all DRAM objects to prevent NVM data corruption due to a buggy DRAM write.

Technique. As illustrated in the Figure 4, all DRAM objects are assigned two 8-byte canaries at the start `C0` and at the end `C1`. Specifically, `C0` is a random value and `C1` is the hash of `C0` and its location (`xor(C0,&C1)`). TENET inspects the integrity of canary bits to detect buffer overflows and underflows.

On-commit validation. When the application commits its writes (§4.1.3), TENET inspects canary bits for all the newly created DRAM objects. A transaction is committed only when both `C0` and `C1` are intact in all the DRAM objects. Otherwise, the transaction aborts and discards all the corrupted objects. An erroneous transaction can corrupt the DRAM objects outside of the current transaction *i.e.*, the ones that are part of other concurrent transactions or the ones that are not part of any ongoing transactions at all. To detect such cases, TENET places an 8-byte canary at the start and the end of the transactions' write set. Note that all the DRAM objects including the shadow master objects are part of a transactions' write set. TENET validates the write set canaries before and after each step of the commit process (§4.1.3). This ensures that a transactions' write set (*i.e.*, DRAM object) has not been corrupted by an erroneous concurrent transaction, particularly between the initial validation (**(1)** in §4.1.3) and the publication of the updates (**(3)** in §4.1.3). However, if the write set canaries are found to be compromised then TENET aborts all the transactions as explained in §4.1.4.

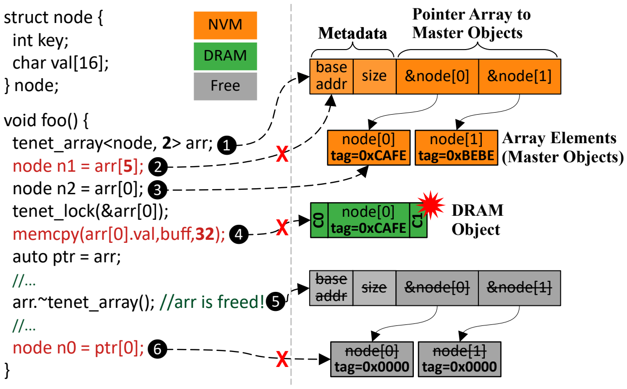
Correctness. Deferring spatial safety checks until the commit time does not violate the correctness as the other concurrent transactions *can not* observe any uncommitted DRAM objects. Although a rare case, to avoid reading a DRAM object that is corrupted (after it commits), TENET performs spatial safety check before dereferencing a committed DRAM object. Subsequently, the DRAM objects (❼ in Figure 4) and the shadow master objects are re-validated before and after copying to the NVM to prevent *Time-of-Check-Time-of-Use (TOCTOU)* bugs [6]. If an DRAM object is found to be corrupted post the copy operation then the corresponding NVM object will be safely reclaimed as part of the transaction abort procedure. Finally, TENET cannot detect the corruptions that occur without overwriting the canaries, aka intra-object overflows. We discuss this further in §6.3.

4.3.2 On-first-dereference Temporal Safety Design

TENET enforces temporal safety for all NVM (master) objects to detect dangling pointer dereference. Accessing an already free-ed (or reallocated) address can corrupt the NVM data due to use-after-free (or use-after-realloc) bugs.

Technique. To detect dangling pointers, TENET assigns an *unique 2-byte tag* for all the master objects, which is stored in the object's header (`0xCAFE` in Figure 4) at the time of its creation. A copy of this tag is also encoded in the *unused* upper 16-bits of the master objects' address. On deallocating the master object, the tag in the objects' header is set to zero.

On-first-dereference validation. When the application accesses a master object for *the first time in a transaction*, TENET validates the pointer to the master object before traversing the version chain (§4.1.1). TENET extracts the tag encoded in the master objects' pointer and compares it with the tag stored in the respective master objects' header. If they match, then it is a valid pointer. When an application accesses a master object with a dangling pointer, tag matching would fail; the tag in the header would either be zero (if the address is already freed) or different random value (if the free-ed address is reallocated). In that case, TENET cuts the version chain access and aborts the transaction.



```
struct node {
  int key;
  char val[16];
} node;

void foo() {
  tenet_array<node, 2> arr; ❶
  node n1 = arr[5]; ❷
  node n2 = arr[0]; ❸
  tenet_lock(&arr[0]);
  memcpy(arr[0].val,buff,32); ❹
  auto ptr = arr;
  //...
  arr.~tenet_array(); //arr is freed!❺
  //...
  node n0 = ptr[0]; ❻
}
```

Figure 5: Memory safety design for arrays. ❷ and ❹ are spatial safety violations due to out-of-bound read (detected by bounds checking) and write (detected using canaries), respectively. ❻ is temporal safety violation due to use-after-free (detected using pointer tags).

Correctness. Once a master object is successfully dereferenced, it can be safely used without any further temporal safety checking *within the transactions' lifetime*. This is because TENET (and TimeStone) uses an RCU-style, epoch-based garbage collection scheme so it never frees an object (and its versions) with live references from other transactions; *i.e.*, an object will be free-ed only when all the transaction that has live references exits. Also, a DRAM object can be dereferenced only via its NVM object and TENET cuts the version chain access upon detecting a dangling pointer, which indirectly guarantees temporal safety for DRAM objects.

4.3.3 Spatial and Temporal Safety for Array Objects

In TimeStone, an array is stored and accessed as a single pointer. Even if the application just reads/writes to one array element, TimeStone dereferences the entire array. Such a design is highly unsafe. For instance, once the entire array is dereferenced, a buggy application can read/write out-of-bounds resulting in an undetected corruption. This is a notoriously hard problem even in the DRAM world. To address this, we redesigned the array interface in TENET. *An array is internally represented as an array of pointers where each array index stores a pointer to its element*. With this design, TENET dereferences only the array index that the application intends to read/write. If the application accesses an index that is out-of-bound, TENET aborts the transaction.

Array interface. ❶ in Figure 5 presents the TENET's array interface. In TENET, each array element is a master object; and an array consists of pointers to these master objects along with the base address and size information. This representation is internal and the application accesses its array in the traditional *C semantics*. We do not present the pseudocode for our interface due to space limitations. Essentially, Tenet retains the C-style semantics by leveraging C++ operator overloading. `tenet_array` class overloads the necessary operators to hide the internal representation. For instance, the array access operator (`[]`) is overloaded to perform bounds checking, then access the master object at the index. Similarly, other operators (=, +, -, etc) are also appropriately overloaded to retain the programmability and to make the interface transparent. However, this representation requires additional memory to maintain pointers to the array elements. An `N` element array requires a space of `N*sizeof(N)`, whereas Tenet requires an additional `sizeof(void*)*N` space to maintain the pointers.

Memory safety validations. Figure 5 illustrates how Tenet enforces memory safety for arrays (`arr` with two elements). The canary-based spatial safety and the tag-based temporal safety apply to every array element. In addition, Tenet performs bounds checking for every array dereference using the base address and size metadata *i.e.*, `index > size` (❷). In ❹, a transaction writes to the `val` out-of-bounds, Tenet detects this violation by inspecting the corrupted canary bits in the commit phase. In ❻, transaction dereferences a dangling pointer (freed in ❺) and Tenet detects it by comparing the tags (`0xCAFE` ≠ `0x0000`) during the object dereference.

4.4 Enforcing fault tolerance Against UMEs

This section explains the synchronous log replication and the off-critical path master object replication design to guarantee fault tolerance against UMEs.

4.4.1 Transaction Log Replication

As illustrated in Figure 4, the primary log pool on the NVM consists of all the transaction logs (`OLog` and `CLog`). Tenet maintains a consistent backup of the primary log pool by *synchronously* replicating the logs on the critical path, *i.e.*, when an `OLog` or a `CLog` in the primary log pool is updated, the corresponding log in the replica log pool is also updated. Atomicity for primary and replica log writes is inherently guaranteed by the transactions' commit protocol (§4.1.3); *i.e.*, Tenet commits a transaction only when both the logs are updated. So, if a crash happens before updating the replica log, then the transaction is considered to be aborted and the partially written log entries are discarded during the recovery phase. Similarly, during log reclamation, the primary log is reclaimed first and the replica log is reclaimed up to the same point to maintain consistency. Tenet ensures that pages in the primary and the replica log pool do not overlap by maintaining two disjoint NVM pools for the primary and replica log pool. In this way, Tenet *can recover from multiple UMEs even if it spans across many pages within a log pool.*

Why replicate logs on the critical path? TimeStone buffers the updates to the master objects in the `OLog` and `CLog` to maximize the write coalescing. Hence, if the logs in the primary pool are corrupted, it may cause a significant amount of data loss during the recovery. As a result, Tenet replicates the primary log pool synchronously to ensure that there is always a consistent backup. Thus, Tenet can simply use the replica log pool to recover the NVM data *without losing any committed updates*. Tenet uses NVM to reduce the performance overhead as the replication is done in the critical path.

4.4.2 Off-critical Path NVM Replication to SSD

Tenet makes three critical design choices for a performant and cost-efficient NVM (master) objects replication: (1) objects are replicated to SSDs instead of NVM to reduce the storage cost overhead, (2) replication is performed out of the critical path to reduce the performance overhead (§4.4.3), and (3) Tenet uses grace period semantics to enforce NVM-SSD consistency to guarantee loss-less recovery (§4.4.4).

4.4.3 Off-critical Path Writes to SSD

Tenet leverages `io_uring` [8] for accelerating SSD writes. `io_uring` is a high-performance asynchronous IO framework. `io_uring` maintains two queues, a submission queue (SQ) where the Tenet adds its disk write requests and a completion queue (CQ) where Tenet can poll for the completed disk writes. Both queues are shared between the kernel and the user space, which further reduces the context-switching overhead for request submission and polling.

Technique. Tenet maintains a per-writer replica buffer in the NVM, where writers enqueue the new master objects that are created in the ongoing transaction and the objects that are updated with the latest checkpoints from the `CLog` (❽ in Figure 4). Tenet then spawns multiple workers to visit the per-thread replica buffer and issue the disk writes using `io_uring`'s submission queue. The workers then poll for the request completion in the `io_uring`'s completion queue and exit only when all the requests are completed. Tenet creates a separate disk file for each master object pool; during replication, Tenet *writes a master object at the disk file offset, same as the objects' corresponding NVM file offset.* This is critical to correctly roll back the corrupted page from the disk to the NVM during the recovery.

4.4.4 Enforcing NVM-SSD Consistency

Although replication is asynchronous, Tenet *guarantees that no committed data will be lost upon either a crash or a UME.* Tenet accomplishes this by leveraging the `OLog`, `CLog`, and grace period detection.

Grace period detection in TimeStone. A grace period is the quiescence period, in which all application threads that entered the critical section (since the start of detection), finish, and exit their respective critical section. A background thread (`gp-thread`) continuously detects the grace period, and publishes the detected grace period timestamp. TimeStone uses the timestamp to safely reclaim/free the obsolete entries/objects in the `TLog`, `CLog`, and the `OLog`. Tenet extends this design to enforce NVM-SSD consistency.

Modified grace period detection in Tenet. To detect a grace period, the gp-thread not only waits for all the threads to exit the critical section but also waits for all master objects that are created/updated by these threads to be written to the SSD. The key invariant is that *when a grace period is detected, it guarantees that all master objects created/updated in that window are persisted to the SSD*. This means that the TLog, OLog, and the CLog will not be reclaimed until the disk writes are guaranteed to be persisted. That is because gp-thread will not publish the grace period timestamp unless the disk writes are completed and without it the logs can not be reclaimed. *In a nutshell, all the updates that are not persisted in the SSD are guaranteed to be either in the OLog (newly created master objects) or in the CLog (updates to the existing master object).*

Guaranteeing consistent loss-less recovery. If a UME occurs before the SSD writes finish, during recovery, TENET can restore the NVM objects with the stale SSD replica (from the previous grace period). Then it uses the CLog to update the existing master objects with the latest checkpoints and uses OLog to recreate the new master objects that are missing in the stale replica. Note that TENET maintains a consistent backup of OLog and CLog at all times (§4.4.1). Also, the OLog and CLog execution are idempotent *i.e.*, re-executing the same log entries multiple times does not violate the consistency. TENET can tolerate multiple UMEs across any number of pages in a master object pool as it replicates to the SSDs. Given at least one of the log pools is consistent, TENET can recover up to the last committed transaction. Note that even if both the log pools are affected by UMEs, TENET can still recover the master objects to the state of last grace period.

4.5 Recovery

(1) Recovering from non-UME crashes. This recovery includes recovering from a system crash or a memory safety violation. Upon restart, the recovery procedure is of two steps: (1) CLog replays, where all the entries in the CLog are replayed to set the master objects to a consistent state. This step is necessary to bring all the master objects to the latest checkpointed state. (2) Then all OLog entries are sorted based on their commit-ts and replayed sequentially in the exact sorted order. This will bring the master objects to the last committed state before the crash occurs. Note that, if the crash happens due to a memory safety violation, a developer should fix the bug to avoid repetitive non-UME crashes.

(2) Recovering from a UME crash. Upon restart, if TENET cannot open its NVM pools, it indicates a UME has occurred. The recovery steps depend on the victim pools' type.

UME in the master object pool. TENET identifies the corrupted physical offset using the ndctl tool [9] and then extracts the corresponding logical file offset. TENET brings the entire page where the corrupted offset belongs from the replica disk file. Then TENET allocates a new NVM page using fallocate and updates it using the disk replica. Finally, it deallocates the corrupted page and removes it from the operating system's bad block list. Once NVM is restored, TENET recovers similar to the non-UME crash as explained in **(1)**, *i.e.*, CLog replay followed by the OLog replay.

UME in a log pool. TENET does not need to access the disk to fix the bad page. Instead, it fixes the affected NVM page by allocating a new empty page. Then TENET uses the uncorrupted backup log pool to perform CLog and OLog replay. At the end of the recovery, it frees all the CLogs and OLogs, and new logs are allocated during the normal execution.

5 Implementation

TENET library is implemented in C and C++ which is ∼11K LoC. The core TENET library includes the TimeStone PTM (∼7K LoC), memory safety checks (∼1.5K LoC), and the NVM-SSD replication (∼2.5K LoC). We rigorously tested TENET with a carefully curated set of unit tests, functional tests, and integration tests along with the offline testing tools such as the Pmemcheck [48], Address sanitizer [80] to ensure correctness of our implementation.

6 Discussion

In this section, we discuss the key takeaways in TENET (§6.1) and the applicability of TENET's ideas on ARM architecture (§6.2). We also discuss the limitations and potential future research directions in §6.3.

6.1 Leveraging the Concurrency Guarantees of PTM

Enforcing low overhead spatial safety. Most PTMs perform out-of-place updates to enforce the Isolation property (ACID) [30, 40, 56, 62, 68, 87], to support concurrent read and write [30, 40, 56], and to enable write batching [30, 56, 87]. These PTMs have at least two separate domains: one in which new updates are made and buffered, and another that contains consistent data (*i.e.*, old updates) to which the new updates are eventually merged. TENET leverages this property to enforce a separate protection domain, such a design enables it to use light-weight techniques such as MPK and canaries to enforce spatial safety *without having to check every access*.

PTMs such as the libpmemobj [47] that perform in-place updates can be modified to perform out-of-place updates as done in Pangolin [94]. Although Pangolin uses microbuffering to perform out-of-place updates, it relies on expensive data checksum to enforce spatial safety *i.e.*, checksum is calculated and verified every time the data is moved to and from the microbuffers. SafePM [27] relies on compiler instrumentation of loads and stores and hence it needs to perform spatial and temporal safety checks at every access resulting in a high performance overhead (§7.3).

Enforcing low overhead temporal safety. Almost all PTMs support a stronger Consistency (ACID) guarantee such as linearizability or serializability. Such PTMs usually perform conflict checks (*i.e.*, read/write set validation) during the commit phase and the transactions are aborted if a read-write conflict is observed during the validation. In the context of temporal safety, this means that objects with live references in any on-going transaction will not be freed until those transactions finish. Unlike the prior PTM works, TENET leverages this property to perform temporal safety checks only at the first dereference and avoids redundant checks during every pointer deference in a transaction. This is because, once an object is dereferenced, it can not be freed by concurrent transactions, a inherent guarantee provided by PTM.

6.2 TENET's Ideas on ARM Architecture

ARM processors support memory domains [2], which is similar to Intel MPK except that the permission switch happens in the OS kernel. Moreover, ARM processors have been supporting virtual address (pointer) tagging (upper 12-16 bits) at the hardware level and it is shipped with the *top byte ignore (TBI)* feature [14, 19, 23]. Therefore, we believe that TENET's ideas can be applied beyond x86 architectures.

6.3 Limitations and Future Work

Protecting against intra-object overflow. Protecting against intra-object overflow is a hard, open research problem. Even the state-of-the-art techniques, such as BOGO [95] do not protect against intra-object overflow. We believe that protecting against intra-object overflow with reasonable performance overhead would require significant architectural changes and/or compiler-level instrumentations because of the fine granularity of protection [46, 90]. However, TENET protects the transactional metadata which are essential for correct execution and recovery from the intra-object overflow. We do this by placing an additional intra-object canary between the metadata section and the application data section in a DRAM object (not shown in the figures). This restricts the corruption to only the application data section of an object.

Protecting against the code outside the transaction. TENET already protects the NVM data from spatial safety violations due to the code outside the transaction by using MPK. However, it is possible to corrupt the DRAM objects outside the transaction and TENET may not detect such corruption, particularly the ones that do not overwrite the canaries. One way to protect the DRAM objects is to protect all the `TLogs` using the MPK and allow to switch permission only within the TENET library. However, as `TLog` is per-thread and there are only 16 MPKs available, we may need to employ MPK virtualization [74] to offer a more fine-grained protection.

Impact of shorter tags. In TENET, we use all the upper 16-bits to store the pointer tag; expansion of address space in the future will reduce the number of available bits thus making the tag range shorter. TENET allows to reuse of duplicate tags across different pointers, but if the bits are too few (e.g., only 4 bits are available), reusing tags may cause false negatives. In TENET, tag reuse becomes a problem, only if the reallocated pointer is assigned with the same tag (that it had before last free), which makes TENET 's temporal safety detection probabilistic. Reusing tags across different pointers or the same pointer with non-consecutive reallocations results in a deterministic detection. As the CPU vendors are extending hardware support for pointer tagging, we believe that expanding this idea to overcome bit limitations (e.g., similar to x86 segmentation overcoming 64KB address limitation) will be an interesting future work.

7 Evaluation

We evaluate TENET by answering the following questions, (1) what are the performance overhead of TENET's memory safety and off-critical path disk replication techniques (§7.1)? (2) How does TENET perform in comparison with the other state-of-the-art memory safe PTMs (§7.3)? (3) What is the tail latency of TENET (§7.4)? (4) How does TENET fare in the bug detection, correction, and recovery stress tests (§7.5)?

Evaluation platform. We use a system with Intel Optane DC Persistent Memory (DCPMM). It has two sockets with Intel Xeon Gold 5218 CPU with 16 Physical cores, 256GB of NVM (2×128GB), 32 GB of DRAM (2×16GB) per socket, and 2×1TB M.2 SSDs (Samsung 970 EVO). We used GCC 11.2.1 with `-O3` flag to compile benchmarks and ran all our experiments on Linux kernel 5.16.12 with `io_uring` support.

Configuration. We preset the size of `TLog` and `OLog` to 8 MB and `CLog` to 32 MB, respectively. We also present the performance analysis for varying log size in §7.4. We use two SSDs for NVM replication *i.e.*, one SSD per socket. Throughout our evaluation, we present two versions of TENET: (1) **TENET-MS** *– which enforces only memory safety (i.e., no NVM/SSD replication)*, and (2) **TENET** *– which enforces both memory safety and NVM/SSD replication for fault tolerance*. For microbenchmarks, we initially warm up the data structures with 1 Million (M) keys followed by executing a mix of lookup, insert, update, and delete operations for 60 seconds as done in the prior PTM works [30, 40, 56, 75–77, 87]. For the real-world evaluation, we use the YCSB benchmark [29] to evaluate TENET's B+Tree based key-value store engine for 10M keys, we use 8 bytes integer keys and 100 bytes values with Zipfian distribution. We present the average performance of 10 runs, with an average error rate of ±1.8%.

7.1 Performance Analysis of TENET

Figure 6 compares the performance of TENET-MS and TENET against the TimeStone for three different workloads with varying read/write ratios. Comparing TENET-MS and TENET with TimeStone will enable us to quantify the overheads due to memory safety and fault tolerance techniques.

7.1.1 TENET-MS vs TimeStone

For the read-dominated workloads, TENET-MS performs mostly on-par (< 5% overhead) or slightly better than the TimeStone. This is because reads in TENET-MS require only temporal safety checks and the overhead from spatial safety checks are negligible due to the lower write ratio. The low overhead temporal safety checks can be attributed to our in-place pointer tagging technique wherein it only requires one shifting operation for extracting the tag from the pointer and one compare operation for validating the extracted tag.

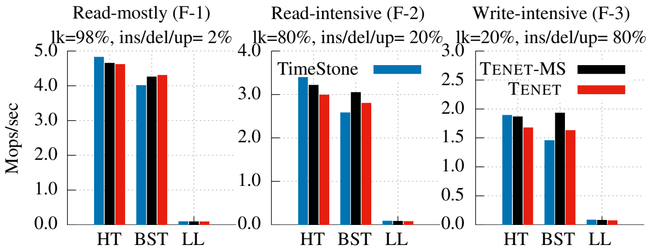


Figure 6: Performance comparison of TENET-MS and TENET against TimeStone for Hash Table (HT), Binary Search Tree (BST), and Linked List (LL) for 24 threads.

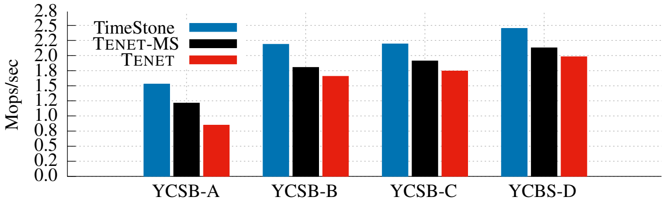


Figure 7: Performance comparison of TENET-MS and TENET against TimeStone for the B+tree key-value store with 24 threads.

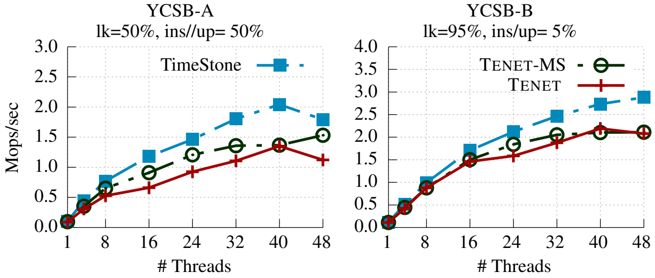


Figure 8: Scalability of TENET-MS and TENET for B+tree

PTM	Spatial Safety	Temporal Safety	UME	NVM Cost
Libpmemobj [47]	No	No	Yes	High
TimeStone [56]	No	No	No	None
SafePM [27]	Yes	Yes	No	Moderate
Pangolin [94]	Partial	No	Yes	Moderate
TENET-MS	Yes	Yes	No	None
TENET	**Yes**	**Yes**	**Yes**	**Low**

Table 1: Comparison of TENET against other PTMs.

For write-intensive workload, TENET-MS performs on par with TimeStone; this shows that our canary based spatial safety checks incur only a minimal overhead. For BST, TimeStone suffers from high transaction aborts due to lock conflicts on parent nodes. Unlike the BST, hash table is inherently more concurrent and incurs lower aborts due to less lock conflicts. Memory safety validation steps in TENET-MS reduce the aborts; our further analysis revealed that TimeStone incurs about 3.5× more aborts than TENET-MS for BST. Consequently, TENET-MS performs on par with TimeStone for hash table and slightly faster in case of a BST.

7.1.2 TENET vs TimeStone

In addition to memory safety, TENET guarantees fault tolerance by performing NVM/SSD replication. For read-mostly workloads, TENET performs on par with that of TimeStone and TENET-MS. Due to a lower write ratio, the number of log writes, and master object writes are less; consequently replication does not add any significant overhead. However, the replication overhead becomes evident as the write ratio increases from 20% to 80% and TENET performs up to 12.6% and 18% slower than the TENET-MS and TimeStone, respectively. As the master objects are inserted/deleted/updated frequently, the replica writes to SSD also increases. Therefore, grace period detection is relatively longer in TENET as the `gp-thread` has to wait for all the SSD writes to complete. A longer grace period detection increases traffic in the `TLog` as the log reclamation becomes slower. Overall, TENET adds a modest overhead (< 18%) over TimeStone while enforcing memory safety and fault tolerance.

7.2 Real-world Workload Evaluation

We built a B+tree-based key-value store using TENET; we chose B+tree (fanout=64) to test and evaluate our array interface but any other data structures can also be used. Figure 7 compares the performance of TENET-MS and TENET key-value store against the TimeStone key-value store.

TENET-MS. TENET-MS is 17% slower than the TimeStone across all YCSB workloads. For data structures (that do not use an array), such as the hash table, every read to a hash node requires only one object dereference because each hash node is a master object. But for a B+tree, reading one leaf node requires a 2× fanout (2×64) number of dereferences as each array element (of the key-value array) is a master object. Although TimeStone incurs the same number of object dereference, the additional temporal safety checks during the object dereferencing in TENET-MS causes a 17% slowdown.

TENET. For write-intensive YCSB-A, TENET performs 41% slower than TimeStone. This is because of lower chances of write coalescing in the `TLog` and `CLog`. As the writes happen at the array element level, the chances of an array index being repeatedly written to is less. This is the worst-case scenario for TimeStone as it relies on maximizing write coalescing on DRAM objects to reduce NVM writes. Lower write-coalescing causes frequent checkpoints (from `TLog`) on `CLog` and frequent checkpoint writebacks (from `CLog`) to the NVM object. TimeStone just performs frequent writebacks to the NVM object; for TENET, increase in the number of writebacks also increases the SSD writes due to replication. This trend is corroborated by the performance of TENET for read-intensive YCSB workloads (B, C, and D), where it exhibits only a 21% slowdown against TimeStone. This is almost half of the slowdown experienced for the YCSB-A workload (41%) as the number of SSD writes are lower in read-intensive workloads. In a nutshell, TENET guarantees memory safety for arrays (TENET-MS) with a modest 17% overhead and providing fault tolerance adds an additional 24% overhead due to the reduced write coalescing in TimeStone.

7.3 Comparison with Other PTMs

Table 1 compares the protection scopes of PTMs; TENET is the only PTM to offer full memory safety and cost-efficient

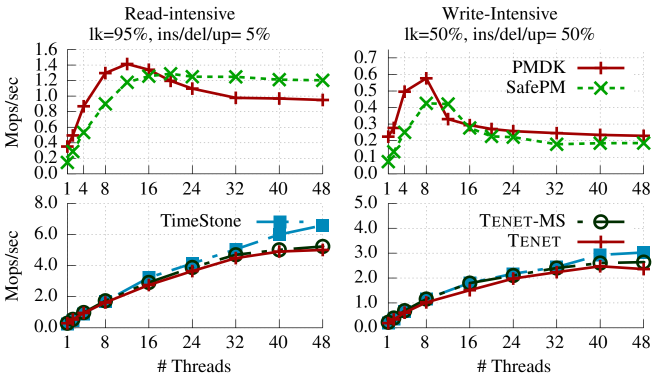


Figure 9: TENET-MS vs SafePM: performance overhead study with hash table for read-intensive and write-intensive workloads.

fault tolerance. We have discussed the limitations in the protection scope of prior works in §2.3. Moreover, TENET incurs a relatively minimal performance overhead as compared to SafePM (Figure 9) and Pangolin which incurs up to 60% and 67% overhead over the `libpmemobj`.[1] To ensure fairness, we compare SafePM and TENET-MS on basis of performance overhead incurred over their respective baseline PTM. Note that SafePM does not guarantee fault tolerance against the UMEs, so we use only TENET-MS for comparison.

As shown in Figure 9, SafePM performs up to 67% slower than the `libpmemobj` across both the workloads. When the `libpmemobj`'s performance saturates after 16 threads, SafePM performs on-par; this is because the high contention overhead in the `libpmemobj` amortizes the memory safety overhead in SafePM. SafePMs' overheads come from: (1) additional undo logging to guarantee crash consistency for the memory safety metadata. Note that this undo logging is in addition to the ones performed by the `libpmemobj` transaction, (2) the memory safety metadata must be accessed for every read and write which further slows down the performance.

Unlike the SafePM, TENET-MS guarantees memory safety with a modest 5%-8% performance overhead; because, (1) it does not require additional crash consistency for memory safety metadata as the pointer tags are embedded in the objects, and (2) memory safety checks are performed only once per transaction (on-commit and on-first-dereference).

7.4 Other Evaluations and Analysis

Scalability analysis. Figure 8 and Figure 9 shows the read and write scalability of TENET-MS and TENET for hash table and B+tree, respectively. Both TENET-MS and TENET show good read and write scalability for B+tree and hash table. The performance difference across thread counts are consistent with what is observed for 24 threads in Figure 6 and Figure 7. For read-intensive workloads, TENET-MS and TENET show less than 5% performance slowdown for a hash table and a 17% (TENET-MS) and 24% (TENET) slowdown for a B+tree. For a write-intensive hash table, TENET-MS and TENET exhibit a 5% and 18% slowdown respectively, while for B+tree, TENET-MS and TENET exhibit a 17% and

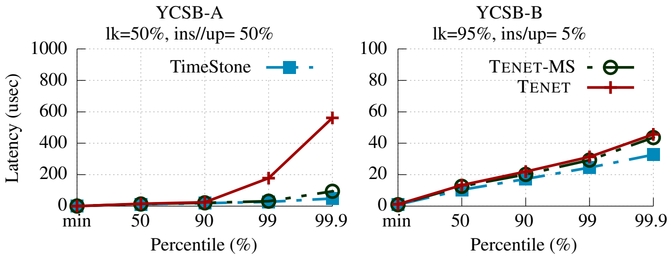


Figure 10: Tail latency comparison of TENET-MS and TENET against TimeStone for B+tree with 24 threads.

44% slowdown. Overall, both TENET-MS and TENET scales on-par with TimeStone; this shows that the TENET*'s memory safety and fault tolerance techniques does not impede the original scalability of TimeStone*.

Storage cost analysis. With TENET, the DRAM space usage is bounded by the size of `TLog` (8MB). TENET stores the replica logs in the NVM and this is bounded by the size of `OLog` and `CLog`. TENET replicates the application data structure to the SSD; given the $/GB of SSD ($0.15) and the NVM ($10) [15, 20], TENET saves ∼60× on storage cost when replicating the entire NVM space (512GB) to the SSD as opposed replicating to the NVM. In addition to the cost benefits, TENET can recover from multiple UMEs spanning across multiple pages while Pangolin can recover only from a single page is corruption.

Tail latency. Figure 10 shows the tail latency of TENET-MS and TENET compared against the TimeStone. As done in prior works [55, 60], we sample 10% of operations so that the tail latency calculation does not overshadow the performance. TENET-MS performs on-par with TimeStone, which shows the efficacy of our memory safety techniques. However, for write-intensive YCSB-A, TENET's tail latency spikes up at the 99th and 99.9th percentile. This is because of the additional writes incurred while performing replication to the NVM/SSD for fault tolerance. For read-intensive workload, TENET's tail latency is almost on par with TimeStone as lower ratio reduces the number of SSD writes. TENET-MS shows similar tail latency to that of the TimeStone across workloads as it does not perform replication. We believe our fault tolerance design can be further optimized for tail latency by making log writes asynchronously, which would be an interesting future work.

Log size sensitivity. To study the impact of log size on the performance, we present the relative performance of TENET for varying log sizes using a concurrent hash table with 1 and 24 threads (Figure 11). We show the performance only for write-intensive workloads as read-intensive workloads are less sensitive to the log size. The X-axis represents the log size, and the Y-axis represents the relative performance normalized to the default log size used in all the previous evaluations. TENET's performance increases up to 21% with the increasing log size. As the log size is decreased, the performance drops to 38%. As the log size increases, the writers spend less time reclaiming log space and hence better performance.

[1] Directly referenced from the paper as Pangolin is not open-sourced.

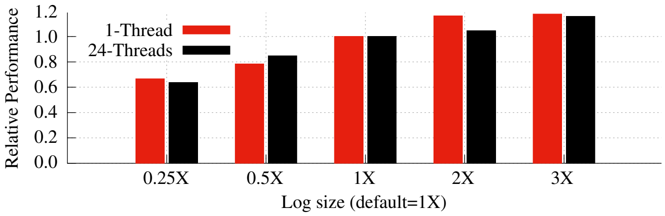


Figure 11: Performance sensitivity of Tenet for varying log sizes.

Alternatively, for smaller log sizes, the writers spend more time reclaiming log space. Tenet requires all SSD writes in a grace period window to be persisted before reclaiming the log space, further increasing the pressure on the writers. So we observe a larger performance drop (38%) for a smaller log size and relatively a smaller performance gain (21%) when the log size is increased. We confirmed that this behavior is consistent across different thread counts and data structures.

7.5 Error Detection and Correction

Spatial safety test. Our test cases select transactions at random to intentionally cause *buffer overrun* bugs on a B+tree leaf nodes' value pointer (`p_val`) and to access the key array (in a B+tree node) *out-of-bounds*. For the buffer overflow bug, the erroneous transactions execute a `memcpy` on the `p_val` for 1KB where the `p_val` pointer is of size 100 bytes. We also tested intra-array overflow with a smaller size of 128 bytes. For out-of-bound access, the erroneous transactions access the key array at index *96*, which is beyond the original fanout (64). For all test cases, Tenet detected spatial safety violations in the commit phase and aborted the transactions, returning an exception to the B+tree code. In our 200 random tests, Tenet detected spatial safety violations 100% of the time.

Temporal safety test. We modified the delete function in our open-chaining hash table benchmark to free the target node and *not update the previous nodes' next pointer (`p_next`)*. A randomly chosen transaction executes the buggy delete logic and spawns read transactions to access the dangling `p_next`. Tenet detected the dangling pointer access during the object dereferencing phase and returned an exception to the application. Further, to test the case where a free-ed address may be reallocated again, we kept allocating a new hash node until the free-ed NVM address was reallocated. Our test case then waits for a transaction to access the *dangling `p_next` (reallocated)*. We repeated both the temporal safety tests 200 times, and Tenet detected dangling pointer access and returned an exception to the application.

UME Test. We used the `ndctl` utility tool (*`ndctl` inject-error*) for injecting a UME at a specified offset [9]. While running the benchmark, we first injected a UME in the log pool, particularly on a randomly chosen `CLog`. Tenet's `SIGBUS` handler received the OS notification and terminated the program gracefully. Upon restart, Tenet rightly identified the corrupted log pool and successfully recovered using the replica log pool. We also injected UME in one of the master object pools and observed that Tenet restored the NVM status successfully using the SSD replica. Both these tests were repeated multiple times and Tenet successfully recovered the hash table without losing any data. The recovery time for Tenet and TimeStone are similar, bounded by `OLog` and `CLog` size (not shown due to space constraints). The SSD access is performed in the background using `io_uring` and the cost is relatively small. Our future work will develop techniques to accelerate recovery.

8 Related Work

DRAM based memory safety techniques. Memory safety violation in the DRAM has been extensively studied in the security community [26, 32, 35, 36, 59, 69–71, 73, 79, 81, 83, 84, 92, 95]. In fact, our work was inspired by this line of research which essentially conveys that memory safety violations are the source of all evils. But the downside of these techniques is that they suffer from high performance overhead (up to 200%). In Tenet, we reduce the performance overhead by leveraging the concurrency properties of the PTM and also by limiting our scope of protection (*e.g.*, no support for control flow attacks). Moreover, applying these DRAM based techniques to NVM is non-trivial as they are not designed to be crash consistent and adding crash consistency to these techniques comes with its own set of challenges and may potentially increase the performance overhead.

NVM bug finding techniques. There are a plethora of works on detecting crash consistency bugs in the NVM software [37, 38, 58, 63, 64, 72]. These techniques primarily focus on detecting bugs that violate crash consistency correctness such as atomicity, linearizability, and persistence ordering bugs; they neither focus on memory safety nor UMEs.

9 Conclusion

In this paper, we propose Tenet. Tenet enforces DRAM/NVM memory domain separation using MPK to prevent NVM writes out of Tenet library. Additionally, Tenet uses canary values and in-place pointer tagging to guarantee on-commit spatial safety and on-first-dereference temporal safety. Further, Tenet proposes off-critical path NVM/SSD data replication to guarantee a performance and cost-efficient fault tolerance for the NVM data against the UMEs. Our evaluations showed the performance efficiency of Tenet's techniques along with a thorough analysis on scalability, storage cost, and tail latency. Overall, Tenet provides enhanced NVM data protection at a modest performance and storage cost as compared to the other state-of-the-art PTMs.

Acknowledgments

We thank the anonymous reviewers and Adam Morrison (our shepherd) for their insightful comments and feedback. This work was partly supported by the National Science Foundation under the grants CCF-2153748, CNS-1910593, and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2022R1F1A1076373).

References

[1] Amazon Signs up for Another 450MW of Solar, Giant Batteries. https://www.datacenterknowledge.com/energy/amazon-signs-another-450mw-solar-giant-batteries.

[2] ARM Developer Suite Developer Guide: Memory access permissions and domains. https://developer.arm.com/documentation/dui0056/d/caches-and-tightly-coupled-memories/memory-management-units/memory-access-permissions-and-domains.

[3] Blast Radius. https://pmem.io/glossary/#blast-radius.

[4] Build Persistent Memory Applications with Reliability Availability and Serviceability. https://www.intel.com/content/www/us/en/developer/articles/technical/build-pmem-apps-with-ras.html.

[5] Chrome: 70% of all security bugs are memory safety issues. https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/.

[6] CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition. https://cwe.mitre.org/data/definitions/367.html.

[7] Dealing with Uncorrectable Errors. https://www.intel.com/content/www/us/en/developer/articles/technical/pmem-RAS.html.

[8] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[9] Error Recovery in Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/error-recovery-in-persistent-memory-applications.html.

[10] Frequently Asked Questions for Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/support/articles/000056000/memory-and-storage/intel-optane-persistent-memory.html.

[11] Google Thinks Data Centers, Armed with Batteries, Should 'Anchor' a Carbon-Free Grid. https://www.datacenterknowledge.com/google-alphabet/google-thinks-data-centers-armed-batteries-should-anchor-carbon-free-grid.

[12] Intel Donates Compute Express Link, a High-Speed Protocol for PCIe 5.0. https://www.tomshardware.com/news/intel-compute-express-link-pcie-5.0,38786.html.

[13] Intel Kills Optane Memory Business, Pays $559 Million Inventory Write-Off. https://www.tomshardware.com/news/intel-kills-optane-memory-business-for-good.

[14] Intel Linear Address Masking "LAM" Ready For Linux 6.2. https://www.phoronix.com/news/Intel-LAM-Linux-6.2.

[15] Intel Optane DCPMM Cost. https://www.anandtech.com/show/14180/pricing-of-intels-optane-dc-persistent-memory-modules-leaks.

[16] Last week Intel killed Optane. Today, Kioxia and Everspin announced comparable tech: Rumors of storage-class memory's demise may have been premature. https://www.theregister.com/2022/08/02/kioxia_everspin_persistent_memory/.

[17] Microsoft: 70 percent of all security bugs are memory safety issues. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[18] Microsoft slashes backup power costs with lithium-ion batteries. https://www.computerworld.com/article/2895064/microsoft-slashes-backup-power-costs-with-lithiumion-batteries.html.

[19] Pointer tagging for x86 systems. https://lwn.net/Articles/888914/.

[20] Samsung EVO NVMe M.2 SSD Cost . https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-970-evo-nvme-m-2-1tb-mz-v7e1t0bw/.

[21] Samsung's Memory-Semantic CXL SSD Brings a 20X Performance Uplift. https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift.

[22] SMART brings Optane memory to AMD and Arm. https://blocksandfiles.com/2022/04/13/smart-brings-optane-memory-to-amd-and-arm/.

[23] The Arm64 memory tagging extension in Linux. https://lwn.net/Articles/834289/.

[24] Timestone Source Code. https://github.com/cosmoss-jigu/timestone/tree/master.

[25] Trends, challenge, and shifts in software vulnerability mitigation. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[26] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[27] Kartal Kaan Bozdoğan, Dimitrios Stavrakakis, Shady Issa, and Pramod Bhatotia. Safepm: A sanitizer for persistent memory. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*, Rennes, France, April 2020.

[28] Brian Choi, Randal Burns, and Peng Huang. Understanding and dealing with hard faults in persistent memory systems. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, online, April 2021.

[29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, Indiana, USA, June 2010. ACM.

[30] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th ACM symposium on Parallelism in algorithms and architectures (SPAA)*, Vienna, Austria, July 2018.

[31] CXL Consortium. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/.

[32] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[33] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. Poseidon: Safe, fast and scalable persistent memory allocator. In *Proceedings of the 21st ACM/IFIP International Middleware Conference*, Virtual, December 2020.

[34] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[35] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. HeapHopper: Brining Bounded Model Checking to Heap Implementation Security. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[36] Chris Evans. The poisoned NUL byte, 2014 edition, 2014. https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html.

[37] 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 28th ACM Symposium on Operating Systems Principles (SOSP)*, online, October 2021.

[38] 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)*, Carlsbad, CA, November 2022.

[39] Bill Gervasi. A Persistent CXL Memory Module with DRAM Performance. In *Storage Developer Conference (SDC)*. SNIA, 2022. https://storagedeveloper.org/conference/agenda/sessions/persistent-cxl-memory-module-dram-performance.

[40] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 913–928, Renton, WA, July 2019.

[41] Pekon Gupta. CXL Attached Persistent Memory: Implementing NVDIMM-N Like Architecture. In *Storage Developer Conference (SDC)*. SNIA, 2022. https://storagedeveloper.org/conference/agenda/sessions/cxl-attached-persistent-memory-implementing-nvdimm-n-architecture.

[42] Jim Handy and Thomas Coughlin. Persistent Memories Without Optane, Where Would We Be? In *Storage Developer Conference (SDC)*. SNIA, 2022. https://storagedeveloper.org/conference/agenda/sessions/cxl-attached-persistent-memory-implementing-nvdimm-n-architecture.

[43] 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 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[44] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[45] IBM. Power Failure Handling - IBM i in a Hosted Environment. https://www.ibm.com/support/pages/power-failure-handling-ibm-i-hosted-environment.

[46] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-fat: Architectural support for low overhead memory safety checks. In *Proceedings of the 48th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, online, June 2021.

[47] Intel. C++ bindings for libpmemobj (part 6) - transactions, 2016.

[48] INTEL. Valgrind: an enhanced version for pmem, 2019.

[49] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2019. https://software.intel.com/en-us/articles/intel-sdm.

[50] Raghunathan Modoor Jagannathan, Sulav Malla, and Parimala Kondety. Power Loss Siren: Making Meta resilient to power loss events, 2021. https://engineering.fb.com/2021/12/16/data-center-engineering/power-loss-siren/.

[51] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[52] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[53] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. Viyojit: Decoupling battery and dram capacities for battery-backed dram. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, June 2018.

[54] Rajat Kateja, Nathan Beckmann, and Gregory R. Ganger. Tvarak: Software-managed hardware offload for redundancy in direct-access nvm storage. In *Proceedings of the 47th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, online, June 2020.

[55] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. Pactree: A high performance persistent range index. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, online, October 2021.

[56] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[57] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. TIPS: Making volatile index structures persistent with DRAM-NVMM tiering. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, online, July 2021.

[58] 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 (ATC)*, Philadelphia, PA, June 2014.

[59] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[60] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating Persistent Memory Range Indexes. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, Los Angeles, CA, August 2019.

[61] Jihang Liu, Shimin Chen, and Lujun Wang. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

[62] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[63] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[64] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, April 2019.

[65] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

[66] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query optimized persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, Virtual, February 2021.

[67] Sulav Malla, Qingyuan Deng, Zoh Ebrahimzadeh, Joe Gasperetti, Sajal Jain, Parimala Kondety, Thiara Ortiz, and Debra Vieira. Coordinated priority-aware charging of distributed batteries in oversubscribed data centers. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pages 839–851. IEEE, 2020.

[68] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. EuroSys17.

[69] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[70] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[71] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, Toronto, Canada, June 2010.

[72] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. AGAMOTTO: How persistent is your persistent memory application? In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, November 2020.

[73] Gene Novark and Emery D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, page 573–584, Chicago, IL, November 2010.

[74] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 241–254, Renton, WA, July 2019.

[75] Pedro Ramalhete, Andreia Correia, and Pascal Felber. Onefile: A wait-free persistent transactional memory. In *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN)*, June 2019.

[76] Pedro Ramalhete, Andreia Correia, and Pascal Felber. Persistent memory and the rise of universal constructions. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, Heraklion, Greece, April 2020.

[77] Pedro Ramalhete, Andreia Correia, and Pascal Felber. Efficient algorithms for persistent transactional memory. In *Proceedings of the 24th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, online, March 2021.

[78] Arthur Sainio and Pekon Gupta. Scaling NVDIMM-N Architecture for System Acceleration in DDR5 and CXL-Enabled Applications. In *PM+CS Summit*. SNIA, 2022. https://www.snia.org/educational-library/scaling-nvdimm-n-architecture-system-acceleration-ddr5-and-cxl-enabled.

[79] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 309–318, Boston, MA, June 2012.

[80] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[81] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[82] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[83] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2017.

[84] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[85] Haris Volos. The case for replication-aware memory-error protection in disaggregated memory. *IEEE Computer Architecture Letters*, 2021.

[86] Li Wang, Zining Zhang, Bingsheng He, and Zhenjie Zhang. PA-Tree: Polled-Mode Asynchronous B+ Tree for NVMe. In *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE)*, Dallas, TX, April 2020.

[87] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. ArchTM: Architecture-Aware, high performance transaction for persistent memory. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, February 2021.

[88] 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)*, Santa Clara, California, USA, February 2016.

[89] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[90] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, April 2021.

[91] Yuanchao Xu, Wei Xu, Kimberly Keeton, and David E. Culler. Scaling NVDIMM-N Architecture for System Acceleration in DDR5 and CXL-Enabled Applications. In *Non-Volatile Memory Workshop (NVMW)*, 2022. `http://nvmw.ucsd.edu/nvmw2022-program/nvmw2022-data/nvmw2022-final5.pdf`.

[92] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual, August 2020.

[93] Da Zhang, Vilas Sridharan, and Xun Jian. Exploring and optimizing chipkill-correct for persistent memory based on high-density nvrams. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Fukuoka, Japan, October 2018.

[94] Lu Zhang and Steven Swanson. Pangolin: A Fault-Tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[95] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 631–644, Providence, RI, April 2019.

[96] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, 2022.

MadFS: Per-File Virtualization for Userspace Persistent Memory Filesystems

Shawn Zhong* Chenhao Ye* Guanzhou Hu Suyan Qu
Andrea Arpaci-Dusseau Remzi Arpaci-Dusseau Michael Swift
University of Wisconsin–Madison

Abstract

Persistent memory (PM) can be accessed directly from userspace without kernel involvement, but most PM filesystems still perform metadata operations in the kernel for security and rely on the kernel for cross-process synchronization.

We present per-file virtualization, where a virtualization layer implements a complete set of file functionalities, including metadata management, crash consistency, and concurrency control, in userspace. We observe that not all file metadata need to be maintained by the kernel and propose embedding insensitive metadata into the file for userspace management. For crash consistency, copy-on-write (CoW) benefits from the embedding of the block mapping since the mapping can be efficiently updated without kernel involvement. For cross-process synchronization, we introduce lock-free optimistic concurrency control (OCC) at user level, which tolerates process crashes and provides better scalability.

Based on per-file virtualization, we implement MadFS, a library PM filesystem that maintains the embedded metadata as a compact log. Experimental results show that on concurrent workloads, MadFS achieves up to 3.6× the throughput of ext4-DAX. For real-world applications, MadFS provides up to 48% speedup for YCSB on LevelDB and 85% for TPC-C on SQLite compared to NOVA.

1 Introduction

Persistent memory (PM) is a promising candidate for next-generation storage devices. PM DIMMs are connected on the memory bus and deliver near-DRAM performance while persisting data across power-offs. They create new opportunities for building storage systems.

With revolutionary hardware available, the software stack needs to evolve accordingly. Traditional kernel filesystems require I/O operations to cross the user-kernel boundary and go through layers of the storage stack, introducing significant software overhead. In response to this observation, many PM filesystems have been proposed to perform I/O in userspace [5, 8, 12, 25, 30, 41, 43]. The challenge is that a userspace process is untrusted and unreliable: it could corrupt metadata and threaten filesystem integrity; it could crash in a shared critical section, blocking other processes. These realities impose challenges for metadata operations and sharing. Existing userspace filesystems bypass the kernel for data operations, but typically still rely on the kernel for metadata management [5, 8, 25, 30] with its inefficient storage stack. In terms of sharing, most userspace filesystems either do not support cross-process sharing [8] or rely on a kernel-granted lease [5, 12, 30].

To address these challenges, we introduce *per-file virtualization*, where a complete set of file functionalities, including metadata management, crash consistency, and concurrency control, are implemented in a userspace virtualization layer and managed on a per-file basis for regular files. For userspace metadata management, we observe that some metadata are private to each file and have a similar trust model to the file data. Thus, we propose *metadata embedding*, where insensitive metadata (e.g., block mapping and file size) are embedded in the file. This enables efficient metadata management in userspace without sacrificing permission enforcement. In particular, embedding block mapping provides additional benefits when copy-on-write (CoW) is used for data crash consistency. For a process with memory-mapped files, existing kernel-level CoW requires updating the page table on file writes, causing expensive TLB shootdowns. With metadata embedding, the block mapping can be changed entirely in userspace without kernel involvement. To support cross-process concurrency control, we use the file data itself as the communication medium and implement non-blocking synchronization. This design simplifies the failure model and provides better concurrency than locks.

Based on per-file virtualization, we present MadFS[1], a library filesystem for persistent memory that provides strong data crash consistency and linearizable concurrency control

[*] Both authors contributed equally to this work.

[1] MadFS stands for m̲etadata̲ embed̲ded f̲ilesystem.

in userspace. MadFS requires no modification to the kernel or application and can run on top of any direct access (DAX) filesystem with `mmap` support (e.g., ext4-DAX). To provide strong data crash consistency, MadFS performs CoW on data updates. MadFS introduces a level of indirection that maps *virtual* blocks seen by applications to *logical* blocks backed by the underlying kernel filesystem. This block mapping is embedded in the file for efficient userspace CoW and maintained as a log for crash consistency. We implement lock-free optimistic concurrency control (OCC) to support concurrent access to the same file cross processes. Specifically, a writer tentatively makes changes in a private workspace. Before committing to the log, the writer detects conflicts by checking the movement of the log tail, and partially redoes the changes if necessary. Compared to lock-based approaches, concurrent readers and writers would not block each other even with overlapping ranges, thus achieving better scalability.

We evaluate MadFS using a variety of microbenchmarks and macrobenchmarks. MadFS achieves up to 3.6× throughput for ext4-DAX on concurrent microbenchmarks. For LevelDB running YCSB workload, MadFS provides up to 48% improvement over NOVA. TPC-C workloads over SQLite on MadFS outperform NOVA by 85%.

This paper makes the following contributions:

- We present per-file virtualization, where a virtualization layer implements a complete set of file functionalities, including metadata management, crash consistency, and concurrency control, entirely in userspace.
- We introduce metadata embedding as a novel metadata management technique for userspace filesystems. Embedding insensitive metadata in the file enables efficient modification in userspace.
- In particular, when CoW is used for data crash consistency, we propose embedding the block mapping, which allows it to be updated without the kernel modifying the page table.
- We introduce lock-free optimistic concurrency control (OCC) for userspace cross-process synchronization, which tolerates process crashes and achieves better scalability.
- Based on per-file virtualization, we present MadFS, a library PM filesystem that maintains the embedded metadata as a compact log. The source code of MadFS is available at `https://github.com/WiscADSL/MadFS`.
- We evaluate MadFS using microbenchmarks and macrobenchmarks to show that it provides high throughput for both single-threaded and multi-threaded workloads.

2 Background and Motivation

Persistent memory (PM) is an emerging hardware technology that provides durability with DRAM-like latency. PM is considered both a new generation of denser memory and a high-performance storage device. In this paper, we explore the storage aspect of PM.

The byte-addressability of PM, like DRAM, enables CPUs to directly read/write data through load/store instructions. After data is stored in a memory location, it may still reside in the CPU cache, so one needs to flush the cache line explicitly (e.g., via `clwb` or `clfushopt`) for persistence. Alternatively, non-temporal stores (e.g., `movnti`) can be used to persist data directly, bypassing the CPU cache. For ordering constraints, a memory fence (e.g., `sfence`) is needed to serialize memory instructions.

One of the commercially available PM products is Intel Optane Persistent Memory [1]. Intel announced the winding down of the Optane business in Q2 2022 [10]. This work is not specific to Intel Optane PM. We only require that the PM is byte-addressable and applications can directly access the data stored on the PM via memory-mapped I/O.

There has been a rich set of work on building more efficient filesystems for PM. In this section, we broadly classify them into userspace and kernel filesystems and then discuss their challenges in metadata management, crash consistency, and concurrency control.

2.1 Filesystems for Persistent Memory

Kernel filesystems. Mature Linux filesystems such as ext4 and XFS introduce direct access (DAX) mode [7, 42], which bypasses the page cache and allows applications to directly access file data stored on PM via memory-mapped I/O. These DAX filesystems only ensure metadata consistency in the presence of failures, while the responsibility of maintaining data consistency on memory-mapped regions falls on the applications. There are also research kernel filesystems designed for PM. BPFS [9] uses a tree layout similar to WAFL [23] and avoids cascading CoW via short-circuit shadow paging. PMFS [14] combines atomic in-place updates, journaling, and CoW to support efficient crash consistency, and also advocates the use of huge pages to reduce paging costs. NOVA [45] implements log-structured metadata for each file and CoW data crash consistency.

Userspace filesystems. With ultra-fast hardware, software overhead becomes non-trivial. Thus, many PM filesystems have proposed to bypass the kernel [5, 8, 12, 25, 30, 41, 43]. FLEX [43] calls `mmap` after `open` and intercepts data operations to handle them in userspace via memory instructions. SplitFS [25] similarly handles data operations in userspace with memory-mapped I/O but relies on a modified ext4-DAX for metadata operations. It introduces a new system call `relink`, which reassigns data blocks from one file to another. For append operations, SplitFS redirects data to a temporary staging file and invokes `relink` on `fsync` to publish the newly written data to the target file. Libnvmmio [8] builds on memory-mapped I/O and equips each block with a journal to provide scalable crash-consistent I/O.

2.2 Challenges in Metadata Management

Metadata safety is critical to filesystem integrity. In kernel filesystems, metadata is managed exclusively by the kernel for security reasons. A major challenge of userspace filesystems comes from untrusted libraries. Thus, many of them still rely on the kernel for metadata management (e.g., SplitFS [25], Strata [30], and KucoFS [5]). Unfortunately, data operations can be tightly coupled with metadata operations, defeating the purpose of kernel bypassing and leading to lower performance. For example, SplitFS appends data to a staging file, but still requires the `relink` system call to swap the data blocks from the staging file to the target one on each `fsync`.

A few filesystems also bypass the kernel for metadata operations. Aerie [41] provides applications with direct access to PM for reading/writing data and reading metadata, while metadata updates are handled by a trusted filesystem service via socket-based remote-procedure call (RPC). One drawback of this approach is that RPCs are expensive and incur the overhead of context switches. Aerie uses batching to reduce the number of RPCs at the cost of visibility. ZoFS [12] introduces a new abstraction called coffer. The dentries, inodes, and data blocks for a directory subtree are stored in a coffer if they share the same permission. ZoFS relaxes the protection domain from file to coffer and relies on the Intel Memory Protection Key (MPK) hardware for security. Due to hardware limitations of MPK, the number of simultaneously memory-mapped coffers cannot exceed 15.

2.3 Challenges in Crash Consistency

Crash consistency is critical to filesystems. PM only guarantees the atomicity of a single 64-bit store, so filesystems need to build their own constructs for crash consistency.

To ensure metadata crash consistency, PM filesystems commonly use journaling [5, 14, 25, 30, 41, 42]. Kernel filesystems adapted for PM, such as ext4-DAX, rely on Linux journaling block device (JBD) [28] for metadata journaling. However, JBD was designed with block devices in mind and writes in whole blocks, causing write amplification [4, 43]. SplitFS also uses JBD for the crash consistency of `relink` and suffers the same problem. Many filesystems tailored for PM leverage the byte-addressability to persist journal/log entries with a finer granularity [14, 45]. NOVA equips each inode with a private log. Cross-file updates are implemented via journaling to update multiple log tails. BPFS [9] uses CoW for metadata updates. SoapFS [13] and ZoFS [12] employ soft update [15] for metadata crash consistency.

For data crash consistency, CoW is commonly used [5, 9, 14, 25, 45]. However, CoW has two major drawbacks when used with memory-mapped I/O. First, huge pages have been shown to have significant performance improvements for PM filesystems due to fewer page faults, less TLB shootdown, and shorter page table walk [14, 24, 25]. However, an open issue brought out by PMFS is that CoW does not work well with huge pages: the granularity of CoW is coupled with the page size, which for huge pages is 2 MB or 1 GB on x86-64. Writing to a sub-page results in copying the entire page, causing significant write amplification [14]. SplitFS's `relink` changes the block mapping at the granularity of 4 KB blocks. This breaks the contiguity of the file on the physical PM and thus prevents the use of huge pages [24].

Second, in addition to huge pages, kernel-level CoW causes expensive TLB shootdowns [2,3,8,40]. During CoW, the page table should be updated so that the virtual address region is backed by the new pages. The kernel needs to flush the TLB on the local core, send an inter-processor interrupt (IPI) to the other cores to flush the remote TLB, and wait for all cores to finish. The whole process can take several microseconds to complete [40], which is expensive for PM devices with sub-microsecond latency [47].

Another option for data crash consistency is data journaling. Strata [30] allows applications to write to a private log in PM and relies on the kernel to digest the data to a slower storage device. Libnvmmio [8] equips each block with a journal and implements background checkpointing. In general, data journaling faces the issue of double writes. Both Strata and Libnvmmio make the digestion/checkpointing asynchronous to remove it from the critical path at the cost of visibility.

Some PM filesystems do not provide data crash consistency, including ext4-DAX, FLEX, PMFS, Aerie, and ZoFS. In this case, applications have to detect and react to inconsistent file data upon failures. Previous studies [35, 36] have shown that many applications fail to handle inconsistent data correctly. Data crash consistency is a desirable property for filesystems if the overhead is acceptably low.

2.4 Challenges in Concurrency Control

For kernel filesystems, the kernel itself acts as a single centralized entity for synchronization. The inode lock ensures that only one thread is operating on the same file at a time. For userspace filesystems, however, concurrency control is challenging, especially in cross-process cases. For example, a process could crash while holding a lock, blocking other processes. To prevent this situation, the lock must be visible to the kernel so that the kernel can release it after a crash (e.g., robust mutex [27]). This introduces additional kernel involvement and can cause processes to sleep on the critical path of data operations.

As a result, most userspace PM filesystems either do not support cross-process synchronization [8] or use lease-based locking [5, 12, 30, 41]. Aerie implements a lock service in the filesystem service. Each application process is equipped with an additional clerk thread to communicate with the lock service and synchronize with others. In Strata and ZoFS, leases are granted by the kernel. KucoFS uses a two-level locking scheme with kernel-granted leases for inter-process synchronization and userspace range locks for intra-process synchronization. In all these cases, there exists a centralized coordinator to manage leases. This adds communication overhead when multiple processes access the same file concurrently.

The lease timeout is another source of complexity. Timeout relies on the assumption about the maximum completion time of an operation, which could be unsafe. For example, a writer starting with a valid lease can finish with the lease expired. In this case, other threads will see partial data. A write operation can take an arbitrarily long time to complete due to kernel CPU scheduling or large I/O sizes. This will cause correctness issues with lease-based locking.

3 Per-File Virtualization

To address these challenges, we propose *per-file virtualization*, where a userspace virtualization layer implements a complete set of file functionalities, including metadata management, crash consistency, and concurrency control, on a per-file basis for regular files.

Kernel-bypassing with metadata embedding. We observe that some of the file metadata (e.g., block mapping and file size) are private to each file, and share the same protection domain as the file data. This allows us to embed a subset of the metadata directly into the file to avoid the slow kernel I/O stack for certain metadata operations, especially those that are tightly coupled with data operations (e.g., CoW changing block mapping). Compared to other techniques for userspace metadata management, this method neither relies on a trusted entity as in Aerie nor expands the protection domain beyond a file as in ZoFS. Permission-related metadata (e.g., access mode, owner, and group) must not be embedded. The kernel filesystem shall still manage the permission and enforce access control when a file is opened. Metadata embedding does not apply to directories since the hierarchical structure must be visible to the kernel to enforce access control. We leverage the mature constructs of the kernel to handle directory operations, while the virtualization layer manages the embedded file metadata and ensures its crash consistency.

Decoupling of block- and memory-mapping for CoW. Embedding block mapping, in particular, enables efficient userspace block management since the embedded block mapping can be modified independently from the memory mapping. This provides two major benefits when using CoW for data crash consistency. First, the granularity for CoW is no longer associated with huge page sizes. CoW can operate at block granularity within the file, while the kernel still sees the file as a contiguous region on the PM. This allows the usage of huge pages during `mmap`. Second, block mapping updates can be done in userspace via store instructions. The kernel no longer needs to modify the page table. The nanosecond-level cache coherence protocol [18, 33] ensures cross-core consistency as opposed to microsecond-level TLB shootdown [40].

Non-blocking concurrency control. The embedding of metadata brings new opportunities for concurrency control in userspace. As a file is now a self-contained entity with both metadata and data stored in it, processes that memory-map the same file can use the shared PM region for cross-process synchronization, without relying on external entities. We argue that locking is not a good candidate for cross-process synchronization, as the lock owner can crash in the middle of a critical section. Detecting the lock owner's crash without the kernel is difficult if not impossible. Instead, we propose to use atomic primitives (e.g., compare-and-swap) to implement non-blocking synchronization, where the suspension or crash of a single process does not prevent others from making progress [19–21]. In this way, inter- and intra-process concurrency control is handled uniformly, and the failure model is greatly simplified. Non-blocking synchronization also brings better concurrency, since operations do not block each other, even with overlapping ranges.

Summary. With per-file virtualization, we aim to push file functionalities into userspace as much as possible. Metadata embedding bypasses the kernel for metadata management. Embedding block mapping enables efficient userspace CoW for crash consistency. Non-blocking synchronization allows cross-process concurrency control to be enforced without kernel involvement. All the techniques are applied on a per-file basis and there is no global data structure.

4 MadFS: Design and Implementation

Based on per-file virtualization, we implement MadFS, a userspace library filesystem overlaid on top of any DAX kernel filesystem supporting `mmap` (e.g., ext4-DAX). It intercepts POSIX I/O calls and requires no modifications to the application. MadFS memory-maps the file on open, so subsequent data operations (e.g., `read` and `write`) can be handled in userspace via load and store. MadFS provides data crash consistency through CoW. It embeds metadata in the file to avoid kernel crossing for block mapping updates and delivers instant visibility. MadFS employs lock-free optimistic concurrency control to provide high concurrency with cross-process linearizability.

The architecture of MadFS is shown in Figure 2. A MadFS file is a self-contained file on the underlying DAX filesystem. Upon file creation, MadFS creates the file on the kernel filesystem and initializes the basic structure to identify itself as a MadFS file. The following discussion assumes operations on the same file.

Embedded block map (§4.1). We introduce a level of indirection that maps *virtual* blocks seen by applications to *logical* blocks managed by the underlying kernel filesystem. We call this indirection the *block map*. The block map is embedded in the file, which allows MadFS to efficiently handle CoW operations in userspace.

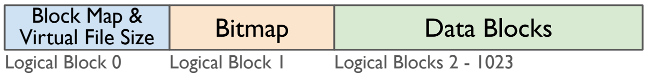


Figure 1: A naive approach for metadata embedding (§4.1)

Compact log-structured metadata (§4.3). To ensure the metadata crash consistency, we maintain the block map as a log persisted in the file. Each block map update is described by a compact 8-byte log entry. For a write operation, MadFS writes to pre-allocated blocks and copies unaligned parts from existing blocks if necessary. MadFS then generates a log entry describing the block map update and finally commits the write by appending the entry to the log. The word-sized (8-byte) log entry ensures the atomicity of the log append and allows for a lock-free concurrency control algorithm.

Lock-free optimistic concurrency control (§4.4). MadFS supports concurrent access to the same file across threads and processes. To achieve high scalability, MadFS employs lock-free optimistic concurrency control (OCC). Concurrent writers do not block each other and are linearized during the log commit. In the case of range overlap, the later writer will detect the conflict during the commit, partially redo the write as needed, and retry the commit. The reader similarly detects overlap and guarantees that it never returns half-written data.

Security. In MadFS, access permission is still enforced by the underlying kernel filesystem during open. To launch an attack, a malicious actor must have permission to write to the file. In this case, the actor could alter the block map, causing others to read the wrong blocks, but this is no different from a traditional filesystem where the actor can directly overwrite file data. For metadata integrity, MadFS treats files as untrusted input and gracefully returns an error on ill-formed files. Furthermore, due to per-file virtualization, the effect of metadata corruption is contained within the file. Similar to other filesystems [12], MadFS does not prevent denial-of-service attacks if the attacker keeps writing to the file.

4.1 Metadata Embedding

To illustrate how metadata embedding allows MadFS to bypass the kernel I/O stack for metadata management, consider a naive design shown in Figure 1. We will later build on this design to add more functionalities.

We denote the blocks backed by the underlying kernel filesystem as *logical* blocks. In this example, the file contains 1024 logical blocks. The first two blocks store metadata; the rest are data blocks, some of which can be unused. We introduce a level of indirection that maps the *virtual* blocks seen by applications to logical data blocks: an application reading the first 4 KB gets the data in the first virtual block, which resides in some logical block. This indirection is maintained in the *block map* as an array of integers. If the virtual block index i maps to logical block index j, then the i-th element of the array is j. The *virtual* file size is the size seen by the applications, and the *logical* file size is the size occupied on the kernel filesystem, which is 4 MB in this example. The bitmap indicates whether a data block is in use or not. Security-sensitive metadata is not embedded and is still managed by the kernel.

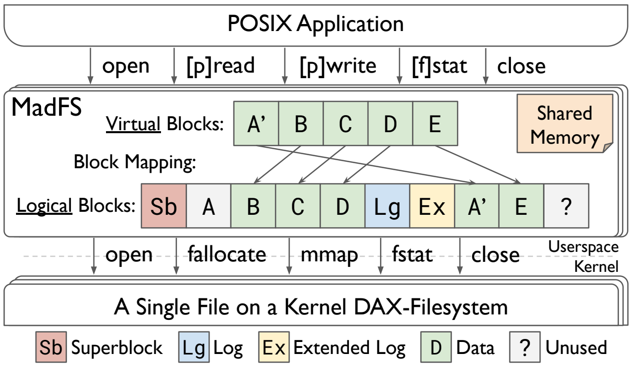


Figure 2: The architecture of MadFS. The application sees *virtual* blocks, which are mapped to the *logical* blocks backed by the kernel filesystem.

The embedding of the block map enables MadFS to perform CoW efficiently in userspace. A write operation proceeds in the following steps: ❶ allocate blocks from the bitmap, ❷ write the user buffer to the allocated blocks and copy unaligned parts of existing blocks if any, ❸ update the block map along with the virtual file size, and ❹ return the old blocks to the bitmap.

In this example, we bypass the kernel I/O stack for write and avoid changing the memory mapping for CoW. However, it only considers a file with fixed logical size (§4.2) and does not ensure metadata crash consistency (§4.3) or enforce concurrency control (§4.4).

4.2 Block Management

To facilitate the dynamic growth of the logical file size, we allow the metadata to be stored anywhere in the file. Figure 2 shows the layout of a MadFS file. The metadata is maintained as a log. We will discuss the log structure in detail in §4.3. For the block layout, there are 5 types of logical blocks:

- Sb *Superblock* is the first block, which contains a magic number that identifies MadFS files and a pointer to the first log block.
- Lg *Log blocks* consist of an array of fixed-size log entries, each corresponding to a metadata update (§4.3). Each log block also carries a pointer to the next one, forming a linked list (Fig. 3).
- Ex *Extended log blocks* store extended log entries, which contain additional information about a metadata update that does not fit into the fixed-size log entry (§4.3).
- D *Data blocks* contain the user data. Each virtual block seen by the application is backed by a logical data block.
- ? *Unused blocks* are blocks that are not referenced by the block map. They appear due to pre-allocation from the kernel filesystem and garbage collection (§4.5).

The rest of this section explains the block allocation mechanism. MadFS stores a per-file bitmap in shared memory for coarse-grained coordination. Each thread maintains a local free list as a cache to avoid frequent accesses to the bitmap. To grow the underlying file from the kernel filesystem, hugepage-aware pre-allocation is used to reduce kernel involvement and minimize page faults.

Per-file bitmap in shared memory. Unlike the example in the previous section (Fig. 1), we no longer persist the bitmap on PM, since we can derive from the log whether a logical block is in use or not. Keeping the bitmap as a soft state is common in log-structured filesystems [37, 45] to simplify crash consistency. We maintain the per-file bitmap information in shared memory to coordinate block allocation across processes. If a process opens a file without a bitmap, it constructs the bitmap according to the log. More details about the shared memory initialization are explained in Section 4.6. Blocks are allocated from the bitmap using atomic compare-and-swap (CAS) instructions for lock-free concurrent operations. This implies that the maximum number of contiguous logical blocks we can allocate at a time is 64.

Thread-local free list. The bitmap is accessed by multiple threads, possibly from different processes. To avoid contention, each thread reserves a free list of blocks. They are not referenced by the block map but are still marked as "taken" in the bitmap. When a thread attempts to allocate new blocks, it first allocates from the local free list; if unavailable, it falls back to the bitmap. When a block is freed, instead of immediately returning it to the bitmap, the block is temporarily kept in the free list. This way, an overwrite-intensive thread keeps reusing the blocks in the local free list and rarely allocates from the shared bitmap. The reserved blocks are returned to the bitmap when the file is closed. In rare cases, a process may crash before the reserved blocks are returned. This results in a temporary leak and the blocks can be reclaimed the next time the bitmap is constructed (§4.6).

Hugepage-aware pre-allocation. So far, the allocation mechanism only guarantees that two threads do not get the same block, but the blocks may not actually be backed by the kernel filesystem. When a block is allocated, the logical block index is returned. Later, when the logical index needs to be converted to a memory address for writing, MadFS checks to see if the block is backed. If not, MadFS calls the `fallocate` syscall to grow the file to a multiple of 2 MB and memory-maps the newly allocated region[2]. The same technique is also used during file creation. Pre-allocation amortizes the cost of kernel involvement, and the choice of 2 MB takes advantage of the huge page support in Linux to reduce page faults and TLB misses. Note that CoW does not break the contiguity of the huge page since it only changes the virtual mapping, which is agnostic to the kernel filesystem.

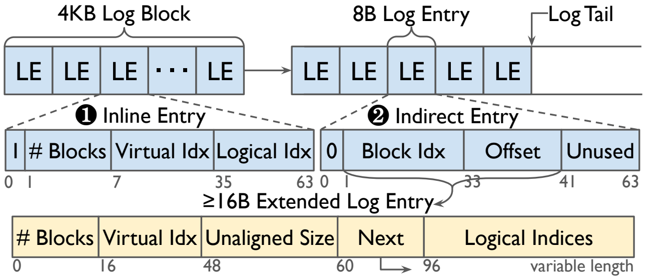


Figure 3: Layout of the log-structured metadata (§4.3). A metadata update is described as either an *inline* log entry or an *indirect* log entry pointing to an *extended* log entry.

4.3 Compact Log-Structured Metadata

In MadFS, a write triggers a block map update, which may span multiple blocks. A write may also expand the virtual file size, which must be modified along with the block map. Therefore, some mechanism is needed to ensure metadata crash consistency. One common choice is journaling. However, journaling is not suitable for non-blocking synchronization because checkpointing requires mutual exclusion. Instead, we structure the metadata as a sequence of log entries, each corresponding to a metadata update. We designed the log entry to be the size of a CPU word (8 bytes) to ensure atomicity and to allow lock-free concurrency control (§4.4).

Log entry layout. As shown in Figure 3, there are two types of 8-byte log entries: ❶ An *inline* log entry is used to represent updates of less than or equal to 64 blocks, which is the maximum number of contiguous logical blocks that the allocator can provide (§4.2). Each inline entry has three fields: a starting virtual block index, a starting logical block index, and the number of blocks the write spans. The three fields together describe a range of virtual blocks mapped to a range of contiguous logical blocks. ❷ An *indirect* log entry is for more complex updates. It carries a pointer to a variable-length *extended* log entry that contains a virtual block range and an array of logical block indices. A write operation with more than 64 blocks will be broken down into multiple allocations, each with a logical index in the extended entry. The unaligned size describes the number of bytes in the last block, which is used to compute the virtual file size. The next field makes it possible to chain multiple extended entries together.

In-memory block table. Because the metadata is now structured as a log, we can no longer directly query the block map and the virtual file size. We use a per-process DRAM data structure called the *block table* to maintain this information. The block table is constructed when a file is opened by scanning through all the log entries. During a read or write, it is queried to obtain the virtual file size and to translate a virtual block index to a logical one. After a new log entry is appended to the log, block table is updated to reflect the metadata update. In the event of a failure, MadFS does not require an explicit recovery phase: the atomicity of the log

[2] `fallocate` and `mmap` are safe to race. `fallocate` is idempotent and commutative. `mapp` supports multiple mappings of the same physical region.

commit is guaranteed by the CPU's 8-byte atomic store, and the log replay during open always puts the block table in a consistent state. We will discuss the concurrency model of the block table later in Section 4.4.

Example. A write operation proceeds in MadFS as follows: ❶ Allocate new data blocks from the local free list or the bitmap. The writer thread also ensures that the allocated blocks are backed by the underlying filesystem and mapped to memory. ❷ Copy the user buffer and unaligned portions to the newly allocated blocks. ❸ Prepare a log entry describing the block map changes. ❹ Append the entry to the log to publish this write. ❺ For overwrites, return the old data blocks to the local free list for recycling.

4.4 Lock-Free Concurrency Control

MadFS supports cross-process sharing with immediate visibility and guarantees linearizability under concurrent access. To achieve these goals, MadFS uses optimistic concurrency control (OCC) [29]. In this section, we explain the concurrency model of the block table, introduce our lock-free OCC protocol, and then discuss the benefits of OCC.

Concurrency model of the block table. The block table is shared across threads within the same process and operates in a single-writer multi-reader manner. For cross-process visibility, before any data operations, MadFS first checks if the log tail has been moved by other processes. If so, it applies newly committed entries to the block table to keep it up-to-date. Within a single process, only one thread can apply new entries at a time, since this procedure would not benefit from having multiple threads doing the same job. Querying the block table is non-blocking, but the thread may see an inconsistent block table if another thread is concurrently updating it. This is not a problem, since such inconsistency can be caught by our OCC protocol described below.

Lock-free OCC. In database literature [29, 48], an OCC protocol typically takes place in the following four phases:
1. *Begin*: Record the begin timestamp for later validation.
2. *Execute*: Read and modify data in a private workspace.
3. *Validate*: Check if data read have been modified by others.
4. *Commit*: Publish the modified data to make them visible.
Compared to lock-based concurrency control, OCC avoids locking the data during the execution phase. However, the last two phases must be executed in a critical section to avoid race conditions, and locks are still used to protect the critical section [29, 31, 39, 48]. In MadFS, the log-structured metadata makes it a good fit with OCC. The monotonically increasing log tail naturally serves as a timestamp. The word-sized log entry can be committed atomically to the tail via compare-and-swap (CAS), ensuring the atomicity of the validate and commit phases and making the OCC protocol lock-free.

Concurrent writers. A writer first updates the block table and records the current log tail for later validation. It then performs CoW and generates an 8-byte log entry. The writer thread attempts to commit the entry to the recorded tail via CAS. If the recorded tail still points to an empty entry, then the CAS can successfully commit the entry. Otherwise, the tail has been moved, and the current thread needs to check for conflicts. A log entry conflicts with the current one if it modifies the unaligned parts copied during CoW. If there is no conflict, the thread simply recommits the log entry to the new tail. Otherwise, the writer recopies the unaligned parts modified by the conflicting log entry and recommits. Note that the unaligned parts copied do not exceed two blocks.

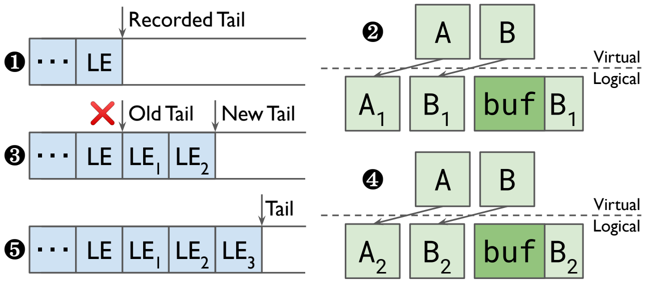


Figure 4: Concurrent writers example (§4.4). Each 🟦 represents an 8-byte log entry. Extended log entries are omitted. Each 🟩 represents a 4 KB data block.

Concurrent readers. A reader also starts by updating the block table and recording the log tail. The reader then copies these blocks to the user buffer. After the copy, if the tail has moved and the added log entries overlap with the range the current thread is reading, the current thread needs to copy the data again. Since the old data blocks are immediately recycled during write (§4.2), the reader must validate up to the latest log tail, so that the blocks read holds valid data.

Example. Figure 4 shows an example of concurrent writers. Suppose a file starts with two virtual blocks A and B with initial contents A$_1$ and B$_1$. A writer wants to `pwrite` 6 KB of data at offset 0 while other threads are concurrently writing to the same file. ❶ We first update the block table and record the current log tail. ❷ The writer does a CoW and generates a log entry to commit. ❸ When the thread tries to commit to the recorded log tail via CAS, it finds that the tail has been moved. ❹ Suppose LE$_1$ remaps block A to A$_2$ and LE$_2$ remaps B to B$_2$. Although both log entries overlap with the current write, the thread only needs to recopy the unaligned part of B$_2$. There is no need to recopy block A since it will be completely overwritten. ❺ The current thread successfully commits the log entry to the latest tail at LE$_3$. ❻ Later, the block table will be updated to reflect the changes in the block map.

Discussion. The non-blocking design ensures that a halted process will not prevent other processes from making progress [19, 20]. This simplifies error handling and eliminates the need to detect process crashes. In addition, this design can provide better concurrency than fine-grained locking because it allows concurrent writers to be non-blocking even if their ranges overlap. Multiple writers can operate on their private blocks in parallel. The order of the operations is linearized during CAS, and conflicts are resolved at the bounded cost of copying 2 blocks. On the other hand, the most fine-grained byte-range locks would not allow them to execute concurrently. Note that the OCC protocol also guarantees system-wide progress, and is thus lock-free [21].

Offset-dependent operations. For concurrent I/O operations, offset-independent calls (e.g., `pread`/`pwrite`) are preferred over offset-dependent ones (e.g., `read`/`write`). However, MadFS still guarantees linearization for offset-dependent operations. MadFS uses a per-process ordered queue: a thread performing an offset-dependent operation adds itself to the queue before proceeding to read/modify the file offset. The order in this queue represents a serial order. When the thread finishes reading or writing the data, it must wait for the previous thread in the queue to finish before committing itself. The whole operation is still optimistic, and the CoW is done in parallel.

4.5 Non-Blocking Garbage Collection

To prevent the log from growing indefinitely, we designed a garbage collector (GC) program to clean up the log. MadFS supports non-blocking GC, which does not block concurrent readers or writers.

Creating a new log. Recall that the log blocks are organized as a linked list with the superblock pointing to the head. This design allows us to use the read-copy update (RCU) [32] technique for non-blocking GC. GC replays the log up to before the currently active block and constructs another linked list of log blocks along with the associated extended log entries. The last block in the new linked list points to the currently active block. Finally, we publish the new log by a CAS on the log head stored in the superblock. A later process that opens the file will use the new log.

Reclaiming the old log. GC cannot recycle the old log immediately because some threads may still be using it. One possible solution is to wait until the next time the bitmap is rebuilt and the space for the old blocks is reclaimed. However, this does not work for long-running processes, which prevents the shared bitmap from being rebuilt. Reference counting the log block is not safe because a process can crash without decrementing the counter.

Our solution is to let each thread report the log block it is currently reading to the shared memory. GC can safely recycle a log block if it is not referenced by any reported log blocks since the block will never be accessed in the future. The reported log blocks and their (direct and indirect) successors cannot be immediately recycled. We call them "orphans" as they no longer have a reference from the log head. To free them in the future, we chain the orphans into a new linked list by adding a `next_orphan` field to each log block in addition to the existing `next` pointer. The head of the orphan linked list is persisted in the superblock. The next time the GC runs, it checks to see if any of the orphan log blocks can be freed following the same rule above.

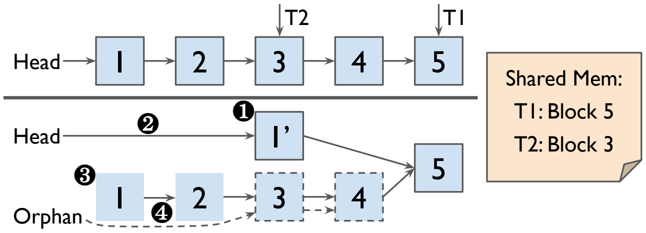


Figure 5: Garbage collection example (§4.5). Each □ represents a 4 KB log block. The `next` pointer is represented by →, and the `next_orphan` pointer is represented by ⇢. Extended log blocks are omitted.

Handling thread crashes. The logical index published in the shared memory is expected to be removed when a thread exits, but a thread may crash before clearing it. We solve this by associating each index with a robust mutex [27] to detect the liveness of the thread. The mutex is locked when the thread is created and unlocked when the process exits or crashes. The GC will try to lock the mutex before accessing the index. Note that the use of the mutex here is only for liveness detection, not mutual exclusion, and no thread is blocked. If GC sees that no thread is currently accessing the file, it can also free the shared memory.

Example. Figure 5 shows an example of garbage collection. There are two I/O threads before GC: T1 is working on the log tail at log block 5, while T2 is behind at 3. ❶ GC reads the current log and creates a new linked list of log blocks 1'→5 with the last block untouched. ❷ The log head pointer is atomically changed to point to the new one. ❸ Blocks 1 and 2 are immediately recycled since all threads have read beyond them. ❹ For the other log blocks up to the tail block, we organize them into an orphan linked list: Orphan⇢3⇢4. GC can free the orphans next time when thread T1 moves on to later log blocks.

Discussion. Concurrent readers and writers are never blocked by the garbage collector. An I/O thread only needs to infrequently update a value in the shared memory when it moves to the next log block. Therefore, the impact on the tail latency is minimal. With the compact log format, we expect the log growth to be slow as a single 4 KB log block can store 510 log entries. As a result, GC runs infrequently.

4.6 Implementation

MadFS is implemented in 4.2K lines of C++ code. It supports 24 POSIX functions, including `[f]open`, `[f]close`, `[p]read`, `[p]write`, `mmap`, `fsync`, `lseek`, `stat`, `unlink`, and `rename`. The rest of this section presents implementation details of MadFS.

Shared Memory. The per-file shared memory is created with the same permission as the file. Its name consists of the inode number and the file creation timestamp for uniqueness. The shared memory stores the bitmap (§4.2) and the current-reading log block index (§4.5). When a process opens a file, it tries to memory map the shared memory. If it does not exist, the process reconstructs the bitmap from the log. The shared memory is removed when the file is removed, the garbage collector cleans up, or the operating system cleans up after the user logs out.

Persistence and ordering. We use the non-temporal `memcpy` from PMDK to copy data to persistent memory, bypassing the CPU cache. We use `clwb` to write the log back to PM without flushing the cache, as they may soon be read by other threads. Memory fences are used to ensure that the data blocks are made persistent before log entries, and that extended entries are persisted before indirect entries.

Decoupling of persistence and ordering. Since each log entry only takes 8 bytes, flushing the entire cache line on each log commit is costly. Instead, MadFS only flushes a cache line when a writer attempts to write to the first log entry of the next line[3]. With an explicit `fsync` call, the last cache line written is flushed to ensure durability, which is similar to `dsync` proposed in OptFS [6]. The ordering of writes is always guaranteed by the memory fence of CAS. Note that at most 8 writes are not persistent without any `fsync`.

Handling `mmap` calls. We support `mmap` using a sequence of `mremap` calls to map the data blocks to a contiguous region of memory. This implementation is not optimized for performance and does not provide a crash consistency guarantee.

Correctness. We use continuous integration for correctness testing on a per-pull-request basis. MadFS passes all 209 test cases in the LevelDB test suites, which make extensive use of checksums and put a heavy load on the filesystem. We use Intel's pmemcheck [38], a fork of Valgrind [34] for PM, to validate the durability of stores made to the PM. We also compile MadFS with Clang Sanitizers [16] to check for data races, memory problems, and undefined behavior.

Conversion tool. We implement a tool to convert files between the MadFS format and the normal file format. Converting a file to a MadFS format is fast. The tool allocates some unused blocks, relocates the first data block to make space for the superblock, and then initializes the superblock. It then commits two log entries to describe the block map: one for the relocated data block and one for the rest. To convert a MadFS file to a normal file, the tool grows the file by the virtual file size, dumps the data blocks in their virtual order, and then calls `fallocate` with the `FALLOC_FL_COLLAPSE_RANGE` flag to deallocate all the blocks previously occupied by MadFS.

5 Evaluation

In this section, we present the experimental results of microbenchmarks and macrobenchmarks. We demonstrate the completeness, performance, and scalability of MadFS by answering the following questions:

- What is the single-thread performance of MadFS? (§5.1)
- Does MadFS scale to multiple threads? (§5.2)
- What is the overhead of open in MadFS? (§5.3)
- Does garbage collection affect tail latency? (§5.3)
- How does MadFS perform on real-world applications (§5.4)

Setup. Our experiments are performed on an Intel x86 machine with a 128 GB Optane DC persistent memory DIMM. The machine is equipped with two Intel Xeon Silver 8-core 4215R CPUs at 3.20 GHz (with 2 hyper-threads for each physical core) and 32 GB of DDR4 memory. We use Ubuntu 22.04 with custom-built Linux kernel 5.1 with NOVA [44,45] and SplitFS [25] included. For all experiments, we pin threads to the core, disable CPU frequency scaling, and drop the kernel cache before each run.

We compare MadFS (on ext4-DAX) to ext4-DAX, SplitFS, and NOVA. Ext4-DAX does not provide data crash consistency. We run SplitFS in the default POSIX mode, which provides a similar crash consistency guarantee as ext4-DAX. In this mode, SplitFS performs overwrites in-place; for appends, it redirects data to a staging file and invokes `relink` system call to update the block mapping on `fsync`. NOVA is a kernel filesystem that uses CoW for data and maintains log-structured metadata. Among the four filesystems, only NOVA and MadFS provide strong data crash consistency.

5.1 Single-Threaded Microbenchmark

To evaluate the baseline performance of MadFS, we designed six microbenchmarks to measure single-threaded throughput under different I/O sizes and access patterns. All operations are repeated 10,000 times, and all writes are followed by `fsync`. Figure 6 shows the results.

Read. For the read experiment, we measure how long it takes to read data under different I/O sizes. MadFS and SplitFS achieve the best performance since the data is served directly from userspace, with most of the time spent on the memory copy. NOVA and ext4-DAX are slower since they need to go through the kernel storage stack. For large read sizes, the difference between NOVA and MadFS becomes small as the kernel overhead is amortized.

Block-aligned overwrite. In both sequential and random cases, MadFS sustains a stable throughput of 2 GB/s for all I/O sizes. ext4-DAX and NOVA do not saturate the device bandwidth due to software stack overhead. ext4-DAX spends non-trivial time on locks (`dax_read_unlock`) and metadata journaling (called in `ext4_iomap_begin/end`). NOVA performs block allocation during CoW with metadata journaling.

[3] The time to the flush cannot be after the last slot of a cache line has been written, since a writer could crash after CAS but before a flush is called.

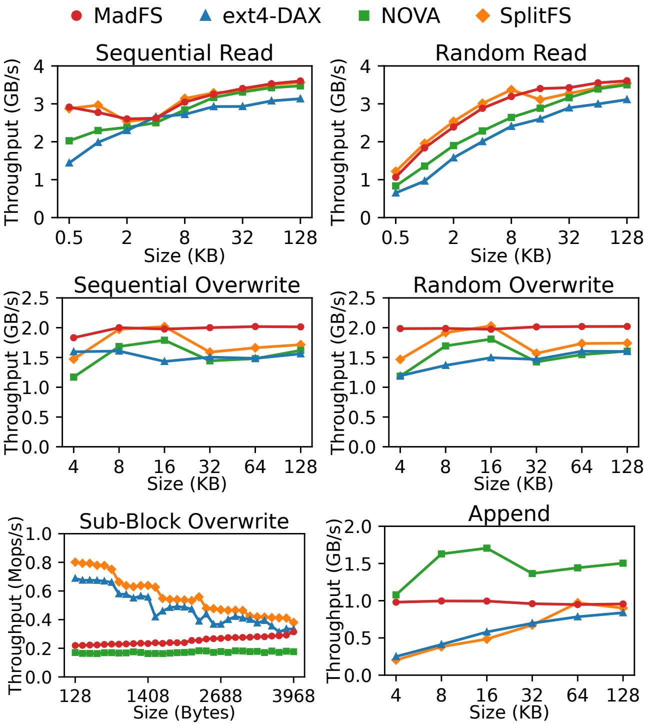


Figure 6: Single-threaded performance. Note for sub-block overwrite, we report throughput in Mops/s instead of GB/s.

SplitFS performs in-place overwrites and does not call the relink system call in this experiment.

Sub-block overwrite. For this experiment, we issue sub-block overwrites and report the throughput in Mops/s. MadFS and NOVA employ CoW for data crash consistency and both show an increase in throughput as the write size increases. This is because with a total of 4 KB to be written to the PM, when the size is larger, more data are copied from the user buffer and fewer from the slower PM. Compared with NOVA, MadFS is 30% to 60% faster in terms of throughput with a 1.5*µs* latency margin. SplitFS and ext4-DAX perform in-place overwrites and do not provide a strong data crash consistency guarantee.

Append. For MadFS and SplitFS, the two userspace filesystems running on ext4-DAX, the peak performance does not exceed 1 GB/s, which is half of the throughput for overwrites. This is due to the block allocation zero-out in ext4-DAX. When the userspace filesystem expands the file size via fallocate, ext4-DAX reserves the blocks to the file. With memory-mapped I/O, the first access triggers a page fault, which causes the kernel to zero out the blocks [26]. These blocks will soon be overwritten by the user data, which halves the effective bandwidth. This is a fundamental issue for userspace filesystems since un-zeroed blocks cannot be exposed directly to the user for security reasons.

NOVA as a kernel filesystem designed for PM does not have this issue and exhibits similar performance to the overwrite experiment. ext4-DAX should not have this issue. However, it reuses a similar code path with the page fault handler and still zeroed out the blocks (called in ext4_map_blocks) before writing to them (in dax_copy_from_iter). For small I/O sizes, SplitFS exhibits similar low throughput as ext4-DAX, since each fsync triggers a relink system call to change the file extent, which involves expensive metadata journaling and inode locking.

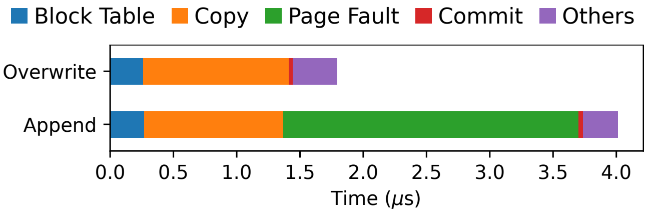


Figure 7: Latency breakdown for 4 KB overwrite and append.

Latency breakdown. Figure 7 shows the time breakdown for 4 KB overwrite and append. Updating the block table involves reading the log entry and applying the changes to the block table. Both overwrite and append take 250 ns on the block table, which is about the same as the latency of accessing 8 bytes from PM. It takes about 1 *µs* to copy the data from DRAM to PM via non-temporal stores. For append, 58% (2.3 *µs*) is spent on the kernel zeroing out. Log commit is as quick as 33 ns, which is about the same latency as a CAS. Others include block allocation, offset calculation, address translation, and block deallocation.

5.2 Multi-Threaded Microbenchmark

In this section, we aim to measure how well MadFS scales when multiple threads access the same file concurrently. We pre-fill a 1 GB file and launch a varying number of threads to read/write the file with offset given by a uniform or Zipfian distribution.

Mixed reads/writes with uniform offset. In this experiment, each thread reads or writes 4 KB at block-aligned offset sampled uniformly at random. With a file size of 1 GB, the probability of two threads operating on the same block is relatively low. Figure 8 shows the result of this experiment. MadFS surpasses other filesystems in all four read-write mixes. Most notably for pure writes, MadFS saturates the device bandwidth at a single thread and sustains the high throughput with more threads. Other filesystems use lock-based concurrency control at inode granularity. SplitFS incurs a performance drop from 1 thread to 2 threads and gradually decreases with more threads. For 95% read, MadFS scales well. It reaches its peak at 11 threads, which matches the device characteristic of the Optane DIMM [47]. SplitFS scales until 6 threads. With more threads, the contention becomes more severe and the throughput drops. With pure read, all filesystems perform well since read operations do not conflict with each other.

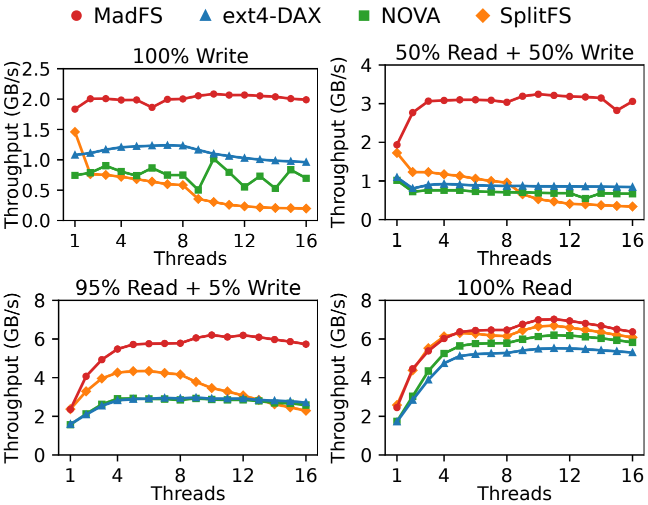


Figure 8: Councurrent 4 KB read/write with uniform offset.

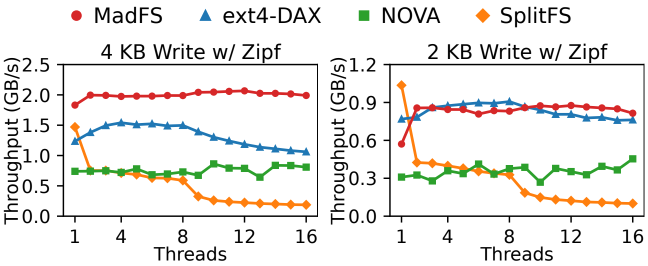


Figure 9: Councurrent pure write with Zipfian offset (θ = 0.9).

Writes with Zipfian offset. To investigate how block-level contention affects scalability, we designed the Zipfian experiments. Each thread writes 4 KB or 2 KB at a block-aligned offset sampled from a Zipfian distribution of θ = 0.9, which results in an access pattern skewed to the first few blocks. Figure 9 shows the result. With 4 KB block-aligned write, the result is similar to the 100% uniform write (Figure 8). The OCC algorithm used by MadFS does not block concurrent threads even if they write to the same block. The order of concurrent writers is linearized during the commit. Since the write is block-aligned, when the commit failed, MadFS only needs to recommit the 8-byte log entry to the new tail and never recopies data (§4.4). Other filesystems use locks at inode granularity, so they do not show significant performance differences between uniform access and Zipfian access. For 2 KB writes, MadFS and NOVA uses CoW and the thread needs to recopy the 2 KB unaligned portion from the new block if newly committed writes overlap with the current one. Nevertheless, MadFS still achieves better performance compared to NOVA. ext4-DAX shows contention with more threads and performs worse than MadFS after 8 threads. Note that only NOVA provides the same strong crash consistency guarantee as MadFS.

Concurrency control. In addition to OCC (§4.4), we experiment with three lock-based concurrency control methods for MadFS and compare their performance under mixed read-write 4 KB workload with uniform block-aligned offset. Spinlock is completely in userspace and cannot handle lock-owner crashes in the cross-process scenario. Mutex is set to be robust so the kernel will release it when the owner dies. Reader-writer lock does not support the robustness feature. Only mutex provides the same robustness guarantees as OCC.

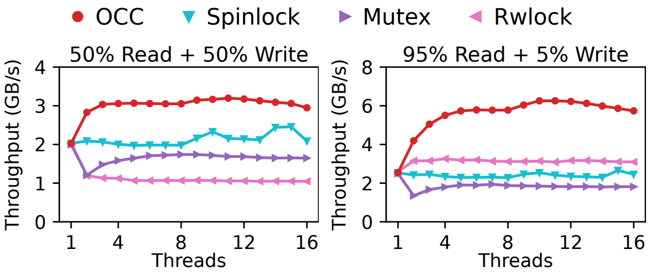


Figure 10: MadFS with different concurrency control methods under uniform 4 KB read/write.

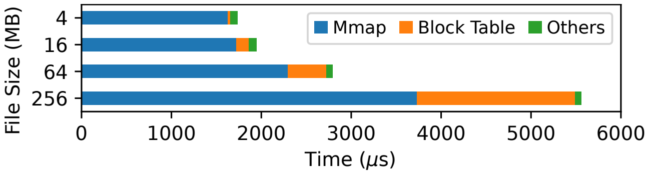


Figure 11: Open latency breakdown. The file size is logical.

Figure 10 shows the result of this experiment. In both workloads, all four concurrency control methods start at the same throughput with a single thread, and OCC surpasses the lock-based concurrency control methods with more threads by a wide margin. With OCC, multiple writers can write to thread-private blocks concurrently without blocking other readers or writers, thus yielding better scalability. The performance of mutex drops from one thread to two threads since mutex puts threads in sleep under contention. Spinlock performs better than mutex as it busy-waits for the lock owner. Reader-writer lock is at the bottom for the 50% read workload due to its operation complexity, but it outperforms spinlock and mutex for the 95% read workload as readers do not block each other.

5.3 Metadata Operations

Open. During file open, in addition to the open system call, MadFS need to memory-map the file and replay the log to build the block table. Memory mapping a file takes a fixed cost of 1616 μs plus 17 μs per 2 MB huge page. The same overhead applies to other userspace PM filesystems as well. The log replay is efficient due to the compact log format, taking only 15 ns for an inline entry and 21 ns for an indirect one (with a 16-byte extended entry).

Figure 11 shows the time breakdown to open a file created by repeated 4 KB appends. The majority of the time is spent on memory-mapping the file, especially for small and medium-sized files. Other times include the open system call. Due to the open overhead, MadFS may not be suitable for workloads with frequent file opens.

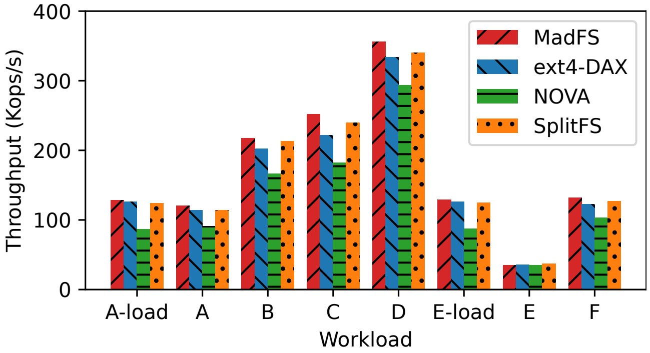


Figure 12: Throughput YCSB workloads on LevelDB.

Garbage Collection. In this experiment, we aim to measure the effect of the GC on tail latency. We have a writer thread repeatedly doing 4 KB overwrite to a 1 GB file. A GC thread runs every 30 seconds to collect old log entries. The average runtime for GC is 9.1 ms, which is 0.03% of the writer's runtime. With GC, the 99.9%, 99.99%, and 99.999% tail latencies for the writer are 5.06 *µs*, 6.46 *µs*, and 20.77 *µs* respectively, compared to 5.05 *µs*, 6.12 *µs*, and 20.18 *µs* without GC. Overall, the GC finishes quickly and imposes negligible overhead on the I/O thread.

5.4 Real-World Applications

LevelDB with YCSB. To show the completeness of MadFS implementation, we run LevelDB [17], a key-value store based on log-structured merge (LSM) trees. We run the YCSB benchmark [46], a common cloud benchmark for database applications. The benchmark includes 6 workloads: A (50% read + 50% update), B (95% read + 5% update), C (100% read), D (95% read + 5% insert), E (5% insert + 95% scan), and F (50% read + 50% read-modify-write). We issue 1 million operations with a value size of 1 KB.

Figure 12 shows the throughput of all YCSB workloads on LevelDB across four filesystems. The overall trend is MadFS > SplitFS > ext4-DAX > NOVA. For read workload C, MadFS outperforms SplitFS, ext4-DAX, and NOVA by 5%, 12%, and 28% respectively. For write-heavy workloads F, the improvements of MadFS over the other three 4%, 7%, and 22% in the same order. All four filesystems perform similarly on workload E as it has most of the data cached in the memory and is not I/O intensive.

SQLite with TPC-C. SQLite is a widely-used relational database management system [22]. It is used as a library embedded into the end program and stores the entire database as a single file on the filesystem. We drive SQLite with TPC-C, an online transaction processing (OLTP) benchmark that simulates order processing in a multi-warehouse wholesale system [11]. TPC-C includes a mix of 5 transaction types: new order, payment, order status, delivery, and stock level. Each transaction involves a series of SQL statements. We run the TPC-C benchmark using the default configuration: 4 warehouses, 1 district, and 200,000 transactions. The size of the resulting database is 444 MB. The implementation of this benchmark is adopted from SplitFS.

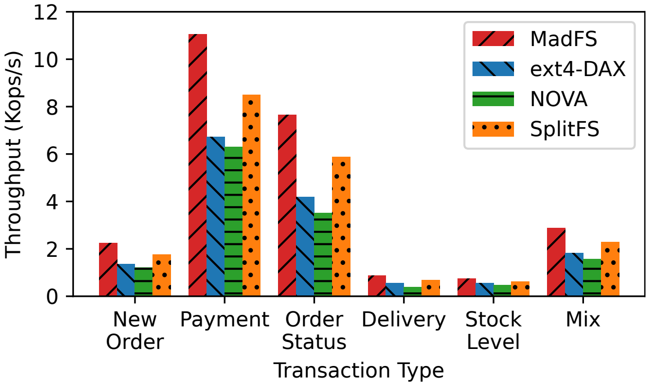


Figure 13: Throughput of TPC-C workloads on SQLite.

Figure 13 shows the throughput for each of the individual transaction types and the mixed workload. MadFS outperforms other filesystems for all types of transactions since writes in SQLite are mostly block-aligned and do not incur CoW for MadFS. On the mixed workload, MadFS is 26% faster than SplitFS, 58% faster than ext4-DAX, and 85% faster than NOVA.

6 Conclusion

In this paper, we present per-file virtualization which aims to push file functionalities into userspace as much as possible. Metadata embedding allows kernel-bypassing for metadata management. In particular, embedding the block mapping enables efficient userspace CoW for crash consistency. Non-blocking synchronization enables scalable, crash-safe concurrency control without kernel involvement. Based on per-file virtualization, we implement MadFS, a library PM filesystem that maintains embedded metadata as a sequence of compact log entries and employs optimistic concurrency control for linearizability. Our evaluation shows that MadFS yields better performance than ext4-DAX, NOVA, and SplitFS.

Acknowledgments

We are grateful to our shepherd Randal Burns and the anonymous reviewers for their valuable feedback and comments. This material was funded by NSF grants CNS-1838733, CNS-1815656, and CNS-1900758, and supported by gifts from Google, PingCAP, Seagate, and VMware. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or any other institutions.

References

[1] Intel® optane™ persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[2] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 27–39, 2017.

[3] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down TLB shootdowns! In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[4] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. Fine-grained metadata journaling on nvm. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13. IEEE, 2016.

[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 *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 81–95. USENIX Association, February 2021.

[6] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243, 2013.

[7] Dave Chinner. xfs: DAX support. https://lwn.net/Articles/635514/. Accessed: 2021-01-13.

[8] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 1–16. USENIX Association, July 2020.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 133–146, New York, NY, USA, 2009. Association for Computing Machinery.

[10] Intel Corporation. Intel Reports Second-Quarter 2022 Financial Results. https://www.intc.com/news-events/press-releases/detail/1563/.

[11] Transaction Processing Performance Council. TPC-C: an On-Line Transaction Processing Benchmark. http://www.tpc.org/tpcc/. Accessed: 2021-01-12.

[12] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 478–493, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, 2017.

[14] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[15] Gregory R Ganger and Yale N Patt. Metadata update performance in file systems. In *OSDI*, volume 94, pages 148–159, 1994.

[16] Google. Google Sanitizers: AddressSanitizer, MemorySanitizer, ThreadSanitizer, LeakSanitizer, and more. https://github.com/google/sanitizers. Accessed: 2021-01-12.

[17] Google. google/leveldb: LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values. https://github.com/google/leveldb, 2011.

[18] Daniel Hackenberg, Daniel Molka, and Wolfgang E Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on microarchitecture*, pages 413–422, 2009.

[19] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[20] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[21] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 522–529. IEEE, 2003.

[22] D. Richard Hipp. SQLite Home Page. https://www.sqlite.org/index.html. Accessed: 2021-01-12.

[23] Dave Hitz, James Lau, and Michael A Malcolm. File system design for an nfs file server appliance. In *USENIX winter*, volume 94, pages 10–5555, 1994.

[24] 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 SIGOPS 28th Symposium on Operating Systems Principles*, pages 804–818, 2021.

[25] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Linux kernel development community. ext4_issue_zeroout identifier - Linux source code - Bootlin. https://elixir.bootlin.com/linux/v5.18.14/source/fs/ext4/inode.c#L417. Accessed: 2021-01-13.

[27] Linux kernel development community. Pthread_mutexattr_setrobust(3) - linux manual page. https://man7.org/linux/man-pages/man3/pthread_mutexattr_setrobust.3.html. Accessed: 2021-01-12.

[28] Linux kernel development community. The Linux Journalling API. https://www.kernel.org/doc/html/latest/filesystems/journalling.html.

[29] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, jun 1981.

[30] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Epoch-based commit and replication in distributed OLTP databases. 2021.

[32] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, 1998.

[33] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *2015 44th International Conference on Parallel Processing*, pages 739–748. IEEE, 2015.

[34] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[35] 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 Conference on Operating Systems Design and Implementation*, OSDI'14, page 433–448, USA, 2014. USENIX Association.

[36] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can applications recover from fsync failures? In *The 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[37] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[38] PMDK team at Intel Corporation. Pmemcheck - persistent memory analyzer. https://pmem.io/valgrind/generated/pmc-manual.html. Accessed: 2021-01-12.

[39] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[40] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 340–349. IEEE, 2011.

[41] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael Swift. Aerie: Flexible file-system interfaces to storage-class memory. *Proceedings of the 9th European Conference on Computer Systems, EuroSys 2014*, 04 2014.

[42] Matthew Wilcox. DAX: Page cache bypass for filesystems on memory storage. https://lwn.net/Articles/618064/, 10 2014. Accessed: 2021-10-22.

[43] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 427–439, 2019.

[44] Jian Xu and Steven Swanson. NOVA is a log-structured file system designed for byte-addressable non-volatile memories, developed at the University of California, San Diego. https://github.com/NVSL/linux-nova. Accessed: 2021-01-12.

[45] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[46] Yahoo. Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB/, 2010.

[47] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 169–182, 2020.

[48] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.

On Stacking a Persistent Memory File System on Legacy File Systems

Hobin Woo
Samsung Electronics

Daegyu Han
Sungkyunkwan University[*]

Seungjoon Ha
Samsung Electronics

Sam H. Noh
UNIST and Virginia Tech[†]

Beomseok Nam
Sungkyunkwan University

Abstract

In this work, we design and implement a Stackable Persistent memory File System (*SPFS*), which serves NVMM as a persistent writeback cache to NVMM-oblivious filesystems. SPFS can be stacked on a disk-optimized file system to improve I/O performance by absorbing frequent order-preserving small synchronous writes in NVMM while also exploiting the VFS cache of the underlying disk-optimized file system for non-synchronous writes. A stackable file system must be lightweight in that it manages only NVMM and not the disk or VFS cache. Therefore, SPFS manages all file system metadata including extents using simple but highly efficient dynamic hash tables. To manage extents using hash tables, we design a novel Extent Hashing algorithm that exhibits fast insertion as well as fast scan performance. Our performance study shows that SPFS effectively improves I/O performance of the lower file system by up to 9.9×.

1 Introduction

Non-volatile main memory (NVMM) has low access latency and byte-addressability similar to DRAM but ensures non-volatility of data similar to secondary storage. Intel's DC Persistent Memory module (DCPMM) is one of the first commercialized NVMM products, which provides exciting performance as storage class memory (SCM). Despite its shortcomings such as (i) latency higher than DRAM, (ii) bandwidth lower than DRAM, (iii) high sensitivity to NUMA effects, and (iv) a larger media access granularity (i.e., 256-byte XPLine), extensive research have been conducted to explore the desirable features of DCPMM, i.e., persistency with much lower latency than NVMe SSDs [7]. While the future of DCPMM is uncertain in short term [31] due to the recent Intel's unfortunate decision to shut down its Optane business, nevertheless, DCPMM has left various positive legacy, including NVMM-aware file systems [24, 37, 39] and key-value stores [11, 20, 21, 23, 34, 36]. Although such systems are still in their infancy, they have shown the potential to significantly outperform legacy systems and thus, other types of NVMM (e.g., MRAM and battery-backed DRAM) are likely to succeed DCPMM in the near future. However, the biggest weakness of current developments such as MRAM and battery-backed DRAM devices is their limited capacity. As such, for the immediate future, small NVMMs are expected to be used in conjunction with traditional storage devices. In this paper, we present a file system that can be deployed with only a relatively small amount of NVMM harnessing the benefits of NVMM, while, at the same time, continuing to make use of the underlying conventional file systems for block storage devices.

Previous studies have attempted to develop *monolithic file systems* that manage both NVMM and block device storage and that determine which device to service the read and write requests based on the I/O characteristics [24, 39]. However, managing multiple storage device types with a single, monolithic file system has its limitations. First, monolithic file systems for tiered storage devices, such as Ziggurat [39] and Strata [24], are hard to tailor for various combinations of multiple block device types. Second, developing a file system from scratch takes considerable time and effort to mature into a stable file system. Moreover, managing multiple tiered storage devices adds even more complexity. Third, from a deployment point of view, monolithic file systems cause a bit of inconvenience as they are oblivious of existing file systems; To deploy these systems in practice, a backup of the enormous number of files managed by legacy file systems must first be made, then the new NVMM and disk setting formatted, and then the backup copied back.

In this paper, we advocate a modular approach through the use of stackable file systems (aka overlay or union file systems) [9, 14, 29, 38]. Specifically, we present *SPFS (Stackable PM File System)*, a stackable file system that can be deployed with only a relatively small amount of NVMM, whose goal is to absorb frequent small synchronous writes required to maintain storage write order. For example, modern I/O stack enforces log entries and commit marks to be flushed to durable storage devices in serialization order such that recovery is pos-

[*] Department of Electrical and Computer Engineering
[†] This work was done at UNIST.

sible. For this, conventional file systems interleave small write requests with expensive `fsync()` system calls, which leads to performance degradation. The primary goal of SPFS is to let NVMM absorb such synchronous small writes and reduce the overhead of enforcing durability in block device file systems.

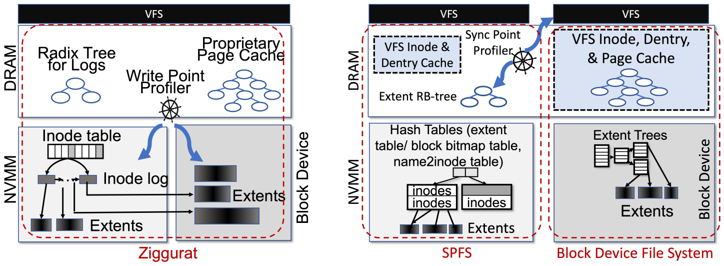
In addition, the NVMM-optimized ("upper") SPFS file system can be stacked on top of any other block device-optimized ("lower") file system *x* to provide a file system that is a union of both. Such a modular approach allows SPFS+*x* file system configurations that provide the best aspects of both NVMM and conventional devices as well as file systems for these devices. More specifically, aside from the performance benefits aforementioned, higher stability as well as flexibility can be attained. This is because we can exploit as the lower file system, any mature file system, e.g., EXT4, XFS, or F2FS, allowing delegation of large or asynchronous writes to the lower file system that can benefit from the highly efficient VFS cache. Furthermore, as SPFS is specifically designed and implemented to absorb frequent small synchronous writes in NVMM, its logic is simple and thus, easy to verify. Also, our modular approach is easier to deploy than monolithic file systems for tiered storage because SPFS can be stacked on any production file system on the fly. This makes deploying and taking advantage of NVMM simple.

In return for these advantages, stackable file systems may double the file system management overhead as it is layering two file systems. Therefore, a stackable file system must be lightweight. To this end, SPFS manages file system metadata in lightweight and efficient hash tables using a novel hashing algorithm that supports efficient lookup as well as scans.

The main contributions of this study are as follows.

- We design and implement SPFS, a stackable file system that allows any kernel file system *x* to reap the performance of NVMM while requiring no changes to *x*.
- SPFS, and its resulting SPFS+*x*, allows leveraging of the strengths of each storage device type, i.e., asynchronous writes of the VFS cache (DRAM), synchronous small writes of SPFS (NVMM), and various desirable features of disk-optimized file systems (SSDs).
- SPFS manages all file metadata in hash tables that ensure fast insertion and lookup. We also propose a novel *Extent Hashing* algorithm to hash key ranges and support extents in hash-based file mappings.
- Our performance study shows that SPFS+EXT4, SPFS+XFS, and SPFS+F2FS improves the performance of the lower file system by up to 9.9×.

The rest of this paper is organized as follows. In Section 2, we present the background and motivation. In Section 3, we present how SPFS profiles synchronous writes and steers them to NVMM. In Section 4, we present how SPFS manages file system metadata using hash tables. In Section 5, we evaluate the performance of SPFS. In Section 6, we conclude the paper.



(a) Monolithic Tiered File System

(b) Stackable File System

Figure 1: Comparison of Ziggurat and SPFS

2 Background and Motivation

2.1 Stackable File System

File systems often make tradeoffs for a specific type of storage device [24]. For example, F2FS [25] is designed to accommodate the characteristics of NAND flash memory-based storage devices. Leveraging the hardware properties of each storage device has been studied for a long time. Such developments are expected to continue as evolution of storage devices (e.g., ultra low latency NVMes, Zoned Namespace SSDs, CXL devices, etc.) continues [5, 8, 15]. Therefore, we question whether it is desirable to have one file system that rules them all, and also question whether the effort of optimizing legacy block device-only file systems needs to be duplicated for monolithic file systems for tiered storage as well. For instance, Ziggurat does not use the well optimized VFS cache. Instead, it implements its own proprietary page cache as shown in Figure 1(a).

A stackable file system is a lightweight file system that runs on top of another file system. It is often used to change the behavior of the lower file system, e.g., encryption, access control, etc., without its own storage device (e.g., eCryptFS [16]), or to combine two mount points into one to provide a single file system image (e.g., UnionFS [14], OverlayFS [9], and AUFS [29]), such that an immutable Docker container image can be provided as a base and the upper level stackable and mutable file system can overlay new files or directories on top of the base. Wrapfs [38] is a small null-layer (i.e., template) stackable file system from which one could implement a new upper level file system. Wrapfs is an implementation of stackable *vnode* interface [30], which allows multiple vnodes to be chained for a single file. Using the vnode chain, a stackable filesystem can interact with the lower file system via VFS interfaces (e.g. `call_read_iter`) or direct operation calls (e.g. `inode_operations.fiemap`). With the vnode chain, stackable file systems can perform various functionalities, such as encrypting/decrypting files or making copies of data blocks in the upper level storage device to hide the data blocks in the lower level file system.

Consequently, if NVMM is used as an intermediate layer in the storage hierarchy, it is natural to design a *stackable* file system for NVMM that can be layered on a variety of existing block device file systems instead of abandoning the legacy block device file systems that have been improved for decades. We believe there exists an unexplored opportunity of layering file systems optimized for each storage device as this allows one to easily get the most out of each storage device type. However, a stackable file system needs to be lightweight as layering two file systems may double the file system management overhead. As such, we aim to design and implement a lightweight hash-based stackable file system.

2.2 Steering Synchronous Writes

On spinning disk drives, the seek time may exceed the data transfer time if writes are small [18, 19]. Even on SSDs, it has been reported that small random writes fail to leverage the full device bandwidth because small random writes cause a large number of invalid pages to be scattered and valid pages are moved to different blocks via garbage collection. To mitigate such problems due to small writes, which we refer to as the *microwrite* problem, various block device file systems, including BetrFS [18, 19] and VT-tree [32], have been designed to absorb the small writes in a log-structured manner. Other remedies such as preallocation [26], defragmentation [22, 26], and block layer I/O scheduling techniques [35] have also been proposed. SPFS relies on these features of the conventional, lower file systems to address the microwrite problem. We believe handing the microwrite problems over to the DRAM cache in the lower file system is the most effective solution as it liberates SPFS to focus on the synchronization overhead (i.e., order-preserving writes), which cannot be resolved by the volatile DRAM cache.

Applications require synchronization mainly for two purposes - durability and *storage order* [35]. However, enforcing storage order by calling `fsync()` often results in frequent small synchronous writes, which leads to significant performance degradation because it prevents I/O parallelism [35]. SPFS steers this order-preserving synchronous writes to fast and durable NVMM while leveraging the VFS cache of the lower file system for *buffered* writes. As a stackable file system, SPFS does not duplicate the VFS cache to avoid the *double copy* problem [12].

Determining whether each write is synchronous or not is a hard problem. Ziggurat [39] and HiNFS [12] use DRAM as a write-back cache for buffered IO and determine if each write is synchronous or not based on the write size and fsync interval (Ziggurat) or based on the latency of each write (HiNFS), respectively. Both approaches are eager in detecting write types in that they determine the write type for each write.

In this work, we design and implement a lazy *Sync Point Profiler* to determine which blocks are to be placed in NVMM, or in DRAM or block device through the lower file system. By default, SPFS forwards incoming writes to the fast VFS cache first, then triggers block migration if certain conditions are met. This lazy approach benefits more from low DRAM latency, unlike the eager approaches of Zigurat and HiNFS.

2.3 Hash-based Global File Mapping

File mapping structures map logical offsets of a file to physical locations on the underlying device. In most traditional file systems, file mapping tables are tree-structured indexes such as extent trees and radix trees [28]. As the number and size of files increase, the size of the file mapping structures also increases. The resizing operation is particularly expensive in tree-based indexes because any update to internal tree nodes conflict with other concurrent operations that access different leaf nodes. To mitigate this problem, traditional file systems use *per-file* mapping structures to isolate concurrent accesses to different files and reduce lock contention.

In contrast to conventional wisdom, Neal et al. [28] recently show that as tree-based per-file mapping structures suffer from multiple levels of indirection and more memory references, a single hash table to manage global file mappings can be beneficial in NVMM. HashFS, the file system that they propose, requires a much smaller number of memory accesses than tree-based mappings, and as such, the performance of global hashing is shown to outperform per-file extent-trees and radix trees [28]. However, still, there is an unresolved limitation in hash-based global file mapping. That is, *block hashing* that is employed in HashFS is not suitable for sequential I/Os because block hashing does not manage extents. Extent-based file systems allow for files to be laid out contiguously on disk space, making sequential I/O fast. Extents also significantly reduce the amount of metadata by storing only two numbers, the first block number and the number of blocks covered by the extent. However, block hashing requires every block number in an extent to be canonically stored in its corresponding bucket, which slows down sequential I/Os. In this work, we develop a novel *Extent Hashing* algorithm to overcome these limitations, which we describe in Section 4.2.

3 Design of SPFS

SPFS consists of four key components that allow SPFS to be stacked with legacy file systems, as shown in Figure 1(b) - (i) the *Sync Point Profiler* that steers order-preserving small synchronous writes to NVMM, (ii) hash-based extent management (*extent table*), (iii) hash-based free space management (*block bitmap table*), and (iv) hash-based name resolution (*name2inode table*).

In this section, we concentrate on the Sync Point Profiler that determines which blocks are to be handled by SPFS and placed in NVMM or by the lower file system to be placed in conventional storage. The hash-based discussions are presented in Section 4.

3.1 File Block Placement Mode

Figure 2 shows the three file block placement modes supported in SPFS - *standalone*, *bypass*, and *stacked* modes. Note that SPFS places NVMM next to DRAM and disks, rather than in the middle of DRAM and block device hierarchy. SPFS is the upper file system, but only manages NVMM,

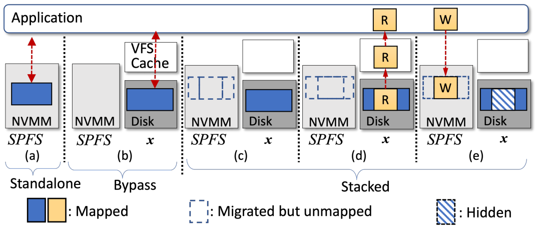
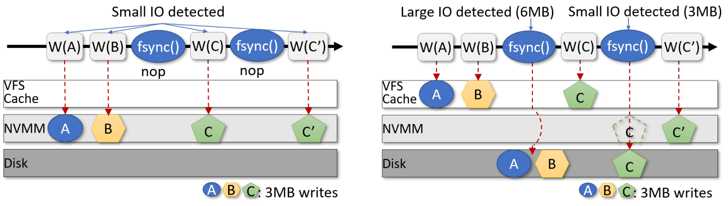


Figure 2: Three File Operation Modes



(a) Write Point Profiler (Ziggurat)

(b) Sync Point Profiler (SPFS)

Figure 3: Write Point Profiler vs. Sync Point Profiler

letting the faster DRAM VFS cache be managed by the lower file system.

Standalone mode: If a synchronous option (O_DIRECT, O_SYNC) is specified for the `open()` system call or if SPFS is used without a lower file system, all file blocks are placed in NVMM as shown in Figure 2(a).

System administrators can also force the use of NVMM on a per-directory basis via extended attribute or `ioctl`. That is, for example, if there is a directory that stores transactional log files or short-lived backup files, the administrator can specify the directory to be used in standalone mode.

Bypass mode: If the synchronous option is not specified for `open()`, NVMM is bypassed by default, placing the file in the lower file system as shown in Figure 2(b). By bypassing writes to the lower file system, read intensive workloads and non-synchronous writes can benefit from the fast VFS cache.

Stacked mode: Figures 2(c), (d), and (e) show the Stacked mode where both SPFS and the lower file system play roles as particular blocks of files are placed in either NVMM or conventional storage. If the *Sync Point Profiler*, to be described in Section 3.2, decides to place a block in NVMM, SPFS gets the file extent information using `fiemap ioctl` and prepares the file mapping in NVMM. In preparing the file mapping, SPFS does not yet physically migrate the extent as shown in Figure 2(c), where the dashed rectangle represents the mapping for the file to be migrated. Physical migration is delayed because the Sync Point Profiler makes the migration decision when `fsync()` is called, i.e., the dirty blocks have already been delegated to the lower file system, and they could have been persisted to disk via periodic write-back of the lower file system. Instead, the migration is deferred until the next write such that `read()` benefits from the low latency of the VFS page cache, as shown in Figure 2(d). Since there is no need to move a migration target block to NVMM unless the block is subsequently updated, physical migration is triggered by subsequent `write()` calls, avoiding the double copy problem [12]. As shown in Figure 2(e), SPFS checks the file mapping and writes blocks in NVMM if the file mapping indicates that the file is mapped in NVMM. By nature of stackable file systems, access to migrated blocks is serviced by the upper file system, that is, SPFS. Thus, the blocks in the lower file system become invisible to the user. When the blocks are actually migrated, the blocks in the lower file system are erased via `fallocate()`. Later, if the entire file is migrated to SPFS, the file is deleted from the lower file system.

3.2 Profiling Mechanism: Sync Point Profiler

Order-preserving small synchronous writes often lead to orders of magnitude IOPS degradation [35] because it serializes potentially parallel activities. Such order-preserving small synchronous writes need to be steered to fast NVMM rather than slow block devices. Therefore, we devise a *Sync Point Profiler* that monitors `fsync()` calls. Specifically, at an `fsync()` call, if the previous `fsync()` call on the same file is within a certain threshold and the amount of flushed data is small, we consider this to be an order-preserving small synchronous write. The rationale behind this is that if the interval is short, there is a high probability that the `fsync()` calls are made with intention to keep the *storage order* [35]. In contrast, if the interval is large, then even if the write size is small, it is unlikely that applications are flushing writes in continued sequence. Thus, these can be serviced by the slower lower file system without much performance degradation. To determine small, we do not take individual write sizes at the point of writes, but take the total number of bytes written to the lower file system at `fsync()`. The rationale behind this is that large writes can benefit from disk bandwidth, and synchronous writes to maintain storage order are usually small (e.g., 4 KB WAL frames in DBMS). The default values in our setting are 1 second for the interval and 4 MB for the size. We take 4 MB as this is the value used in Ziggurat's synchronicity predictor's policy [39], while 1 second was chosen as we observe the performance of SPFS is insensitive to the threshold time unless it is set too small. In Section 5, we quantify the performance effects of the profiler parameters.

Figure 3 highlights the key differences between the Ziggurat's synchronicity and write size predictor and the SPFS Sync Point Profiler. The example in Figure 3(a) where an application issues four 3 MB small writes (`A`, `B`, `C`, and `C'`) shows how Ziggurat makes its decision for each individual `write()` (thus, Write Pointer Profiler) and eagerly persists small writes according to its *fast-first* policy. Its write size predictor will detect the first two writes, `A` and `B`, as small and store them in NVMM. When `fsync()` is called, its synchronicity predictor will detect the total number of bytes is larger than 4 MB and treat the file as an asynchronous file.

Nevertheless, A and B have already been flushed to NVMM. The write size predictor also steers C into NVMM since it is small. The second fsync(), however, considers the file as a synchronous file as only 3 MB (C) was written. Consequently, the next write C′ will also be written in NVMM. In conclusion, we see that all writes are stored into NVMM. As we will show later in Section 5, Ziggurat's profiling method fails to leverage faster DRAM and shows similar performance as the NVMM-only file system NOVA because it aggressively steers most writes to NVMM.

SPFS, on the other hand, makes block placement decisions when fsync() is called. Using the same example as above, Figure 3(b) shows how differently the SPFS profiler services the writes. For A and B, they are initially written to the VFS page cache allowing them to make use of the DRAM. Upon the first fsync(), because the total write size is 6 MB, both A and B are written to the block device via the lower file system. Similarly, C is also written to the page cache. When the second fsync() is called, the lower file system flushes C from the cache to disk, but at the same time, SPFS detects small synchronous writes and migrates its block mapping, (not data blocks), to NVMM. When subsequent writes are requested to some of the blocks of C (C′), these writes are steered to NVMM and directly written onto.

3.2.1 Migration to Lower File system

Compared to Ziggurat, SPFS uses NVMM sparingly. However, when the NVMM space is running low, SPFS selects victim files and migrates them to the lower file system. Note that the primary goal of SPFS is not to cache frequently accessed files but to absorb order-preserving small synchronous writes. Therefore, even if NVMM has free space, SPFS migrates a file to the lower file system if its access pattern changes, e.g., if the access pattern is read intensive, *demoting* the file to the lower file system can benefit from the VFS page cache.

SPFS uses a metric called *Sync Factor* to determine which file's recent I/O pattern is well suited for the criteria of order-preserving small synchronous writes. The formula that calculates the Sync Factor (SF) at time t is given by

$$SF_t = \alpha \cdot weight(IO\_type) + (1-\alpha) \cdot SF_{t-1}$$

where α is the attenuation factor ($0 < \alpha < 1$), i.e., the formula employs exponential moving average to attenuate the effect of old file accesses. $weight(IO\_type)$ is a fixed positive value if the current I/O at time t satisfies the Sync Point Profiler's condition. Otherwise it is zero, i.e., if a file is read-intensive or updated in large units, its Sync Factor gradually decreases. Sync Factor is maintained per file and updated only upon an I/O request. Therefore, its computation overhead is negligible.

When the NVMM space is running low, SPFS migrates the files with low Sync Factor back to the lower file system in the background. Administrators can also set a hard limit on the Sync Factor so that files can be migrated back to the lower file system if their Sync Factors are lower than the hard limit, even if NVMM has free space.

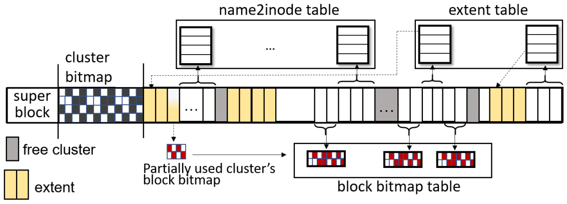


Figure 4: NVMM Space Layout for SPFS

4 Hash-based Block Management

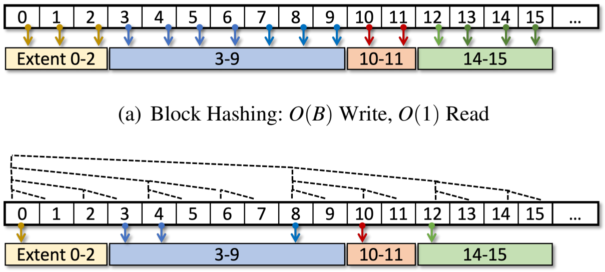
While SPFS is a stackable file system, it is also a standalone hash-based NVMM file system. As a file system, SPFS requires file system metadata to be managed persistently, and thus, metadata management overhead can be doubled. Since the target workload of SPFS is synchronous I/Os to a large number of small files, the conventional per-file mapping structures may waste storage space [28]. Therefore, similarly to HashFS [28], SPFS manages file block mapping information using global hash-based structures. Furthermore, SPFS reduces the size of the hash table by indexing extents, not blocks. However, to the best of our knowledge, no efficient means of hashing extents, i.e., range data, is known. To overcome this limitation, we propose a novel *Extent Hashing* algorithm.

In this section, we describe hash-based free space management (*block bitmap table*), hash-based extent management (*extent table*), and hash-based path-name resolution (*name2inode table*) in SPFS.

4.1 Free Space Management

Using extents, SPFS effectively reduces the aggregate size of file mapping metadata. The aggregate size of the file mapping structures is particularly important for a lightweight stackable file system because small mapping structures leave more room for file data blocks. SPFS employs dynamic hashing (in particular, CCEH [27]) to dynamically adjust the size of the multiple hash tables and efficiently manage the NVMM space. Specifically, if a hash collision cannot be avoided by linear probing or cuckoo hashing, SPFS dynamically allocates and assigns NVMM blocks to each hash table, namely, extent table, block bitmap table, and name2inode table.

SPFS manages data blocks at the granularity of 4 KB but metadata blocks at 256 bytes (also referred to as XPLine, the unit of physical access to DCPMM) by default. This is to reduce the waste of NVMM space as well as to avoid write amplification on hardware, which can be caused by read-modify-write operations. However, if we use small blocks, which is 1/16 of the traditional block size, SPFS needs to keep track of a 16× larger number of blocks, which leads to high metadata management overhead. To reduce this overhead, SPFS groups 16 contiguous free blocks into a *cluster* of 4 KB and manages the locations of free clusters in the *cluster bitmap* as in conventional file systems. For partially used clusters, we manage the locations of free blocks using *block bitmap hash table* and classical volatile segregated lists.



(a) Block Hashing: $O(B)$ Write, $O(1)$ Read

(b) Extent Hashing: $O(logB)$ Write, $O(logB)$ Read

Figure 5: Block vs. Extent Hashing

Figure 4 shows the layout of physical NVMM space for SPFS. The first 4 KBytes is the *superblock* that contains various metadata including the file system magic number, block/cluster/ inode size, the number of clusters, the number of inodes, metadata for the three hash tables, etc. Then comes the cluster bitmap, where each bit in the cluster bitmap indicates whether all blocks in the corresponding cluster are free or not. If any block in a cluster is in use, its corresponding bit in the cluster bitmap is set to one. Since the cluster bitmap uses one bit per cluster of 4 KB, the space overhead for the cluster bitmap is no larger than that of traditional file systems that manage free space at the granularity of 4 KB blocks.

The cluster bitmap does not indicate which blocks in a cluster are free or in use. Hence, each partially used cluster requires another metadata, the *block bitmap*, which is indexed in the *block bitmap* table. When SPFS allocates some, but not all, blocks in a cluster, it creates and inserts a block bitmap into the *block bitmap table*. The block bitmap table is used only for the clusters that are partially allocated. If a cluster has no free block, which is a common case for files larger than 4 KB, or if all blocks are free, which is also a common case when the file system is initially formatted, no block bitmap is needed in SPFS. To manage free blocks of partially used clusters and serve memory allocation requests quickly, SPFS manages volatile segregated lists constructed from the persistent block bitmap table. For a block allocation request, we select a segregated list based on the allocation request size.

4.2 Extent Hashing

SPFS indexes extents in a hash table called *extent table*. To the best of our knowledge, SPFS is the first hash-based file system that indexes extents using a hash table. HashFS [28], the state-of-the-art hash-based NVMM file system that also manages the file mapping information in a global hash table, requires every block number to be canonically stored in its corresponding bucket. That is, HashFS indexes blocks, not extents, as illustrated in Figure 5(a). Therefore, HashFS not only significantly increases the aggregate size of file mapping structures, but it also slows down writes because writing an extent of B blocks requires as many as B `store` instructions and cacheline flushes.

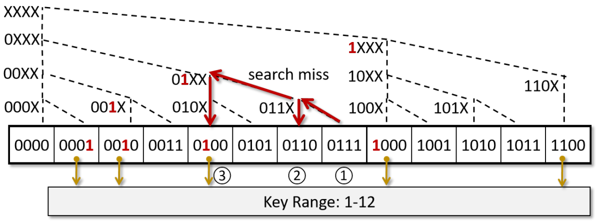


Figure 6: Searching Extent Hash Table

Algorithm 1 *Insert(inode, cluster_num, len, extent)*

```
if (len $\leq$ 0) return
current_key = hash(inode,cluster_num)
bucket = find_bucket(current_key)
    /* e.g., bucket_array[current_key%NumBuckets] */
bucket.store(inode, extent)
if len = 1 then
    return
else if cluster_num is odd then
    $stride\_size \leftarrow 1$
else
    if cluster_num != 0 then
        $TNZ \leftarrow$ ffs(cluster_num)$-1$
        $stride\_size \leftarrow$ previous_pow_of_two(min($len$, $1 \ll$ TNZ))
    else
        $stride\_size \leftarrow$ previous_pow_of_two($len$)
    end if
end if
Insert(inode, cluster_num + stride_size, len - stride_size, extent)
```

In contrast to block hashing, our novel *Extent Hashing* selects only a few buckets based on the binary representation of cluster numbers, as shown in Figures 5(b) and 6. Extent Hashing bounds the number of pointers for a given extent by log_2B, where B is the number of blocks in an extent.

The insertion and search algorithms of *Extent Hashing* are presented in Algorithms 1 and 2. The algorithms are short but they work with sophisticated bitwise operations such as `ffs` and `fls` (find the first/last bit set in a key) operations. Extent Hashing can be used with any hashing scheme including static and dynamic hashing schemes with various ad hoc optimizations such as linear probing, chaining, and cuckoo hashing. However, for ease of explanation, we will assume we are using static hashing and explain the insert and search algorithms using a walk-through example shown in Figure 6. In the example, we assume hash keys are of 4 bits, and the hash function `hash(inode, cluster_num)` returns `cluster_num` for ease of presentation. Note that Algorithms 1 and 2 can be implemented with any hash function and any hash table implementation.

Insert: Suppose we insert a range of keys [1,12] ($[0001_2, 1100_2]$) as shown in Figure 6. Initially, we start with the first key and store a pointer to the extent in its corresponding bucket. In the example, we store a pointer in bucket 0001_2. Then, we check how many consecutive zero bits are in the postfix of the hash key of the current bucket, which we refer to as *TNZ* (trailing number of zeros). I.e., TNZ is the number of consecutive zeros in the binary representation with no non-zero digits to the right of it. Since 0001_2 has no zeroes

Algorithm 2 *Search(inode, cluster_num, hash)*

```
$pos \leftarrow cluster\_num$
$mask \leftarrow (1 \ll \text{fls(cluster\_num)}) - 1$
while $true$ do
    key = hash(inode,pos)
    bucket $\leftarrow$ find_bucket(key)
    if bucket.contains(inode, cluster_num) then
        return bucket.getExtent(inode, cluster_num)
    end if
    if $pos = 0$ then
        break
    end if
    pos $\leftarrow$ pos & ((mask$\ll$ffs(pos)) & mask)
end while
```

on the right side of the rightmost bit 1, its TNZ is 0. The TNZ determines how many buckets to skip, i.e., the distance between the current bucket and the next bucket where we store the same pointer to the extent. We refer to this distance as *stride length*, which is set to 2^{TNZ}. In the example, since TNZ is 0, the stride length is $2^0 = 1$. Hence, we move to the next bucket (0010_2) and store another pointer to the extent.

The hash key of the current bucket (0010_2) has one zero after the rightmost bit 1. Hence, the TNZ is 1 and the stride length is 2, i.e., (2^1). Therefore, we skip the next bucket and move to the next next bucket (0100_2). Then, we store another pointer there and check the next stride. Since the current TNZ is 2, the stride length is 4. So, we move to bucket 1000_2 (0100_2+4). In bucket 1000_2, we have three consecutive zero bits in the postfix. So, the stride length is 8 (2^3). However, the next bucket offset $(1000_2 + 8)$ cannot exceed the range of the given extent. Hence, we decrease the TNZ value one by one $(2^3 \rightarrow 2^2)$ until the next bucket position is within the given key range. Finally, we store another pointer in bucket 1100_2 $(1000_2 + 2^2)$, and the insertion is complete.

Although the extent size is 12, only 5 pointers are stored in the hash table. If the size of a given extent is B, Extent Hashing stores a maximum of $2 \times log_2 B$ pointers in the worst case. In the best case, we store just one pointer in the hash table. As such, we can significantly reduce the number of pointers (from B to $2 \times log_2 B$) compared to block hashing, in particular, when the extent size is large.

Search: Although the extent hash table does not have a pointer for each hash key, we can find the extent using any hash key within the key range. The search algorithm shown in Algorithm 2 works as follows. If a query searches for an extent using a hash key k, whose binary number is $(b_1 b_2 b_3 b_4)_2$, we first look up bucket$[(b_1 b_2 b_3 b_4)_2]$. If the bucket does not have a pointer to the extent (i.e., a search miss occurs), it could be because the current bucket is not the starting point of a stride, not because the hash table does not contain that data. Therefore, we need to compute the starting index of a possible stride by flipping the trailing non-zero bits starting from the rightmost one and moving left. Thus, assuming b_4 is a non-zero, the next bucket we look up is bucket$[(b_1 b_2 b_3 0)_2]$. If this bucket, again, does not have a pointer to the extent, we continue in the said manner and look up, in sequence, bucket$[(b_1 b_2 00)_2]$, bucket$[(b_1 000)_2]$, and bucket$[(0000)_2]$.

For example, suppose a query searches for hash key 7 (0111_2) in the example shown in Figure 6. Then, we look up bucket$[0111_2]$ (step ①), which will fail as it does not have a pointer to the extent. Then, we search bucket$[(0110)_2]$ (step ②) and, finally, bucket$[(0100)_2]$ (step ③), which has a pointer to the extent.

The best-case complexity of this search algorithm is $O(1)$, but its worst-case complexity is not constant, but $O(log_2 B)$ where B is the number of buckets. That is, Extent Hashing trades-off search performance for insertion performance.

Probabilistic Fast Lookup: To strike a balance between insertion and search performance, we develop a *fast lookup* optimization. This optimization keeps track of which stride length is the most common and has each query first access the bucket with the most common stride length. For example, the most common stride length in Figure 5(b) is 4 due to the pointers in bucket 0, 4, 8, and 12. Note that there is one pointer with stride 1 in bucket 3, and there is also one pointer with stride 2 in bucket 10. If a query searches for cluster 7 where the most common stride length is 4, the fast lookup optimization searches bucket 4 (4 = 7 - (7%4)) before it accesses bucket 7 and 6 following the search path in order. The rationale behind this optimization is as follows. The search algorithm requires each query to access the nearest buckets in a log scale because the extent size is not known to queries. However, it may result in unnecessary accesses to a large number of buckets if the extent size is large. Therefore, if the bucket corresponding to the most common stride length is first checked, there is a chance of reducing the number of bucket visits. Since the stride length increases in power of 2, i.e., the number of different stride lengths is limited to log scale, the overhead of keeping track of the common stride length is not significant.

4.3 Path-name Resolution

SPFS manages directory entries in another hash table called the *name2inode table*. The name2inode hash table stores file/directory name and directory entry block number pairs using the hash key generated from the VFS dentry, its parent inode number, and the file name. Since SPFS indexes each file/directory entry rather than the full file path, renaming a directory does not affect other files in its sub-directories.

As a stackable file system, the name2inode hash table has directory entries only if the directory has a regular file in NVMM. Stacking files/directories in SPFS follows the standard conventions of stackable file systems [9, 14, 29, 38], i.e., i) if a given regular file name appears in both the upper and lower file systems, then the lower file is hidden; ii) if a given name is a directory, directory entries are combined; iii) if a `readdir` request does not find a directory entry from the name2inode hash table, SPFS forwards the `readdir` request to the lower file system. The directory entry is stored in the name2inode table only when blocks are migrated from a lower file system to SPFS, if it does not have one already.

One of the drawbacks of using a hash table is that directory entries in the same directory are normally stored in different buckets that causes problems to `readdir`. To resolve this problem, SPFS provides two options. One is to add two persistent pointers to each inode in the name2inode hash table to construct a doubly linked list for the inodes in the same directory. SPFS performs micro-logging (i.e., I/O operation-level logging) when file metadata is updated because multiple indexing structures need to be updated in a failure-atomic manner. The other option is to construct a volatile *readdir index* in DRAM when SPFS is mounted. The readdir index is different from the dentry cache in that it manages the entire structure of all directories in the file system regardless of whether a directory is loaded or not. Therefore, the volatile readdir index must be constructed when SPFS is mounted, and it must be persisted as a persistent index when SPFS is unmounted. Upon a system crash, we may lose updates in the volatile readdir index unlike the persistent readdir chain. To recover from system failures, the readdir index can be reconstructed from scratch by scanning the name2inode hash table. Although the second option increases the memory usage slightly, the low DRAM latency helps improve performance by up to 8% if the workloads are metadata-intensive (that make extensive use of calls such as `create`, `unlink`, and `rename`). For the performance study presented in Section 5, we use the latter option.

4.4 Recovery

SPFS performs micro-logging when file metadata is updated so that `fsck` can rollback uncommitted I/O operations. If a system crashes while creating a file, `fsck` will look up the name2inode hash table using the file name in the operation log and delete its corresponding entry. It will also delete the directory entry using the block number stored in the I/O operation log. In addition, `fsck` will walk the directory tree structure and perform a sanity check as in classic file system recovery methods.

5 Evaluation

We implement SPFS [1] in Linux kernel 5.1 We validated the reliability, robustness, and stability of SPFS using the POSIX file system test suite [3] and the Linux Test Suite [2]. SPFS passed both test suites successfully. In the following, we focus only on the performance aspect of SPFS.

5.1 Experimental Setup

We run experiments on two testbed servers, one with DCPMM and the other with NVDIMM-N. DCPMM server has dual Intel Xeon Gold 5215 processors (10 cores, 2.50 GHz), 128 GB of DDR4 DRAM, 256 GB of Optane DCPMM (2 × 128 GB), and a 2 TB Samsung 860 EVO mSATA SSD. The NVDIMM-N server has dual Intel Xeon Gold 5218 processors (16 cores, 2.30 GHz), 192 GB of DDR4 DRAM, 16 GB Dell EMC NVDIMM-N, and 512 GB Samsung 970 PRO NVMe SSD. On the NVDIMM-N server, we evaluate SPFS in a virtual environment (16 cores and 32 GB DRAM) using QEMU. Despite the future of DCPMM is uncertain, CXL Type 3 memory devices that provide durability will work with the existing PMDK (or OpenMPDK) ecosystem, and their latency will be higher than that of DRAM (170∼250 nsec) [1]. Therefore, we present the performance on the DCPMM server to evaluate how SPFS performs with NVMMs slower than DRAM.

Table 1: Filebench Workload Characteristics

Workload	File Size	R/W Size	# threads	R:W	# files
Fileserver	128 KB	1024 KB	50	1:2	100K
Webproxy	16 KB	1024/16 KB	100	5:1	100K
Webserver	16 KB	1024/16 KB	100	10:1	100K
Varmail	16 KB	1024/16 KB	16	1:1	100K
OLTP	10 KB	2/2256 KB	200	20:1	10

Table 2: FIU Workload Characteristics

Workload	Dataset Size	Read Size	Write Size	fsync (%)
Moodle	54 GB	55 GB	31 GB	38.922
Usr1	161 GB	171 GB	8 GB	86.025
Usr2	1.5 GB	5 GB	1 GB	75.114

We first quantify the performance effect of Extent Hashing, evaluate the performance of SPFS in standalone mode, and compare SPFS against EXT4-DAX and NOVA on the DCPMM server. Then, we quantify the performance effect of each stackable design of SPFS. Finally, we deploy SPFS on top of three popular Linux file systems, namely, EXT4, F2FS, and XFS, and compare the performance of SPFS+*x* against *x* and Ziggurat in both DCPMM and NVDIMM-N servers. File systems are mounted with the default mount options on top of the storage targeted by each design: (1) EXT4, F2FS, and XFS: SSD, (2) NOVA (Copy-on-Write (CoW) mode), EXT4-DAX, and SPFS in standalone mode: DCPMM (3) SPFS+*x* in stacked mode and Ziggurat: DCPMM+SSD or NVDIMM-N+SSD.

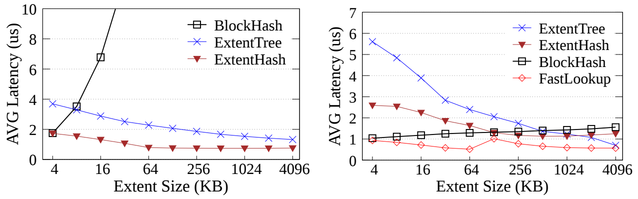
We run experiments using the Flexible I/O tester (FIO) micro-benchmarks [6] and the Filebench macro-benchmarks [33] as well as SNIA's FIU Filesystem SysCall Traces [10]. Tables 1 and 2 show the characteristics of the Filebench and FIU Filesystem SysCall Traces workloads, respectively. We also experiment with RocksDB [4] using the YCSB benchmark [13].

5.2 Analysis of Extent Hashing

In the first set of experiments, we compare the performance of file mapping structures - i) *per-file* `ExtentTree`, which is implemented using the FAST and FAIR B+tree [17], ii) *global* *`BlockHash`* (proposed and used in HashFS [28]), and iii) *global* *`ExtentHash`*. Both global block hashing and Extent Hashing are implemented on CCEH [27]. We evaluate the performance of indexing using microbenchmarks.

In the experiments shown in Figure 7, we measure the performance of indexing the extents that make up 8000 256-MB files with varying extent sizes, i.e., the larger the extent

[1] The code is available at https://github.com/DICL/spfs.



(a) Insert

(b) Search (Random)

Figure 7: Performance of File Mapping Structures

size, the fewer extents are indexed. Figure 7(a) shows the average latency of inserting one extent to each index. As the extent size increases, the insert latency of `ExtentTree` and `ExtentHash` decreases because the index size decreases. Specifically, when the extent size is 4 KB, the tree height is 4, but when the extent size is greater than 256 KB, the tree height is reduced to 2.

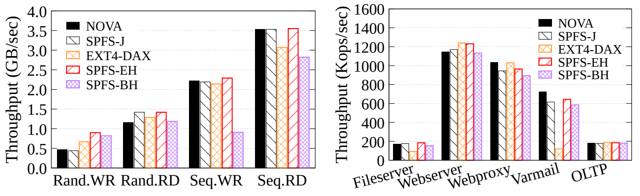
Block hashing shows the worst insertion performance as the extent size increases because the number of pointers to index increases. Specifically, when the extent size is 4 MB, it has to update and call `clwb` for as many times as 1024. As such, its insertion latency is up to 906× higher than that of `ExtentTree`. Extent Hashing shows the fastest insertion latency because hash-based indexes updates fewer number of cachelines than FAST and FAIR B+tree.

Figure 7(b) shows that when the extent size is smaller than 128 KB, `BlockHash` outperforms `ExtentTree` and `ExtentHash` due to its constant lookup cost. Note that, `ExtentHash` accesses multiple buckets following the search path described in Section 4.2. However, as the extent size increases, `ExtentTree` benefits from the reduced index size, making the performance of all indexes similar.

`FastLookup` denotes the performance of Extent Hashing with the fast lookup optimization that we described in Section 4.2. Fast lookup is an optimization affected by probability, but it finds an extent in $O(1)$ with very high probability in the experiments. Therefore, `FastLookup` outperforms `BlockHash`, which suffers from a much larger number of pointers in the hash table. We also observe in the experiments with FIU Filesystem SysCall traces that the probability of finding an extent in $O(1)$ in Usr1 and Usr2 workloads is as high as 60% and 98%, respectively.

5.3 Standalone Mode with DCPMM

We now compare the performance of SPFS in standalone mode against NOVA and EXT4-DAX. We run the experiments in DCPMM server because SPFS is not intended for use in standalone mode for small NVDIMM-N. To evaluate the performance effect of Extent Hashing, we faithfully implemented the block hashing scheme as proposed in HashFS. We denote the performance of SPFS with Extent Hashing and block hashing as SPFS-EH and SPFS-BH, respectively. Both SPFS-EH and SPFS-BH run in *metadata* mode, i.e., they do not guarantee strong data consistency, but only metadata consistency is guaranteed as in EXT4-DAX. SPFS-J denotes



(a) FIO Results

(b) Filebench Results

Figure 8: Performance in Standalone Mode (DCPMM)

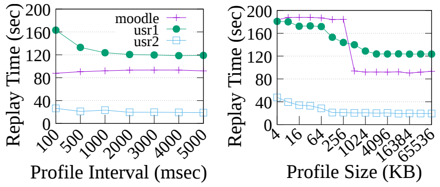
the performance of SPFS-EH in *journal* mode, which logs all data and metadata changes by copying the data into an undo log region if the write size is smaller than 256 KB. If the write size is larger than 256 KB, it performs CoW as in NOVA. We set this threshold size to 256 KB because it conservatively balances the logging overhead and fragmentation issue. If the threshold size is smaller, it leads to fragmentation, i.e., extents are frequently split because CoW allocates a new extent. If it is larger, the logging overhead becomes non-negligible.

5.3.1 FIO Results

The FIO benchmark is used to evaluate sequential and random *read* and *write* performance. Each workload accesses a 10 GB file, and read/write sizes are set to 256KB and 4KB for sequential and random workloads, respectively. Figure 8(a) shows the results. SPFS-EH shows up to 60% higher sequential write throughput than SPFS-BH because block hashing requires much larger metadata accesses. The sequential read throughput of SPFS-BH is also 20% lower than SPFS-EH because larger file mapping metadata adversely affects read performance as well as write performance. For the same reason, the random read and write throughput of block hashing is also 16% and 9% lower than that of Extent Hashing, respectively. In particular, FIO allocates very large extents in advance regardless of the type of workload, i.e., even for random I/Os. Despite large extents, `SPFS-BH` indexes a large number of individual blocks and the lookup performance deteriorates.

NOVA shows similar sequential read and write performance with SPFS-EH. However, the random read and write throughput of NOVA is 19% and 39% lower than that of SFPS-EH because NOVA provides strong data consistency whereas SPFS-EH supports only metadata consistency. With data journaling enabled, SPFS-J shows similar write performance with NOVA as both of them perform CoW. On the other hand, SPFS-J shows 1.2× higher throughput than NOVA for random reads because SPFS manages data blocks in units of extents in DRAM while NOVA indexes write logs in units of pages in DRAM.

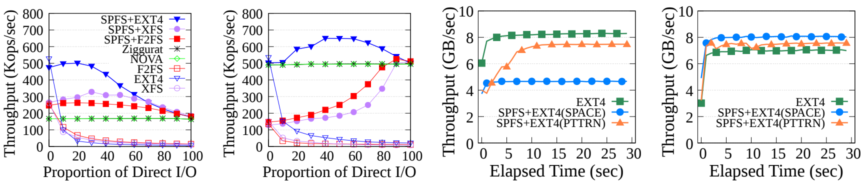
As a stackable file system, SPFS does not have to enforce strong data consistency if the lower file system does not require strong data consistency. Eliminating the logging overhead, SPFS-EH shows up to 40% performance improvement for the random write workload compared to NOVA. Since EXT4-DAX also does not log data blocks, it shows higher random write throughput than NOVA and SPFS-J. However,



(a) Interval

(b) Size

Figure 9: Profile Parameters (NVDIMM)



(a) Write (DCPMM)

(b) Write (NVDIMM)

(c) Read (DCPMM)

(d) Read (NVDIMM)

Figure 10: Performance Effect of Delegating I/O Requests to Lower File System

EXT4-DAX is consistently outperformed by SPFS-EH for all FIO workloads. Furthermore, for sequential reads and writes, EXT4-DAX is even outperformed by NOVA and SPFS-J despite the fact that they guarantee stronger data consistency. This is because NOVA and SPFS take advantage of CoW for sequential I/O.

5.3.2 Filebench Results

Figure 8(b) shows the experimental results with the Filebench workloads. For the *Fileserver* workload that creates, deletes, reads, appends, and copy files in large I/O units, i.e., 1 MB for reads and writes (copy) and 16 KB for appends, SPFS-J shows 5.7% higher throughput than NOVA (167.67 vs. 176.37) because SPFS-J benefits from the efficient extent-based metadata management, whereas NOVA replaces write logs in units of pages in DRAM. EXT4-DAX shows the worst performance because it suffers from the overhead of unconditional block initialization. Unlike EXT4-DAX, NOVA and SPFS initialize the unwritten portion of the cluster only when needed. SPFS-J and SPFS-EH show similar performance with the *Fileserver* workload because it does not overwrite existing blocks and the logging overhead is negligible. SPFS-BH has 16% lower throughput than SPFS-EH because the read and write granularity of Fileserver workload is 1 MB and Extent Hashing manages large extents more efficiently.

Webserver is a read intensive workload where each thread opens, reads, and closes a file, and every 10th read operation appends a small data to a log file. In this workload, SPFS-J and NOVA show comparable performance.

The *Webproxy* and *Varmail* workloads create and delete many small files in a single directory. In these two workloads, SPFS is outperformed by NOVA because SPFS frequently allocates and deallocates blocks for directory entries, and thus performs metadata journaling for file system consistency, whereas NOVA appends directory entries in a log-structured fashion and hides deallocation overhead via background garbage collection. As a result, NOVA shows up to 9% and 17% higher throughput than SPFS-EH for Webproxy and Varmail, respectively. Efficient directory management is of paramount importance in the native file system, but the primary goal of SPFS is to serves NVMM as a persistent writeback cache to NVMM-oblivious filesystems. Therefore, directory management performance is not optimized. We leave the directory management optimization for future work. EXT4-DAX shows very poor performance for *Varmail* because of two reasons. One is the unconditional block initialization problem mentioned earlier. The other reason is because of additional memory copy overhead from metadata journaling. This overhead is negligible in other workloads because journaling is done in the background. However, *Varmail* calls `fsync()` frequently, which incurs metadata journaling overhead thereby affecting the workload throughput.

The *OLTP* workload emulates database transactions at the file system level. In this transactional workload, synchronous writes affect file system throughput the most. Since all file systems store synchronous writes in NVMM, they do not show any meaningful difference. SPFS-BH shows only 2% lower throughput than SPFS-EH because of small (i.e., 2 KB) random reads/writes that rarely benefit from extents.

5.4 Quantification of Stackable Design

5.4.1 Parameters for Sync Point Profiler

In this section, we quantify how the profile interval and the write size threshold for the Sync Point Profiler affects performance using three FIU Filesystem SysCall Traces workloads. We present the results on the NVDIMM-N server, but the results on the DCPMM server are almost the same.

Figure 9(a) shows that the performance of SPFS is insensitive to the profile interval unless it is set small (< 500ms). Obviously, the interval between transactional writes can vary across applications. Therefore, for the rest of the experiments, we choose 1 second as the default. Figure 9(b) shows the results as the write size parameter is varied (x-axis). It shows that the replay time of the `Moodle` workload improves significantly when the write size parameter is set to be equal or larger than 1 MB because this workload has relatively large 1 MB synchronous writes. For `Usr1` and `Usr2`, we see that performance gradually improves as the write size increases, but then remains relatively constant beyond 1 MB. Based on these observations, we conservatively set the default write size parameter to 4 MB - a sufficiently large value that was also used as the default value in Ziggurat [39]. All results that follow use this value.

5.4.2 Delegating I/O Requests to Lower File System

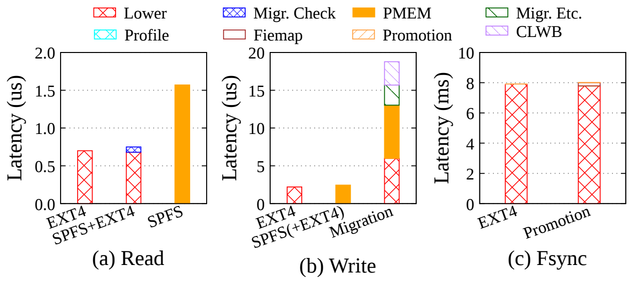
We now perform synthetic microbenchmark experiments to validate our proposition that migrating files to NVMM does not always guarantee better performance. That is, we analyze which types of I/Os benefit from promotion and when they benefit from demotion.

Performance Effect of Delegation: As a stackable file system, SPFS shines when synchronous and asynchronous I/O workloads are mixed. To test various mixed workloads, we use *diomix*, which is a synthetic workload generated from a mix of two sequences of file operations, one for buffered I/O (BIO) and the other for direct I/O (DIO), of the *Fileserver* workload of Filebench, and whose ratio between BIO and DIO can be controlled.

Figure 10(a) shows the results for *diomix* in DCPMM server, as the DIO rate changes. We disable background demotion to only quantify the effect of promotion. We observe that the I/O throughput of Ziggurat is insensitive to the DIO rate and that it fails to leverage the faster page cache in DCPMM server, which leads to the same performance as NOVA. As a result, they are consistently outperformed by SPFS+*x*, which delegates BIO to EXT4, F2FS and XFS, and benefits from the low latency of the page cache in DRAM. Note that EXT4, F2FS, and XFS also benefit from the page cache, and when there is no DIO, each file system shows 10%, 3%, and 1% higher throughput, respectively, than its SPFS+*x* counterpart. This is because of the overhead that comes from the stackable design. Specifically, SPFS+*x* looks up its name2inode table just to find out it does not have the requested file. This exemplifies the importance of indexing performance in SPFS. We observe sharper and then continued performance decline as the rate of DIO increases. Unlike EXT4, F2FS and XFS, the throughput of SPFS+*x* show much smoother curves as SPFS+*x* detects the I/O types and steers the BIOs to the lower file system while absorbing the DIOs in DCPMM, benefiting from the device aware stackable design of SPFS+*x*.

Figure 10(b) shows the results for the same *diomix* workload in NVDIMM server. Because the page cache of the lower file system has the same access latency with NVDIMM, SPFS+*x* does not benefit from delegating write requests to the lower file system but suffers from its stackable design overhead. Again, the performance of Ziggurat is similar to NOVA as it aggressively steers most writes to NVMM. In contrast, SPFS+*x* is designed to use NVMM conservatively and gradually demote files in the background. We could not evaluate Ziggurat for the case when NVMM is full as it crashes.

Performance Effect of Demotion: While promotion improves write performance of the lower file system, it may degrade read performance when NVMM is slower than DRAM. In the experiments shown in Figures 10(c) and 10(d), we pre-populate file systems with 64 16 MB transactional log files, i.e., SPFS stores them in NVMM, and run a synthetic microbenchamrk that reads random blocks from those files. In DCPMM server, the read throughput of EXT4 file system is higher than SPFS+EXT4 because it eagerly copies all the requested blocks to the page cache, whereas SPFS+*x* demotes files, i.e., copies them from DCPMM to DRAM/disk in a lazy manner. Specifically, `SPFS+EXT4(PTTRN)` demotes a file to the lower file system and copies to the page cache if the file access pattern changes to be read-intensive. As more read requests are processed, the Sync Factors of promoted files decrease, and when they become lower than a hard limit set by administrators, they are demoted to the lower file system such that they can benefit from the page cache in DRAM. Thus, read performance of `SPFS+EXT4(PTTRN)` improves over time to a level similar to page cache performance. This result confirms the well-known fact that performance is improved by placing frequently accessed data in the fastest memory, which Ziggurat has neglected.



(a) Read (b) Write (c) Fsync

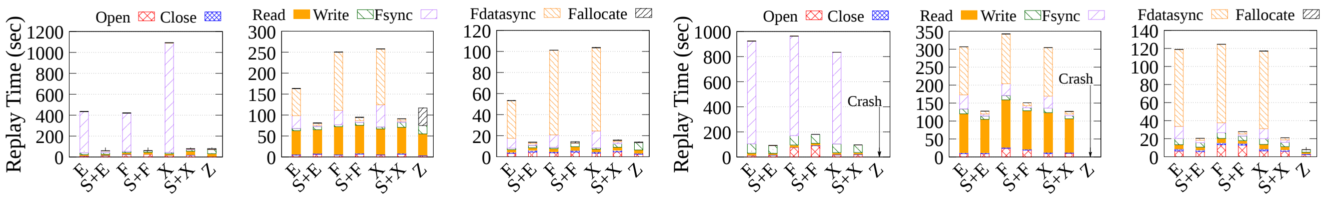
Figure 11: Latency breakdown of each mode in DCPMM

If such a hard limit on the Sync Factor is not set by administrators, a file is not demoted to the lower file system unless the NVMM space is running low (denoted as `SPFS+EXT4(SPACE)`. Therefore, `SPFS+EXT4(SPACE)` does not demote files and read requests to those files suffer from higher access latency of DCPMM. In contrast, demoting files from NVDIMM to the page cache on the NVDIMM server does not improve read performance, but counteracts it. As a result, `SPFS+EXT4(SPACE)` shows the highest throughput in NVDIMM server. In the default settings, background demotion is triggered when more than 80% of NVMM space is used. When files are demoted to the lower file system in the background, the foreground write throughput of SPFS+*x* is reduced by up to 40% due to the limited bandwidth of NVMM and also due to conflicting SPFS metadata updates. To minimize performance interference, SPFS suspends the background demotion while foreground processes perform I/O, unless there are no free blocks in NVMM.

5.4.3 Stacking Overhead

As a stackable file system, SPFS places additional latency on the lower file system in exchange for improving the performance of small synchronous writes. In the experiments shown in Figure 11, we breakdown the latency of read, write, and fsync using a synthetic workload that performs random reads and writes to 1 MB file that consists of a single extent.

`SPFS+EXT4` denotes the read latency when the extent is not found in SPFS and is read from the lower file system - EXT4. The stacking overhead (i.e., the overhead to check whether the corresponding extent has been migrated to NVMM or not) accounts for 9.89%, and thus `SPFS+EXT4` shows similar latency with `EXT4`. `SPFS` denotes the read latency when the



(a) Moodle (DCPMM) (b) Usr1 (DCPMM) (c) Usr2 (DCPMM) (d) Moodle (NVDIMM) (e) Usr1 (NVDIMM) (f) Usr2 (NVDIMM)

Figure 12: FIU Trace Replay Time S:SPFS, E:EXT4, X:XFS, F:F2FX, Z:Ziggurat

extent is in SPFS, i.e., NVMM. Due to the high latency of Optane DCPMM, the read latency of `SPFS` is about 2.26× higher than that of EXT4.

In Figure 11(b), the write latency of EXT4 (`EXT4`) is similar to the write latency when the write is steered to NVMM (`SPFS(+EXT4)`). That is, unless fsync is called, there is not much difference in latency whether the write is steered to DCPMM or the VFS cache. `Migration` denotes the latency of the first write to an extent that the profiler decided to migrate from the lower file system to SPFS. `Migration` has a high latency, but it is a one time tax. Once migrated, subsequent fsync calls will be replaced with nop..

Figure 11(c) shows the fsync latencies when SPFS bypasses fsync to the lower file system and when it decides to promote a file in the lower file system to SPFS. The promotion overheads such as `Fiemap` and `Profile` account for 2.68% and 0.01% of total latency.

5.5 Stacked Mode Performance Comparison

Finally, we run real world trace FIU and YCSB workloads and compare the performance of SPFS in stacked mode (SPFS+*x*) against file systems for large block devices, i.e., EXT4, XFS, F2FS, and Ziggurat.

5.5.1 FIU Traces

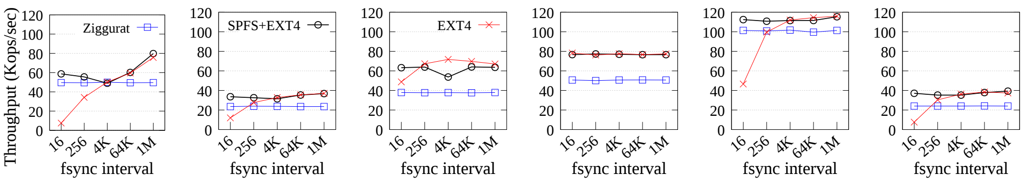
Figure 12 shows the performance results using the FIU Filesystem SysCall Traces [10]. For these experiments, we measure the replay time on each file system as we submit the file system operations from the traces in batches. Thus, for all results for the FIU workloads, lower is better.

Although the FIU workload consists of traces from six applications - *Backup*, *Gsf-filesrv*, *Ug-filesrv*, *Moddle*, *Usr1*, and *Usr2*, the performance results of *Backup* (1.2 TB, 0.001% `fsync()`), *Gsf-filesrv* (190 GB, 0.326% `fsync()`), and *Ug-filesrv* (812 GB, 0.001% `fsync()`) are not presented because Ziggurat crashes for those large FIU workloads not only in NVDIMM but also in DCPMM servers and also because they are not transactional workloads, i.e., `fsync()` calls account for less than 0.3%. Even if we evaluated the performance of Ziggurat by reducing the size of those workloads small enough to fit in DCPMM, we observed that Ziggurat is outperformed by EXT4, F2FS, and XFS, and SPFX+*x* because Ziggurat fails to leverage the fast VFS cache. The performance of SPFS+*x* (`SPFS+EXT4`, `SPFS+F2FS`, and `SPFS+XFS`) is similar or slightly worse than that of *x* (EXT4, F2FS, and XFS) for the workloads where `fsync()` calls are rarely made. Without `fsync()` calls being made and steering writes to NVMM, the added overhead of the stacked file system tends to make SPFS+*x* perform worse than *x*.

With *Moodle*, *Usr1* and *Usr2*, calls to `fsync()` are frequently made. Therefore, EXT4, F2FS, and XFS suffer from high synchronization overhead while SPFS+*x* and Ziggurat eliminate this overhead by steering synchronous writes to NVMM. Thus, for *Moodle*, SPFS+*x* reduces the trace reply time in DCPMM server to only 14%, 15%, and 7% of the *x* counterparts EXT4, F2FS, and XFS, respectively. Similarly, for Usr1, SPFS+*x* shows 2×, 2.6×, and 2.8×, and for Usr2, 3.8×, 7.1×, and 6.5× faster trace replay times, compared to the *x* counterparts EXT4, F2FS, and XFS, respectively.

Ziggurat also outperforms EXT4, F2FS, and XFS for the *Moodle*, *Usr1*, and *Usr2* workloads. Compared to Ziggurat, `read()`, `write()`, and `fallocate()` are consistently faster with SPFS+*x*. The write time of Ziggurat is higher than SPFS+*x* because it profiles and steers each individual write to NVMM while SPFS+*x* migrates them in a lazy manner, that is, only at `fsync()` calls with intervals less than one second and aggregate flush sizes less than 4 MB. For `fallocate()`, Ziggurat spends a significant amount of time initializing allocated blocks. However, SPFS creates files on the lower file systems first, such that it benefits from the highly efficient *uninit* and *unwritten* states (i.e., allocated and mapped but uninitialized blocks) of the disk file systems, which prevents applications from reading garbage blocks even if allocated blocks have not yet been initialized. However, due to the stacking overhead, `open()` is slower in SPFS+*x* than Ziggurat. Also, `fsync()` and `fdatasync()` are faster with Ziggurat because they are no-ops if previous writes were steered to NVMM. Overall, due to faster `write()` and `fallocate()`, SPFS+*x* is up to 1.44× and on average 1.16× faster than Ziggurat in DCPMM server.

On the NVDIMM server, we could not run the *Moodle* and *Usr1* workloads with Ziggurat because their sizes are larger than the NVDIMM size. For the *Usr2* workload where the average I/O size is 1.2 KB, Ziggurat steers most writes to NVDIMM, and thus shows the performance of the in-memory file system - NOVA and outperforms SPFS +*x*. Similar to the results on the DCPMM server, SPFS+*x* improves the performance of *x* by up to 9.9×, 2.4×, and 5.8× for the Moodle, Usr1, and Usr2 workloads, respectively.



(a) Load (b) Workload A (c) Workload B (d) Workload C (e) Workload D (f) Workload F

Figure 13: YCSB Throughput of RocksDB with Varying Frequency of `fsync()` (DCPMM)

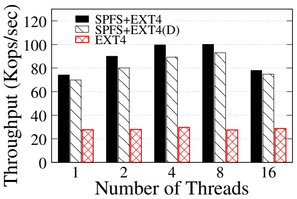


Figure 14: Load Throughput (NVDIMM)

5.5.2 RocksDB

Finally, we evaluate the performance of SPFS+EXT4, Ziggurat, and EXT4 using RocksDB v.6.2.2. For the experiments shown in Figure 13, we load the database with 4 million 1 KB records in the loading phase (`Load`) in the DCPMM server. In the transactions phase, 4 million queries in uniform distribution are submitted in batches for each workload. RocksDB offers various write options including whether to call `fsync()` to flush dirty pages. Our experiments measure YCSB throughput while varying the `fsync()` interval by enabling the `setSync` option for every 16 writes, 256 writes, 4096 writes, and so on, that is, in multiple of 16 increments. Figure 13 shows the throughput results, which can be summarized as follows.

EXT4: EXT4 performance improves as we increase the `fsync()` interval. If `fsync()` is called often, EXT4 suffers because of the slow block device. As `fsync()` is called less, EXT4 benefits from the low latency of the page cache and performance improves.

Ziggurat: Ziggurat performance is insensitive to the `fsync()` interval. This is because Ziggurat profiles individual writes, and as all writes are smaller than 4 MB, they are all steered to NVMM resulting in the same performance as NOVA.

SPFS+EXT4: In contrast, SPFS+EXT4 considers the total number of written bytes to be flushed by `fsync()`. Therefore, if `fsync()` is called per every 4096 or fewer writes, SPFS stores the 4 MB or smaller synchronous writes in NVMM and significantly reduces the `fsync()` overhead. Thus, for the `Load` workload, `SPFS+EXT4` shows 7.7× and 1.6× higher throughput than EXT4 when the `fsync()` interval is 16 and 256 writes, respectively. We observe all small WAL log files are migrated to NVMM as expected, whereas all SSTable files, which contain key-value records, are stored in the EXT4 file system because its size (64 MB) is much larger than the profiling threshold of SPFS (4 MB). Nonetheless, SPFS+EXT4 outperforms Ziggurat, which stores both WAL and SSTables in NVMM. This is because SPFS leverages DRAM, NVMM, and SSD characteristics altogether, while Ziggurat relies only on NVMM. If `fsync()` is called less frequently, e.g., `fsync()` is called every 64K or 1M writes, SPFS+EXT4 stores all writes in EXT4 and benefits from the VFS cache and periodic writebacks. Therefore, `SPFS+EXT4` shows similar performance with EXT4.

In the experiments shown in Figure 14, we measure YCSB Load throughput on the NVDIMM server, varying the number of client threads while the `fsync()` interval is fixed to 256. Due to the frequent `fsync()`, EXT4 does not scale with the number of client threads. However, the throughput of SPFS+EXT4 increases up to 8 threads because it absorbs the synchronous writes in NVDIMM. When the number of client threads is 16, the throughput degrades because the number of total threads (i.e., client and background compaction threads) exceeds the number of available cores and memory contention occurs. Note that SPFS+EXT4(D) denotes the performance of SPFS+EXT4 when NVDIMM is full and the background demotion migrates files from NVDIMM to the lower file system. Due to the performance interference, the demotion decreases the throughput by up to 7%.

6 Conclusion

Managing two different storage devices with completely different properties in a single file system has practical limitations. In this study, we designed and implemented *SPFS*, a stackable file system for NVMM that exploits the performance of NVMM for order-preserving small synchronous writes and yet takes advantage of the faster DRAM cache as well as the large capacity that legacy block device file systems provide. In addition, SPFS manages all file system metadata in dynamic hash tables that are built on Extent Hashing that exhibits fast insertion as well as fast scan performance.

We perform extensive evaluations and compare SPFS with state-of-the-art file systems. In standalone mode, SPFS shows comparable performance to NOVA, while in stacked mode, SPFS+x improves performance by up to 9.9× compared to the lower file system x executing alone.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable comments and feedback. We also thank our shepherd, Youyou Lu for guiding us during the revision process.
This research was supported in part by Samsung Electronics, and also by NRF (grant No. NRF2022R1A2C2091680), IITP (grant No. 2021-0-01817), and ETRI (grant No. 20ZS1310). Much of this work was done when Hobin Woo was enrolled in the Master program at Sungkyunkwan University. Daegyu Han contributed equally as he took over the work after H. Woo graduated. The corresponding author is Beomseok Nam.

References

[1] Compute Express Link CXL Latency How Much is Added at HC34. https://www.servethehome.com/compute-express-link-cxl-latency-how-much-is-added-at-hc34/.

[2] Linux Test Suite. https://linux-test-project.github.io/.

[3] POSIX File System Test Suite. https://github.com/pjd/pjdfstest.

[4] RocksDB. https://rocksdb.org/.

[5] Samsung SZ985 Z-NAND SSD. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf.

[6] Jens Axboe et al. FIO (Flexible I/O Tester). https://github.com/axboe/fio.

[7] Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. Persistent Memory: A Survey of Programming Support and Implementations. *ACM Computing Surveys (CSUR)*, 54(7):1–37, 2021.

[8] 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 2021 USENIX Annual Technical Conference (ATC)*, pages 689–703, 2021.

[9] Neil Brown. Overlay Filesystem. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[10] Daniel Campello, Hector Lopez, Ricardo Koller, Raju Rangaswami, and Luis Useche. Non-blocking Writes to Files. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 151–165, 2015.

[11] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1077–1091, 2020.

[12] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. HiNFS: A Persistent Memory File System with Both Buffering and Direct-Access. *ACM Transactions on Storage (TOS)*, 14(1):1–30, 2018.

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010.

[14] Puja Gupta, Harikesavan Krishnan, Charles P. Wright, Mohammad Nayyer Zubair, Jay Dave, and Erez Zadok. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical report, Stony Brook University, 2004.

[15] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–162, 2021.

[16] Tyler Hicks, Dustin Kirkland, and Michael Halcrow. eCryptFS. http://www.ecryptfs.org/.

[17] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *Proceedings of the 16th Usenix Conference on File and Storage Technologies (FAST)*, pages 187–200, 2018.

[18] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, 2015.

[19] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. BetrFS: Write-Optimization in a Kernel File System. *ACM Transactions on Storage (TOS)*, 11(4):1–29, 2015.

[20] 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 17th Usenix Conference on File and Storage Technologies (FAST)*, pages 191–205, 2019.

[21] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 993–1005, 2018.

[22] 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 *Proceedings of the 17th USENIX*

Conference on File and Storage Technologies (FAST), pages 65–78, 2019.

[23] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 161–177, 2022.

[24] 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)*, pages 460–477, 2017.

[25] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286, 2015.

[26] Avantika Mathur, M. Cao, and A. Dilger. Ext4: The Next Generation of the Ext3 File System. *;login: Usenix Magazine*, 32, 2007.

[27] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage (FAST)*, pages 31–44, 2019.

[28] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. Rethinking File Mapping for Persistent Memory. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 97–111, 2021.

[29] Junjiro R. Okajima. AUFS - Another Union Filesystem. http://aufs.sourceforge.net/.

[30] D. Rosenthal. Evolving the Vnode Interface. In *USENIX Summer*, 1990.

[31] Simon Sharwood. Last Week Intel Killed Optane. Today, Kioxia and Everspin Announced Comparable Tech: Rumors of Storage-Class Memory's Demise May Have Been Premature. https://www.theregister.com/2022/08/02/kioxia_everspin_persistent_memory/.

[32] Pradeep J Shetty, Richard P Spillane, Ravikant R Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building Workload-Independent Storage with VT-Trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–30, 2013.

[33] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *;login: USENIX Magazine*, 41(1):6–12, 2016.

[34] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 93–111, 2021.

[35] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO Stack for Flash Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 211–226, 2018.

[36] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC)*, pages 349–362, 2017.

[37] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, 2016.

[38] Erez Zadok, Ion Badulescu, and Alex Shender. Extending File Systems Using Stackable Templates. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC)*, pages 57–70, 1999.

[39] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, pages 207–219, 2019.

Citron: Distributed Range Lock Management with One-sided RDMA

Jian Gao Youyou Lu Minhui Xie Qing Wang Jiwu Shu[*]
Tsinghua University

Abstract

Range lock enables concurrent accesses to disjoint parts of a shared storage. However, existing range lock managers rely on centralized CPU resources to process lock requests, which results in server-side CPU bottleneck and suboptimal performance when placed in a distributed scenario.

We propose Citron, an RDMA-enabled distributed range lock manager that bypasses server-side CPUs by using only one-sided RDMA in range lock acquisition and release paths. Citron manages range locks with a static data structure called segment tree, which effectively accommodates dynamically located and sized ranges but only requires limited and nearly constant synchronization costs from the clients. Citron can also scale up itself in microseconds to adapt to a shared storage of a growing size at runtime. Evaluation shows that under various workloads, Citron delivers up to 3.05× throughput and 76.4% lower tail latency than CPU-based approaches.

1 Introduction

Large-scale distributed applications have high demands to access shared storage resources concurrently [60, 65]. File systems designed for high-performance computing (HPC), for example, are usually required to handle massive parallel I/O requests to different parts of a single data file [11, 32, 49]. Disaggregated memory pools have the need to allow multiple clients to access the same memory space simultaneously, possibly with different access patterns [19, 40, 50]. These systems require the capability to correctly and efficiently coordinate concurrent accesses to a large-scale shared storage.

Lock is a common and essential approach to enabling correct concurrent accesses to a shared storage. A wealth of research contributes to designing mutual exclusive locks (i.e., *mutexes*) and their variants [20, 22, 29, 33, 76], which grant exclusive access (or write) permission of the shared storage to at most one client at any time. Still, mutexes can be too coarse-grained and, thus, inefficient. For this reason, range locks become a preferable alternative since they allow finer-grained concurrency, i.e., clients simultaneously operating at disjoint parts of the same shared storage resource [35, 39].

Existing distributed range lock managers (DRLMs) grant and revoke locks with *centralized* server-side[1] CPUs through a remote procedure call (RPC) interface. However, with prevalent high-speed networks, CPU-oriented DRLMs cause performance bottlenecks. First, they limit throughput because of the mismatch between high network packet rate (e.g., 215 Mops/s for NVIDIA ConnectX-6 [58]) and limited CPU resources and that they need to perform CPU-consuming traversal and modification to complex dynamic data structures upon lock operations (e.g., the interval trees in Lustre [47]). Second, they also incur high queueing latencies (4-5 network roundtrip times, §2.2) due to the RPC paradigm. In latency-sensitive scenarios like memory pools, a CPU-based DRLM can become a major latency contributor in the critical path.

Remote Direct Memory Access (RDMA) offers a chance to avoid the CPU bottleneck with its one-sided verbs that can bypass server-side CPUs. However, taking this chance requires a comprehensive re-design of the range lock protocol. First, existing DRLMs are built atop dynamic data structures, but RDMA incapacitates them due to the lack of support for dynamic remote memory allocation. Second, these DRLMs are also RDMA-unconscious and perform many memory accesses to their data structures in lock operations, which turn into excessive network roundtrips when using one-sided RDMA, overshadowing RDMA's high performances.

This paper proposes Citron, an efficient distributed range lock manager. Citron acquires and releases range locks using only one-sided RDMA to lift the burden off server-side CPUs with an RDMA-conscious lock protocol based on static data structures to exploit the full performance potentials of the RDMA hardware. Specifically, Citron retrofits *segment tree*, an RDMA-friendly static data structure, to manage lock entries. Thus, Citron simplifies lock conflict resolution into the communication between ancestor and descendant nodes on the tree. To effectively handle dynamically positioned and sized lock requests, Citron develops a protocol tightly interwoven with the range lock specs, the segment tree's memory layout, and the one-sided RDMA semantics. Clients lock at different levels of the segment tree and pay nearly constant costs to synchronize with conflicting peers. In the best case, lock acquisition takes only two RDMA roundtrips.

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

[1] For disambiguation, in this paper, we use different terms for different purposes: *servers* are counterparts of clients; *machines* are computers in the distributed system; *nodes* are components of tree data structures.

For a shared storage whose size grows, Citron provides a mechanism to scale itself up (i.e., expand its capacity), leveraging the structural self-similarity of segment trees. Citron enables scaling up the lock tree to a proper size with one-sided RDMA and minimum server-side CPU intervention.

Citron offers several benefits. First, it is CPU-efficient. To our knowledge, Citron is the first DRLM that uses only one-sided RDMA for lock acquisitions and releases, which obviates the server-side CPU bottleneck. Second, Citron delivers high performance. Evaluation shows that Citron outperforms CPU-based range lock managers by up to 3.05× in throughput and 76.4% in latency under different workloads.

2 Background

2.1 RDMA

RDMA is a network protocol with low latency, high throughput, and low CPU overhead. Due to these benefits, numerous distributed file systems [2, 3, 26, 42, 44, 46, 47, 77], transaction systems [5, 16, 31, 41, 72], and lock managers [13, 54, 79] are built atop or compatible with RDMA.

Machines must equip RDMA-capable NICs (RNICs) to communicate with RDMA. Clients first post RDMA verbs to queue pairs (QPs) and later poll the completion queues (CQs) associated with the QPs for completion events. RDMA supports one-sided verbs, including `read`, `write`, atomic `compare-swap` (CAS), and atomic `fetch-add` (FAA). Furthermore, a wide range of off-the-shelf RNICs (e.g., from Mellanox Connect-IB to NVIDIA ConnectX-7 [51, 57–59]) also support *masked atomic verbs* [56], which perform similarly to standard atomic verbs but have more flexibility.

For masked-CAS, users need to provide a compare bitmask and a swap bitmask. The compare and swap steps are each performed with regard to the corresponding bitmask. The masked-out bits will not get compared or swapped.

For masked-FAA, users need to provide a bitmask that splits the 8 bytes into different fields. Each set bit in the bitmask indicates the left boundary of a field, and FAA is performed separately within every field. The field boundaries can occur at any position; non-byte-aligned fields are allowed.

2.2 Distributed Range Lock Management

CPU-based centralized solutions. Most existing DRLMs rely heavily on server-side CPUs [3, 9, 47]. However, this kind of solutions are notorious for their high CPU overheads and the ensuing CPU bottleneck, including limited throughput and high queueing latencies; see Figure 1(a).

First, executing complex range lock operations with limited CPU resources not only bottlenecks the throughput but also inevitably harms co-locating CPU-demanding services that have little chance of being offloaded to the RNIC (e.g., path traversal in a distributed file system). Second, in the RPC paradigm, server-side CPUs fetch and process RPC requests from the RNIC-side queues. Under high concurrency, the requests will queue in the RNIC, which results in high queueing latencies. Also, when processing a range lock request, a CPU core cannot process other ones in the same queue, even if they do not conflict with each other logically.

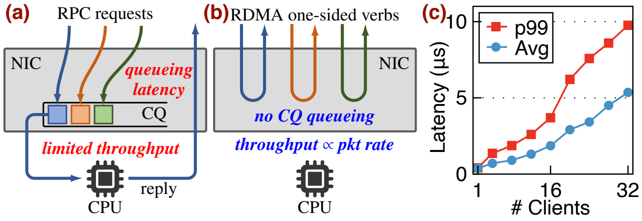


Figure 1: Flaws of RPC-based DRLMs and our motivation.

We demonstrate the high latencies by running eRPC [30] on a testbed consistent with §4.1. Clients synchronously send RPCs to one server, and the RPC handler runs for 100 ns. We measure the server-side queueing latency, i.e., the time between the RPC arrives at the NIC and the CPU processes the RPC, similarly to 2LClock [27]. Figure 1(c) shows the results. With 32 clients, the average queueing latency is 5.4 µs, more than 2× RDMA roundtrip times (RTTs). The p99 latency even reaches 9.8 µs (4-5 RTTs).

Mutex-based decentralized solutions. Dividing ranges into segments and associating each with a mutex is a strawman solution to decentralized range lock management [35], but it is only efficient when the access granularity is static and priorly known. In the case of unaligned or dynamically-sized ranges, this solution can suffer from a significant 92% throughput decline and 5.65× higher tail latencies; see §4.2.

3 Design

Our design goal is a high-performance DRLM that leverages one-sided RDMA to eliminate server-side CPU bottlenecks. As shown in Figure 1(b), a one-sided RDMA-based DRLM can remove the queueing latencies and offer higher throughput by offloading all lock operations to the RNIC's tailored ASIC, thus exploiting the full performance potentials of the RNIC.

3.1 Challenges and Design Principles

Challenge 1. We need a one-sided RDMA-conscious data structure that can efficiently manage dynamically positioned and sized range locks and resolve their conflicts.

➥ **Static tree structure for dynamic ranges.** Citron maps each requested range as precisely as possible to a constant number of nodes on a *segment tree*, a static data structure, to effectively manage dynamic range lock entries.

Challenge 2. To achieve low latency and high throughput, we must tailor the lock protocol to reduce the critical path lengths despite the complex range lock semantics.

➥ **Minimized synchronization overhead.** Citron's protocol couples tightly with RDMA semantics and the segment tree's

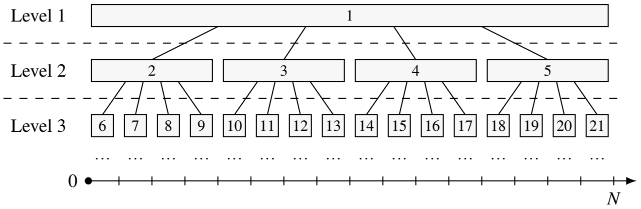


Figure 2: The structure and node indices of the lock tree.

layout, minimizing synchronization costs to nearly constant. Acquiring a lock requires a minimum of only two roundtrips.

Challenge 3. Real-world storage resources are not always fixed-size. Therefore, we must also efficiently handle a possibly dynamically growing storage size.

➥ **Runtime capacity expansion.** While segment trees are static, smaller trees can be seen as subtrees of larger ones. Citron leverages this characteristic to enable scaling up the tree's capacity at runtime using a few server-side CPU cycles.

3.2 Basic Assumptions

Address space. Citron maintains range locks within an abstract address space $[0,\infty)$. Multiple real-world scenarios fit in this model, e.g., LBA ranges in distributed NVMe-oF namespaces [62] and byte ranges in file systems [3, 47].

Cluster infrastructure. Aside from a lock server that hosts Citron's components (§3.3) in its DRAM, Citron requires that there is a cluster manager (CM) and a metadata server (MDS). The CM coordinates configuration changes (§3.5.5) and detects client failures (§3.10). The MDS maintains the addresses of Citron's components to enable the use of one-sided RDMA. The CM and the MDS need not run on independent machines: they can run on the lock server behind an RPC interface. There are already mature solutions for CM and MDS [17, 23, 48], so we need not discuss them here.

Clock well-behavedness. Citron assumes that the clocks of all clients are *well-behaved*, i.e., they advance at nearly the same speeds. Note that Citron does not require the clocks to be *synchronized*. Prior studies report that the clock frequency variation in a productional network is at most ±100 ppm [43] or even ±20 ppm when static errors are filtered out [53], which means that the clock drift is only up to ±0.1 ns or ±0.02 ns per microsecond, more than sufficient for Citron.

3.3 Components of Citron

Citron maintains range locks with a *lock tree* and a *spillover mutex*. The lock tree is responsible for locks within $[0,N)$, and the spillover mutex is for $[N,\infty)$, where N is specified at initialization time. Citron further includes a *maximizer* to enable clients to scale up the lock tree, i.e., to increase N.

Lock tree. The lock tree is a segment tree [4] – a perfectly balanced tree in which each node represents a continuous range. The root represents the entire range $[0,N)$; for every non-root node, it and its siblings each receive an equal and continuous share of the range represented by their parent. Such a structure determines that *the range represented by any node intersects only with its ancestors and its descendants*.

Orthodox segment trees are binary trees [4]. However, we define the lock tree in Citron as a quaternary segment tree, which means that the degrees (i.e., numbers of children) of all internal nodes are all four. Also, leaf nodes represent ranges of size 64, not the 1 in the orthodox definition. These designs aim to limit the tree height and, thus, the number of necessary RDMA verbs to post per lock request.

Since all internal nodes have the same degrees, there is no need for pointers in the lock tree. Instead, all nodes are placed in a continuous flat array by level order and indexed by positive integers (cf. heaps [74]). Tree navigation is simply node index arithmetics. For example, Figure 2 shows the lock tree's first three levels and the node indices, in which the widths of nodes correspond to their represented ranges. From this figure, we can easily verify that for a node with index x,

$$\textsc{Child}(x,i) = 4x-2+i \quad (i=0,1,2,3)$$
$$\textsc{Parent}(x) = \lfloor (x+2)/4 \rfloor$$

Spillover mutex. The spillover mutex represents $[N,\infty)$, i.e., it handles out-of-bound parts (w.r.t. the lock tree) of range lock requests. It can adopt any design that is friendly to one-sided RDMA. We use DSLR [79] to implement this mutex.

Maximizer. The maximizer is an initially-zero 8-byte variable accessible by one-sided RDMA. A client modifies this variable when it locks a range that is not contained within $[0,N)$. We will detail the usage of the maximizer in §3.9.

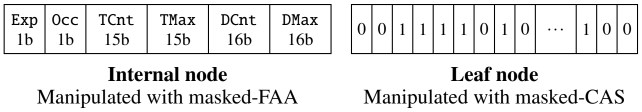


Figure 3: 64-bit representations of internal and leaf nodes.

3.4 Formats of Lock Tree Nodes

All nodes in the lock tree are 8-byte variables accessible by all kinds of RDMA one-sided verbs. Internal nodes and leaf nodes have different formats and are manipulated by different RDMA atomic verbs, as shown in Figure 3.

Leaf nodes. Leaf nodes are 8-byte bitmaps in which each bit is associated with a unit of the shared storage. A set bit in the bitmap indicates the corresponding unit of the resource occupied by some client, and vice versa. Clients use RDMA masked-CAS to set and clear each of the 64 bits.

Internal nodes. Each internal node divides into six fields. `Exp` and `Occ` are flags, and the remaining four are counters. Clients use RDMA masked-FAA to modify these fields.

`{TCnt,TMax}` and `{DCnt,DMax}` are two counter pairs that follow the idea of Lamport's bakery algorithm [38, 79]. Specifically, in each counter pair, `Max` is the next available

Algorithm 1 Acquire range locks from Citron

```
1: procedure AcquireRangeLock($l$, $r$)
2:     $A \leftarrow [l,r) \cap [0,N)$
3:     if $[l,r) \cap [N,\infty) \neq \varnothing$ then                ▷ Out-of-bound
4:         Acquire the spillover mutex
5:     if $A \neq \varnothing$ then                                   ▷ In-bound
6:         AcquireLockOnTree($A.left$, $A.right$)
```

"ticket number," and `Cnt` is the ticket number that is currently holding the lock. A client gets a ticket by performing an FAA on the `Max` field, polls the `Cnt` field until it matches the ticket, enters the critical section, and finally performs an FAA on the `Cnt` field when the client finishes. The two counter pairs are for different purposes: `{TCnt,TMax}` counts lock requests at "this node," while `{DCnt,DMax}` counts those at descendants.

As for the flags, `Exp` (stands for *expanded*) notifies clients of a lock tree scale-up event. `Occ` (stands for *occupied*) blocks conflicting lock requests at descendants if it is set. A node with the `Occ` flag set will be called an occupied node.

Like prior studies [79], the bit widths of counters impose a hard limit on the maximum concurrency of the system. There may not be more than $2^{15} - 1 = 32767$ clients accessing the same Citron instance concurrently; otherwise, the overflowing counters can put Citron into an erroneous state. Nevertheless, this restriction is tolerable in most scenarios.

3.5 Lock Acquisition

Algorithm 1 shows how a client acquires a lock on a range $[l,r)$. Since mutexes are already well-studied, here, we omit the details about the spillover mutex and focus on the lock tree. Without loss of generality, we now assume $[l,r)$ is fully contained within $[0,N)$. Algorithm 2 shows the whole lock acquisition procedure, which consists of two steps:

1. split the range properly into sub-ranges, such that each of which corresponds to a single tree node;
2. acquire locks on each sub-range in ascending order.

For each sub-range and the corresponding node on the lock tree (denoted as *node* hereinafter), the second step further decomposites into four phases:

2(a). lock *node* if it is internal;
2(b). wait until all locks at *node*'s ancestors are released;
2(c). lock *node* if it is a leaf, otherwise occupy it;
2(d). notify *node*'s ancestors and wait for its descendants.

Below, we elaborate on each of the two steps and the four phases of the second step. For convenience and readability,

- we call our protagonist "Alice": she is a client trying to acquire a range lock, and we describe what she will do;
- we use the adjectives *low* and *high* to describe tree nodes that are far from and close to the root;
- we describe masked-FAA with variadic arguments (a pair per field to FAA) instead of bitmasks (Line 3).

Algorithm 2 Acquire a range lock from the lock tree

```
1: ▷ Function signatures of RDMA masked atomic verbs ◁
2: def MaskedCAS($addr$, $cmp$, $cmpMask$, $swap$, $swapMask$) → boolean
3: def MaskedFAA($addr$, $field_1$, $add_1$, [$field_2$, $add_2$, [...]]) → uint64

4: procedure AcquireLockOnTree($l$, $r$, $k = 2$, $m = 4$)              ▷ Step 1
5:     $nodes \leftarrow$ SolveKnapsack($l$, $r$, $k$)
6:     for all $node \in nodes$ in ascending order do
7:         repeat $ret \leftarrow$ LockNode($node$, $l$, $r$, $m$) until $ret$ = Acquired

8: procedure LockNode($node$, $l$, $r$, $m$)                            ▷ Step 2
9:     if $node$ is internal then                                       ▷ Phase (a)
10:        $ticket \leftarrow$ MaskedFAA($node$, TMax, 1)
11:        repeat $val \leftarrow$ RdmaRead($node$) until $val$.TCnt = $ticket$.TMax
12:    $cleared \leftarrow node$                                         ▷ Phase (b)
13:    while $cleared \neq root$ do
14:        $\{anc\} \leftarrow$ RdmaRead(all ancestors of $cleared$)
15:        if $root$.Exp = 1 then return Aborted
16:        $next \leftarrow$ the lowest node in $\{anc\}$ with Occ $\neq 0$
17:        if $next$ = nil then break
18:        repeat $val \leftarrow$ RdmaRead($next$) until $val$.Occ = 0
19:        $cleared \leftarrow next$
20:    if $node$ is a leaf then                                          ▷ Phase (c)
21:        $mask \leftarrow$ bitmask of $[l,r) \cap node.range$
22:        if not MaskedCAS($node$, 0, $mask$, $mask$, $mask$) then
23:            if MaskedCAS kept failing for too long then
24:                return LockNode(Parent($node$), $l$, $r$, $m$)
25:            else
26:                goto Line 12
27:    else
28:        MaskedFAA($node$, Occ, 1)
29:    $t_0 \leftarrow$ current time                                     ▷ Phase (d)
30:    $\{anc_{\text{notify}}\} \leftarrow$ every $m$-th ancestor of $node$
31:    MaskedFAA($\{anc_{\text{notify}}\}$, DMax, 1), RdmaRead($root$)
32:    if possible time limit excess then return Aborted
33:    if $\{anc_{\text{notify}}\}.highest$.Exp = $root$.Exp = 1 then return Aborted
34:    if $node$ is internal then wait until $t_0 + T_{\text{wait}}$
35:    for $desc \in \{node$ and its internal descendants within $m$ levels$\}$ do
36:        repeat $val \leftarrow$ RdmaRead($desc$) until $val$.DCnt = $val$.DMax
37:    return Acquired
```

3.5.1 Step 1: Split the range

In this step, Alice decides which node(s) to lock. This step incurs zero network traffic because Alice knows the structure of the lock tree in advance and can do all computations locally.

With a segment tree, any continuous range can be expressed as an aggregate of $O(\log N)$ tree nodes [4]. As a result, Alice has to lock $\Theta(\log N)$ nodes to precisely lock the range $[l,r)$ in the worst case. However, this can result in high latencies since the nodes must be locked sequentially to prevent deadlocks.

A strawman solution is to simply lock the lowest node whose represented range completely covers $[l,r)$. However, this can result in severe false conflicts. For example, imagine that Alice wishes to lock $[N/2-1, N/2+1)$: the lowest node that covers this small range would be the root, which unfortunately conflicts with all other lock requests.

Citron strikes a balance by allowing Alice to lock up to k nodes that cover the requested range together. To reduce false lock conflicts, Citron tries to minimize the covered but unrequested range. This optimization goal can be formulated into a tree knapsack problem [37] and solved by existing algorithms. Our knapsack algorithm has a time complexity of $O(k^2 \log N)$,

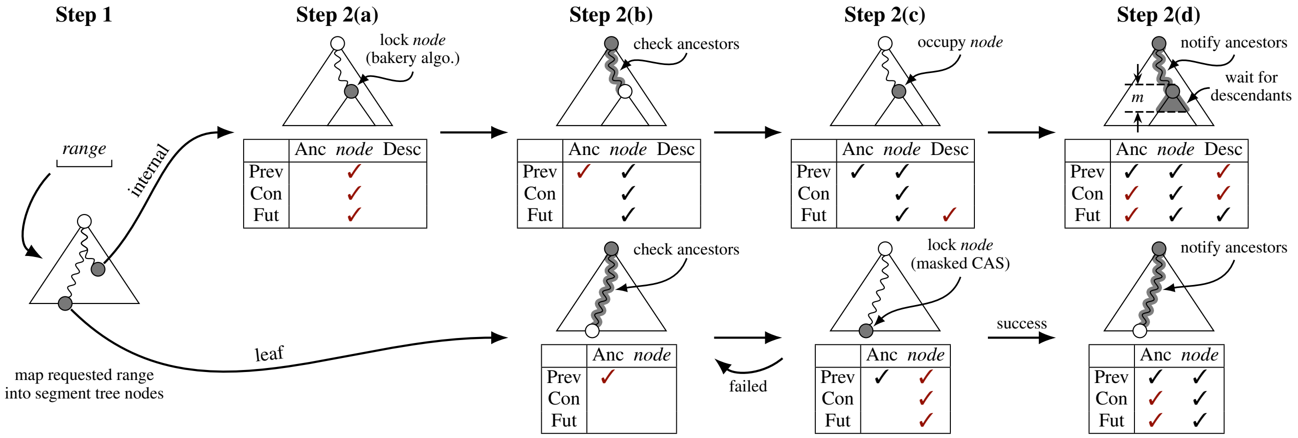


Figure 4: Demonstration of the lock acquisition workflow and the lock conflicts resolved by each phase. *Table rows are the time dimension (Prev = Previous, Con = Concurrent, Fut = Future), and table columns are the space dimension (Anc = Ancestors, Desc = Descendants). Lock conflicts occurring at any time and any position will all be resolved.*

which usually finishes within 0.5 µs when properly optimized and hardly harms lock acquisition performances. Increasing *k* trades latencies for fewer false conflicts and vice versa. In our implementation, we fix *k* to 2 for low latency purposes.

Procedure AcquireLockOnTree in Algorithm 2 shows how this step works. Alice first runs a knapsack algorithm to find the optimal combination of the nodes to lock (Line 5). Then, she locks the nodes sequentially (Lines 6-7).

3.5.2 Step 2(a): Lock an internal *node*

In this phase, Alice acquires the lock at *node* if it is an internal node. The workflow follows Lamport's bakery algorithm [79]. Specifically, Alice first increments the `TMax` field of *node* to get a "ticket." Then, she polls the `TCnt` field until it matches the `TMax` field of the ticket (Lines 10-11).

Alice does not lock *node* if it is a leaf. Instead, she defers locking *node* to Step 2(c) to facilitate failure recovery (§3.10). Were she to lock *node* here, Citron would be unable to recover to a normal state if Alice crashed before releasing her lock.

3.5.3 Step 2(b): Wait for *node*'s ancestors

From this phase, Citron starts to resolve conflicts between different nodes. The major principle is that among multiple concurrent lock requests, Citron prioritizes the smallest range because it is usually also the most latency-sensitive one.

As we have discussed before, all ancestors of *node* conflict with it. If there is a held lock at one of *node*'s ancestors, Alice must wait until it is released. Furthermore, there can be higher occupied ancestors of *node*, which belong to lock requests that arrive earlier than Alice's but are still waiting because Citron prioritizes smaller ranges. To ensure fairness, Alice should also wait for these lock requests to complete.

This phase consists of multiple iterations. In each iteration, Alice first reads the ancestors of *node* from the lowest one possibly occupied (Line 14) and checks their `Occ` flags to see if occupied nodes exist. If there are any, the lowest one is selected (Line 16). Alice waits until the lock at the selected node gets released (Line 18), which ends her current iteration. In the next iteration, Alice only needs to check the ancestors of the previously selected node (Line 19). She repeats this process until *node*'s all occupied ancestors are released.

Note that Alice cannot read the ancestors of *node* only once because other clients might issue new range lock requests to nodes higher than any existing lock. Due to unlucky timing, these clients can occupy the nodes they are locking without being aware of the lock requests below. The lock protocol ensures that these clients will get notified of all lock conflicts in Step 2(d) (§3.5.5), so there are no correctness concerns. However, Alice must repeatedly check *node*'s ancestors to detect these possible new lock requests.

3.5.4 Step 2(c): Occupy or lock *node*

Alice can ensure no held locks at *node*'s ancestors now. The remaining lock conflicts can only locate at *node*'s descendants for an internal *node*, or *node* itself for a leaf *node*.

If *node* is internal, Alice needs to set its `Occ` flag with an RDMA masked-FAA. The reason is that Alice should wait for lock requests at *node*'s descendants (because they are more prioritized than Alice's), but she must not wait indefinitely. By setting *node*'s `Occ` flag, newly arriving lock requests at *node*'s descendants will detect and wait for Alice in their Step 2(b), which ensures finite wait time for Alice.

If *node* is a leaf, Alice needs to post an RDMA masked-CAS verb to lock the corresponding bits of *node*. On success, Alice finishes this phase. On failure, she must return to the beginning of Step 2(b) because other clients could have set the `Occ` flags of *node*'s ancestors, as discussed above.

Starvation avoidance. When *node* is a leaf, a series of failed masked-CASs might cause lock starvation. Citron offers a workaround: if Alice keeps getting masked-CAS failures for a certain period, instead of returning to Step 2(b), she can optionally set *node* to its parent, restart the lock acquisition procedure, and switch to the starvation-free Lamport's bakery algorithm since *node* is now internal (Line 24).

3.5.5 Step 2(d): Notify ancestors, wait for descendants

Let *desc* be an arbitrary descendant of *node*. Assume Bob is another client that is trying to acquire a range lock at *desc* concurrently with Alice. Although Bob conflicts with Alice, they are both unaware of each other's existence. A mechanism is therefore needed to allow Bob to notify Alice of a lock conflict and also to allow Alice to detect this conflict.

There are two strawman solutions. In the first solution, Bob is responsible for notifying all ancestors of *desc*. However, this can result in high latencies for small range lock requests: the lower *desc* is, the more ancestors it needs to notify. In the second solution, Alice is responsible for checking all descendants of *node* for possible conflicts. However, this can result in excessive network traffic since the number of *node*'s descendants increases exponentially as *node* becomes higher.

Inspired by *meet-in-the-middle* (MITM), a common idea in computer science, we employ a combination of the solutions above to synchronize Alice and Bob. Specifically, Citron maintains a globally consistent parameter: m, the MITM distance. Starting from *desc*'s parent node, Bob notifies *desc*'s every m-th ancestor (Lines 30-31). Alice, on the other hand, checks all *node*'s descendants within m layers on the lock tree, as well as *node* itself (Lines 35-36). This solution ensures that no matter where *node* and *desc* locate, Alice will check a node that Bob notifies and thus detect the lock conflict.

For Bob, this solution reduces his notification overheads by a factor of m, which is efficient enough even with a small m. For Alice, she needs to read and check $(4^m - 1)/3$ nodes. Recall that the nodes of the lock tree are placed in the memory by level order and will only form m continuous blocks in the memory layout. Therefore, Alice only needs to post m RDMA reads, which is an acceptable cost. In our implementation, we set $m = 4$. Also, to avoid contention of RDMA atomic verbs, Citron does not notify nodes in the top $m-1$ levels of the lock tree except for *node*'s parent. Instead, Citron replaces them with nodes in the m-th level.

We still need to ensure that Bob notifies *desc*'s ancestors before Alice checks one of them. To this end, Alice waits for a period of time T_{wait} before she checks *node*'s descendants (Lines 29 & 34). Bob ensures that he finishes notifying *desc*'s ancestors before Alice stops waiting; otherwise, he aborts his lock request (Line 32). The foundations of this solution are (1) the well-behavedness of the clients' clocks and (2) the fact that the server-side RNIC executes inbound writes and atomics as if in a global order (i.e., linearizability).

For convenience, we assume an imagined global wall clock in the following discussion. Suppose Alice waits in the time interval $[t_0, t_0 + T_{\text{wait}})$, where t_0 is a global time point. Therefore, the deadline for Bob's notification is $t_0 + T_{\text{wait}}$.

Recall that Bob performs RDMA reads to ancestors of *desc* in Step 2(b) to check if there are any occupied nodes. From this phase, Bob can find a time point t_1 at which Alice has not started waiting. Specifically, if any RDMA reads find an occupied ancestor of *node*, t_1 is the time when Bob posts the last of those. Otherwise, t_1 is the post time of the last RDMA read in Step 2(b). Since Alice only starts waiting after she sets *node*'s `Occ` flag in Step 2(c), $t_1 < t_0$ must hold because of RDMA's linearizability. Say Bob finishes notifying *desc*'s ancestors at time t_2. Bob verifies that

$$t_2 - t_1 \le (1-\delta) \cdot T_{\text{wait}} \tag{1}$$

where δ is the bound of clock drift. Since Bob does not know where *node* locates at, he needs to record a t_1 and verify the equation above for every ancestor of *desc*.

Despite the non-existence of an imagined global wall clock, Bob can use his local clock to compute $t_2 - t_1$ because it is well-behaved. We use the number from Sundial [43] and set $\delta = 10^{-4}$. Bob will abort his lock acquisition process if Inequation (1) does not hold. The process of aborting a lock request is the same as releasing the lock, and we will detail the procedure in §3.6.

To decide T_{wait}, we count the maximum number of RDMA roundtrips from t_1 to t_2 and reserve a unit of time for each roundtrip. On our testbed, the RDMA RTT is around 2 µs; conservatively, we reserve 5 µs for each roundtrip. Therefore, $T_{\text{wait}} = 15$ µs: Alice waits for up to two RDMA reads in Step 2(b) and a batch of RDMA masked-FAAs in Step 2(c)+(d).

A complete Step 2(d). In the discussions above, we make Bob notify Alice and make Alice wait for Bob. However, we can imagine swapping the roles of Alice and Bob to see that they are actually symmetric clients and need to do what each other does. In other words, Alice needs to both notify *node*'s ancestors and wait for notification from *node*'s descendants.

Tuning the parameters. Step 2(d) relies on properly selected m and T_{wait} to perform well. Increasing either parameter will trade performance of large range lock requests for small ones, and vice versa. Therefore, clients can profile the performance of lock requests (e.g., throughput and lock abort rate, §4.7) and send the profiled data to the cluster manager (CM), enabling the CM to make tradeoffs and tune the parameters.

The CM can employ a two-phase commit (2PC) protocol to adjust the parameters. Suppose we wish to change (m, T_{wait}) from $(m_{\text{old}}, T_{\text{old}})$ to $(m_{\text{new}}, T_{\text{new}})$. The CM first broadcasts $(m_{\text{new}}, T_{\text{new}})$ to all clients. Upon receiving the parameters, a client acknowledges the CM and, in lock acquisition, uses

- $\min\{T_{\text{old}}, T_{\text{new}}\}$ to determine whether it should abort,
- $\max\{T_{\text{old}}, T_{\text{new}}\}$ when waiting for *node*'s descendants,
- $\min\{m_{\text{old}}, m_{\text{new}}\}$ when notifying *node*'s ancestors, and
- $\max\{m_{\text{old}}, m_{\text{new}}\}$ when checking *node*'s descendants.

After confirming that all clients have already received the new parameters, the CM sends "commit" messages to make clients switch entirely to $m = m_{\text{new}}$ and $T_{\text{wait}} = T_{\text{new}}$.

Algorithm 3 Release a range lock back to the lock tree

```
 1: procedure ReleaseLockOnTree($l$, $r$)
 2:     for all $node \in$ locked nodes do
 3:         if $node$ is a leaf then
 4:             $mask \leftarrow$ corresponding bitmask of $[l, r)$
 5:             MaskedCAS($node$, $mask$, $mask$, 0, $mask$)
 6:         else
 7:             MaskedFAA($node$, Occ, $-1$, TCnt, 1)
 8:         for all $anc \in$ notified ancestors of $node$ do
 9:             $val \leftarrow$ MaskedFAA($anc$, DCnt, 1)
10:             if $anc$ is the highest notified node and $val$.Exp $\neq 0$ then
11:                 Fetch and update the new tree configuration if needed
12:                 Continue the for all loop for new ancestors of $anc$
```

3.6 Lock Release

Algorithm 3 shows the lock release procedure. If *node* is a leaf, Alice unlocks it with masked-CAS (Lines 3-5); otherwise, she vacates *node* by adding the `TCnt` counter and clearing the `Occ` flag with masked-FAA (Line 7). Also, for all *node*'s ancestors that have been notified during lock acquisition, Alice adds their `DCnt` counters (Lines 8-9). All these RDMA verbs can be batched together to reduce latency.

3.7 Proof Sketch of Correctness

Range locks in Citron consist of nodes on the lock tree and possibly a spillover mutex, all of which are acquired separately and sequentially. The correctness of the spillover mutex is already proven [79]. Therefore, we only need to prove the correctness of a lock on a single tree node. Alice is still our protagonist in the proof sketch.

Safety. Safety means that Citron does not simultaneously grant locks to Bob – a conflicting client – and Alice. As shown in Figure 4, no matter when Bob arrives and where he locates on the lock tree, Citron will resolve the lock conflict between Alice and him. Specifically, Steps 2(a)+(c) ensures that no conflicting clients exist at *node*: 2(a) for an internal *node*, and 2(c) for a leaf *node*. Steps 2(b) and 2(c)+(d) ensure respectively that no held locks exist at *node*'s ancestors and descendants. Step 2(d) further ensures that:

1. if Bob is at a descendant of *node*, Alice will wait until he releases his lock or aborts;
2. if Bob is at an ancestor of *node*, he will wait until Alice releases her lock or aborts.

Therefore, when Alice enters the critical section, no conflicting held locks may exist. □

Liveness. Liveness means that without infinitely long critical sections, Alice's lock acquisition procedure will always take some finite amount of time to return (the result could be Aborted, though). Specifically, Step 1 is finite. Step 2(a) employs Lamport's bakery algorithm, which is starvation-free. Step 2(c) is obviously finite for an internal *node*, and is also finite for a leaf *node*, thanks to the starvation avoidance mechanism. For Steps 2(b) and 2(d), Alice can only wait for a finite number of clients in each phase. Because these clients will eventually abort or release their locks, these phases are also finite. To sum up, lock acquisition takes a finite time. □

3.8 Fast Path Optimization

Several optimizations apply to the lock acquisition path when Citron is not under severe contention.

First, RDMA ensures that it will not reorder any one-sided verb before previous writes and atomics in the same QP [66]. Thanks to this ordering guarantee, Alice can batch the RDMA verbs in the lock acquisition path together. In Step 2(b), all reads in an iteration can be batched (Line 14). In Step 2(d), the notification to *node*'s ancestors and the read to the root can be batched (Line 31). Further, Steps 2(a) and 2(b) can be optimistically batched in the hope that Step 2(a) immediately succeeds. More aggressively, Steps 2(c) and 2(d) can also be batched, but Alice needs to roll back the notification to *node*'s ancestors in Step 2(d) (by adding `DCnt`) if *node* is a leaf and the masked-CAS verb in Step 2(c) fails.

Second, if *node*'s children are all leaf nodes, Alice can explicitly lock all *node*'s descendants to skip the wait time in Step 2(d). Specifically, she post masked-CAS verbs to all its children to set all 256 bits from 0 to 1. If all these masked-CAS verbs succeed, she can skip the wait and directly enter her critical section. Otherwise, she needs to fall back to the regular lock acquisition path and also clear the bits of modified children nodes. The RDMA masked-CASs can be batched with the atomic verbs in Steps 2(c) and 2(d).

With these optimizations, optimistically, acquiring a lock takes only two RDMA roundtrips, ensuring low latencies.

3.9 Scaling the Lock Tree

In real-world scenarios, a shared storage resource can be of a dynamic size (e.g., append-only log). If the storage size grows, Citron's lock tree might be unable to cover the lock requests, which can cause performance degradation. On the other hand, if the storage size shrinks, maintaining unused nodes in the lock tree will result in extra memory consumption. Therefore, it is necessary for Citron to react to storage size changes.

3.9.1 Scale up

The size of a storage resource can grow upon writes. When this happens, out-of-bound lock requests not contained within $[0, N)$ will contend for the spillover mutex. Overprovisioning the lock tree will inflate Citron's DRAM footprint, of which a considerable percentage is wasted, while a stop-the-world synchronized scaling up mechanism will cause significant synchronization overheads. To solve this problem, leveraging the structural self-similarity of segment trees, i.e., small trees can be viewed as subtrees of large ones, Citron provides an option to scale up the lock tree at runtime.

Citron uses masked-CAS to decide how large the lock tree should scale up to. Note that *masked-CAS offers bitwise-OR semantics when the compare mask is zero*. For this reason, Citron contains a maximizer: the clients can OR the right

Algorithm 4 Scale up the lock tree

```
1:  procedure ScaleUpLockTree
2:      Acquire the spillover mutex
3:      Send an RPC to server to allocate free and zeroed memory
4:      for all $node \in$ top $m$ levels of the old lock tree, by index order do
5:          $v \leftarrow$ MaskedFAA($node$, Exp, 1)
6:          for $i = m, 2m, 3m, \ldots$ do
7:              $anc \leftarrow i$-th ancestor of $node$ in the new lock tree
8:              MaskedFAA($anc$, DCnt, $v$.DCnt, DMax, $v$.DMax + $v$.Occ)
9:      Update the metadata service to renew the lock tree configuration
10:     RdmaWrite($maximizer$, 0)
11:     Release the spillover mutex
```

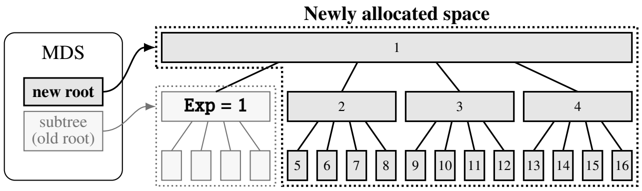


Figure 5: Demonstration of a lock tree scale-up process.

boundaries of out-of-bound lock requests to the maximizer with one-sided masked-CAS, enabling the detection of such lock requests. The maximizer's value is at most 2× of the actual maximum, which is accurate enough because the minimum scale-up factor of the lock tree is 4×.

If Alice is willing to scale up the lock tree, she can read the maximizer via RDMA and perform the scale-up if she finds a non-zero value. Algorithm 4 shows the procedure of scaling up. Alice first acquires the spillover mutex (Line 2) to both ensure lock safety and prevent simultaneous scale-up attempts. Then, Alice sends an RPC to the server to allocate the expanded part of the segment tree (Line 3). The original lock tree is not moved and will form a new enlarged tree with the newly allocated nodes, as shown in Figure 5. Alice then sets the Exp bits of all nodes in the top m levels of the old lock tree to notify other clients of the scale-up event (Lines 4-5). She also needs to propagate these nodes' DCnt and DMax counters to their new ancestors (Lines 7-8). Note that an occupied node accounts for an extra unit of DMax. Finally, Alice updates the metadata service with a renewed configuration containing addresses of both original and expanded parts of the lock tree, clears the maximizer, and releases the spillover mutex to finish scaling up the lock tree (Lines 9-11).

With off-the-shelf RNICs that do not support one-sided memory allocation, we must rely on the server-side CPUs to allocate memory and register it to the RNIC. However, this is a lightweight task compared with CPU-based lock management and can hardly cause any server-side CPU bottleneck.

Handling scale-ups in lock acquisition. When acquiring a lock (Algorithm 2), Bob, another client, must take into consideration that Alice can concurrently scale up the lock tree. Specifically, in Step 2(b), Bob checks the Exp flag whenever he reads the root (Line 15): a set Exp indicates a concurrent scale-up. In Step 2(d), Bob needs to read the root after notifying *node*'s ancestors. If the Exp flags of the highest notified ancestor of *node* and the root are both set (Line 33), there must be a concurrent scale-up. Bob handles concurrent scale-ups trivially: he aborts and retries.

Handling scale-ups in lock release. The lock tree can also be scaled up after Bob acquires a lock and before he releases it. Alice will notify the new ancestors of *node* on behalf of Bob. Therefore, Bob should also add the DCnt counters of the new ancestors of *node* (Algorithm 3, Lines 10-12).

Node index arithmetics. For a node x in the enlarged part of the lock tree, by simply offsetting the number of nodes in the old lock tree that lie ahead of x in the level order, the node index arithmetics rules described in §3.3 still hold.

Impact to Step 2(d) of lock acquisition. Scaling up the lock tree can break the continuous memory layout of the tree nodes in the memory, which can affect Step 2(d) of the lock acquisition path. To detect Bob, Alice originally only needs to post m RDMA reads, but with a scaled-up lock tree she will possibly need to post more. However, even in the worst case, the number of RDMA reads is only $m(m+1)/2$, which is still acceptable when m is small (e.g., for our $m = 4$ setting, the number of RDMA reads is 10) because all these reads can be parallelized. The extra reads can also be reduced by setting a larger minimum scale-up factor (e.g., 16×).

3.9.2 Scale down

Different to scaling up, scaling down cannot be triggered by writes and is in most cases intrinsically a blocking operation. For example, in file systems, calling `ftruncate` to shrink a file will take its inode mutex and block all other I/O attempts. During a blocking scale-down operation, Citron can safely shrink its lock tree by removing all nodes except a subtree.

3.10 Handling Client Failures

To enable recovery, all clients must agree with a lease time T_{lease} and that a range lock must be released within T_{lease}.

Detection. Citron relies on the cluster manager (CM) to detect client failures. The CM notifies the lock server to destroy the RDMA QPs that were connected with the failed clients.

Recovery. Citron recovers lazily. Alice detects a failure if she spins at a place for longer than T_{lease} during lock acquisition, including Lines 11, 18, and 36 in Algorithm 2. Also, Alice suspects a failure if she fails too many times at Line 22.

Line 11. Alice detects a failure when TCnt and Occ are both unchanged for T_{lease}. Shen then waits for up to $(\Delta - 1) \cdot T_{\text{lease}}$, where Δ is the gap between *node*'s TCnt and her ticket's TMax. If TCnt and Occ remain unchanged, Alice sets *node*'s TCnt field to her ticket's TMax and clears *node*'s Occ flag to recover.

Line 18. Alice detects a failure when TCnt of an ancestor of *node* is unchanged for T_{lease}. She aborts the current lock request and tries to lock that ancestor instead, reducing the problem to the situation of Line 11, which we have already discussed above.

Codename	Type	Lock Management Scheme	Description
`MT`	I	Maple tree [21]	A modern data structure dedicated to efficiently managing disjoint ranges, ported from Oracle Linux UEK.
`IT`	I	Interval tree [47]	A representative implementation of interval tree ported from Lustre, in which it is used to manage range locks upon file I/O requests.
`LLC`	I	Lock-free linked list [36]	A range lock manager that chains lock entries in a lock-free linked list.
`LLD`	II		Same as above, but all the CPU atomic instructions are replaced with RDMA one-sided atomic verbs to make the lock manager decentralized.
`SS`	II	Static segmentation [35]	The whole range is divided into fixed-size segments, each associated with a DSLR [79] instance, a state-of-the-art RDMA-based decentralized mutex.

Table 1: Baseline systems used in evaluation.

Line 36. Alice detects a failure when `DCnt` is unchanged for $H \cdot T_{\text{lease}}$, where H is *node*'s height in the lock tree. Since $H \geq 1$, Alice is sure that no clients are holding locks at *node*'s descendants and can set *node*'s `DCnt` to its `DMax` to recover.

Line 22. Alice cannot distinguish between lock starvation and client failures when she repeatedly fails to lock *node* with masked-CAS. However, she can acquire a lock at *node*'s parent and then check if *node* is zero. If not, Alice detects a failure and zeroes *node* with an RDMA write to recover.

The recovery time is dominated by the user-defined lease time T_{lease}. Aside from waiting for lease expiration, Alice only needs one RDMA operation to perform the recovery. In practice, T_{lease} is usually set to several milliseconds (e.g., 10 ms in [79]); with larger ranges, T_{lease} can also be longer.

4 Evaluation

In this section, we use a number of benchmarks to evaluate Citron, seeking to answer the following questions:

- How does Citron compare against existing lock managers? (§4.2, §4.3)
- What are the performance effects of the fast path? (§4.4)
- How well does Citron scale up itself? (§4.5)
- How does splitting ranges reduce false conflicts? (§4.6)
- What is the lock abort rate of Citron? (§4.7)

4.1 Experiment Setup

Our testbed consists of 4 machines, one acting as the lock server and the other as clients. Each machine is equipped with two Intel® Xeon® Gold 5220 CPUs running at 2.20 GHz, 256 GB DDR4-2666 DRAM, and a Mellanox ConnectX-6 RNIC via PCIe 3.0 ×16 interface. All machines run Ubuntu 18.04 with Linux kernel version 4.15.0 and are connected by a Mellanox QM8790 InfiniBand switch.

Lock tree configuration. Except in §4.5 (in which we need to scale up the lock tree), the lock tree is always initialized with $N = 2^{28}$. This is to simulate a large-scale scenario where 1 TB space is divided into 4 KB pages and managed by Citron. As a result, the lock tree contains 5.6 million nodes and the Citron instance takes up 42.7 MB of memory, which is only about 0.004% of the total storage amount.

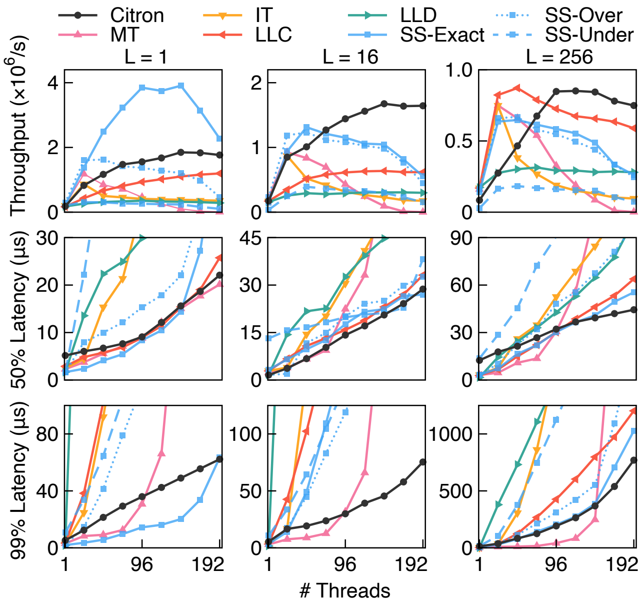


Figure 6: Throughputs and latencies of Citron and baseline systems with different range lock sizes.

Baseline systems. All baseline systems are shown in Table 1, which can be classified into the following two types.

I. Server-side CPUs are fully responsible for acquiring and releasing locks, and they accept clients' requests using eRPC [30], a state-of-the-art RDMA RPC engine.

II. Clients leverage one-sided RDMA to acquire and release range locks and server-side CPUs are idle.

The number of threads. The server machine runs 18 RPC server threads when evaluating baselines of type I. For clients, we enable hyperthreading and run up to 64 worker threads in each client machine, each thread on a separate logical core. As a result, the maximum number of clients is 3×64 = 192.

4.2 Microbenchmarks

In this experiment, we set the range sizes to $L = 1$, $L = 16$, and $L = 256$ respectively. For static segmentation (`SS`), we consider three different situations in terms of the segment size: (1)

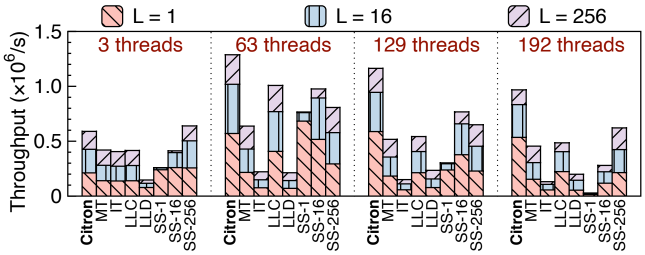


Figure 7: Throughputs by range size of Citron and baseline systems under a mixed-size workload.

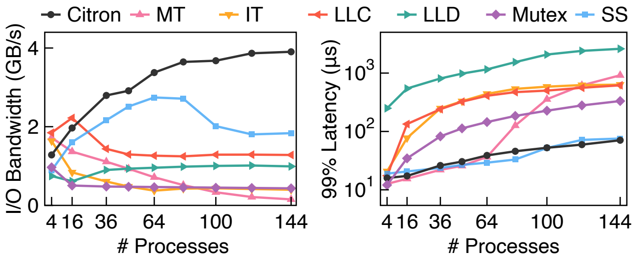


Figure 8: Throughputs and latencies of Citron and baseline systems under the BT-IO workload. *Latency is log scale.*

exactly the range size L (SS-Exact), (2) overestimated to $8L$ (SS-Over), and (3) underestimated to $L/8$ (SS-Under). The left borders of the requested ranges are subject to a Zipfian-0.9 distribution on $[0, N-L]$. Figure 6 shows the results.

In terms of median latency, in almost all cases, Citron performs comparably to the best of the baselines, namely MT, LLC, and SS-Exact. This matches our expectation because Citron needs a similar number of RDMA roundtrips to these baselines to acquire a range lock. We focus on the lock manager's tail latencies below.

When $L = 1$, range locks are equivalent to mutexes. As expected, Citron underperforms SS-Exact. It delivers 44.6% lower throughput (i.e., locks granted per second) and 1.83× higher p99 latency on average. However, this gap is because the access granularity is static and correctly known in advance. If this requirement is not met, the performance of SS will drop dramatically: SS-Over and SS-Under deliver peak throughputs of only 83.2% and 16.6% compared to that of Citron, and they suffer from 3.77× and 4.68× higher p99 latencies, respectively on average. The results demonstrate that the static segmentation mechanism is unfit for dynamic workloads whose I/O granularities vary.

When L is 16 or 256, because of unaligned ranges, static segmentation causes severe false lock conflicts and degrades performance. Citron delivers 28.7% and 38.6% higher peak throughputs and significantly lower tail latencies than SS.

Type I baseline systems that rely heavily on server-side CPUs are all bottlenecked by CPUs under high contention. Citron avoids such bottleneck and has 1.56× and 1.76× peak throughputs than these baselines for $L = 1$ and $L = 16$. When $L = 256$, Citron shows similar peak throughput to LLC but in average 24.6% lower tail latencies. Under low contention, due to the efficient eRPC engine and low CPU burdens, the queueing latencies are lowered to a sub-microsecond level and the baselines can show tail latency advantages to Citron. Unfortunately, such advantages vanish quickly as the number of clients increases.

LLC performs significantly better than LLD because their lock management scheme, i.e., the lock-free linked list, is CPU-friendly but RDMA-unfriendly. LLC performs pointer chasing which has very limited overheads on the CPU. However, with RDMA, each step of pointer chasing takes one RDMA roundtrip, leading to high latencies and low performance. This demonstrates the unfeasibility of simply porting existing range lock managers to one-sided RDMA.

We also test a mixed workload where each of the range sizes described above accounts for one-third of the client threads. For SS, we test three granularities: 1, 16, and 256. Figure 7 shows the results. Citron delivers higher throughputs than baselines under high contention: it outperforms the best baseline by 27.7%, 51.7%, and 55.9% with 63, 129, and 192 client threads, respectively. Also, Citron grants higher throughput to small range locks without starving large ones.

In summary, Citron delivers the overall best performance for different range sizes under high contention. However, Citron can be suboptimal for mutex-only workloads.

4.3 Application Benchmarks

We build a distributed in-DRAM file system CitronFS to evaluate Citron and baseline lock managers under realistic workloads. CitronFS follows Octopus's design [46] but stores all data in DRAM. It implements cacheless file I/O that can be protected by either per-file byte-range locks or inode mutexes. The server machine serves file metadata, while the three client machines stores file data.

4.3.1 BT-IO: a non-conflicting I/O workload

This experiment runs the Class D BT-IO [75] workload in the NAS Parallel Benchmarks [55] with different process counts. This application performs non-conflicting interleaved writes and reads to a total of 135.8 GB of data in a single file; different process counts lead to different I/O granularities ranging from 2040 B to 16320 B. Figure 8 shows the results.

With Citron, the I/O bandwidth of CitronFS reaches a maximum of 3.90 GB/s, which is 3.05× and 2.13× to those with LLC and SS-Exact, the best CPU-based and one-sided RDMA-based baselines, respectively. Citron outperforms LLC and SS-Exact by 1.89× and 1.52× on average. Compared with LLC and other CPU-based baselines, Citron delivers a 73.4% p99 latency reduction on average.

The underlying reason is that the range locks uniformly span the whole range because BT-IO is a non-conflicting workload. Therefore, CPU-based range lock managers need to maintain larger data structures for more concurrent lock

Lock Scheme	Throughput (kops/s)	Reader 99% Latency (µs)
Citron	781.4	59.6
MT	220.6	2043.4
IT	503.1	118.7
LLC	847.5	430.9
LLD	144.8	442.0
SS-4KB	575.3	56.1
Mutex	173.8	295.2

Table 2: Throughputs and latencies of Citron and baseline systems under the Filebench OLTP workload.

entries and perform memory (de)allocations more frequently, aggravating the CPU bottleneck. Citron, SS-1, and LLD avoid such bottlenecks by using only one-sided RDMA. Compared with LLD, Citron requires much fewer network roundtrips and has significantly lower latencies. Compared with SS-1, Citron leverages RDMA masked-CAS, which can obviate false lock conflicts and also minimize the number of locks to acquire, resulting in higher performances.

4.3.2 Filebench OLTP: a conflicting I/O workload

This experiment runs the Filebench [68] OLTP workload modified to run distributedly. This application runs reader and writer threads that operate on a dataset of 10 data files and a log file, all 10 MB sized. Each client runs 1 log file writer, 10 data file writers, and 50 readers. In each client, readers perform random 8 KB reads to data files, while writers perform 100 random 8 KB random writes evenly to all data files per 1000 reads and one 256 KB write to the log file per 3200 reads. We use 4 KB (i.e., page size) as the granularity for static segmentation (SS). Table 2 shows the results.

Overall, CitronFS delivers the second highest I/O throughput with Citron, 7.8% lower than that with LLC. However, LLC shows 7.2× p99 latencies compared with Citron, which demonstrates that Citron can avoid the CPU bottleneck by eliminating the RDMA CQ queueing latencies.

Another baseline system, SS-4KB, shows similar latencies to Citron for readers because they share similar numbers of necessary RDMA roundtrips to acquire and release a lock. However, Citron delivers 35.8% higher throughput for two reasons. First, when using SS-4KB, CitronFS is bottlenecked by the inefficient log writer that needs to acquire 64 mutexes to perform a write. Second, Citron is more friendly to the CPU cache because it reduces memory footprint by compressing lock entries into bitmaps, which brings higher performance thanks to Intel's Data Direct I/O technology [24].

4.4 Effects of the Fast Path

We measure the performance of Citron with and without the fast path optimization (§3.8) to understand its benefits. We use the same fixed-size microbenchmark as in §4.2 and set the range sizes to $L = 16$ and $L = 256$, respectively, to evaluate the fast path for both small and large ranges. The left borders of the requested ranges are subject to a Zipfian-α distribution on $[0, N-L]$, for which we adjust the skewness factor α from 0 (i.e., uniform) to 0.99 (i.e., highly skewed). We fix the number of threads to 96. Figure 9 shows the results.

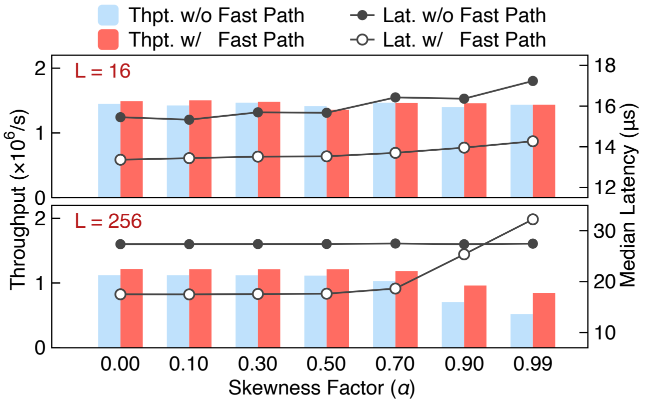


Figure 9: Performance effects of the fast path optimization.

For $L = 16$, the fast path does not significantly affect Citron's throughputs because it simply batches the RDMA verbs in Steps 2(c) and 2(d) in the lock acquisition workflow. However, by batching RDMA verbs together, the fast path saves a network roundtrip from the critical path, reducing the median latency by 2.4 µs (21.5%) on average.

For $L = 256$, the fast path contributes to higher throughput and lower latency. The fast path effectively eliminates the wait time in Step 2(d), which reduces lock acquisition latency and increases the throughput. On average, enabling the fast path improves the throughput by 34.3%. When $\alpha \leq 0.70$, the fast path reduces the median latency by 11.5 µs (39.4%). However, when most lock requests conflict with each other ($\alpha > 0.70$), the fast path lengthens the critical path and wastes RDMA IOPS, resulting in increased median latency (4.8 µs, 17.4% higher than that without the fast path when $\alpha = 0.99$). Note that non-conflicting lock requests still benefit from the fast path, which brings higher throughput.

Also, we observe that the fast path shows no significant impact on the p99 latencies. The reason is straightforward: the tail latencies stem from lock requests that cannot benefit from the fast path. We omit the results due to limited space.

4.5 Performance with Scale-ups

We use a trace collected from the hard-write workload of the IO500 benchmark [25] to evaluate the scale-up process of Citron. In this workload, 64 I/O threads repeatedly write 47 008 B data to a large shared data file in parallel. Offsets of the writes continue to increase, resulting in a constantly growing file size. The whole trace consists of 12.8 million writes. We only acquire and release range locks without performing writes to avoid shadowing the impacts of scale-up events.

We initialize Citron with $N = 2^{10}$ (i.e., 4 MB size, has scale-ups) and compare the results with $N = 2^{28}$ (i.e., no need for scale-ups). When $N = 2^{10}$, each client machine runs a

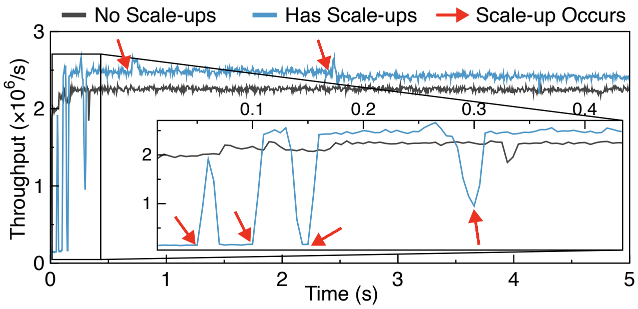


Figure 10: Lock/unlock-only throughput of Citron with and without scale-ups under the IO500 hard-write workload.

background thread that polls the maximizer once per 10 ms and scales up the lock tree whenever necessary. The server runs one eRPC thread to serve lock tree metadata queries and memory allocation requests. Figure 10 shows the results.

The scale-up process takes only tens of microseconds to complete and hardly blocks other lock requests. Hence, upon a scale-up, there will be an immediate increase in the throughput, as shown in Figure 10. In the first 300 ms of the experiment, the whole lock tree is still small despite being scaled up. The write offsets quickly grow beyond its size, causing the clients to contend for the sole spillover mutex and thus a throughput decline of up to 92.6%. After that, however, the throughput decline before the 4th scale-up is only 57.5%. The reason is that the lock tree is already large enough, and client threads are not perfectly synchronized; therefore, only a part of the threads contend for the spillover mutex. The throughput keeps almost stable afterward for similar reasons.

It is worth noting that the stable throughput with scale-ups is slightly higher than that without. Specifically, before and after the 6th scale-up, the throughput advantages are 9.7% and 7.4%, respectively. The reason is that the lock tree only grows larger when necessary, resulting in a smaller tree height, reducing the number of RDMA verbs per lock request, thus bringing higher performance.

4.6 False Conflict Rate

We measure the false conflict rate of Citron to understand the effects of our range splitting mechanism in Step 1 of the lock acquisition path (§3.5.1). Two lock requests constitute a false conflict if they do not overlap but lock conflicting nodes. We measure the false conflict rate with different L (i.e., requested range lock size) and different k (i.e., the maximum number of nodes to lock per request). The left borders of the requested ranges are independently subject to a uniform distribution on $[0, N-L]$. We repeatedly issue two concurrent lock requests and detect whether they conflict with each other logically and actually. The false conflict rate is calculated as the number of false conflicts divided by the number of all lock requests. Figure 11 shows the results.

For all $L \leq 64$, $k = 2$ is sufficient to split the requested range into leaf nodes on the lock tree, eliminating false conflicts because Citron employs RDMA masked-CAS. As a result, Citron can achieve its highest throughput for prevalent small range lock requests in real-world workloads.

Increasing k beyond $k = 2$ brings minor benefits. Compared with $k = 1$, setting $k = 2$ reduces the false conflict rate by two orders of magnitude (to relatively 3.6% on average), whose absolute value is around 10^{-4}, virtually negligible. To further reduce this rate for an order of magnitude, we need $k = 5$, which results in 3 more nodes to lock and more than doubled lock acquisition latencies. Therefore, we trade that marginal throughput improvement for lower latency and adopt $k = 2$ in our implementation of Citron.

4.7 Lock Abort Rate

We measure Citron's lock abort rate to understand the efficacy of the synchronization mechanism in Step 2(d) of the lock acquisition path (§3.5.5). Low abort rates indicate the strong practicability of Citron. Aside from hardware issues such as the RNIC capabilities, the abort rate can be affected by the following three configurable factors:

1. *#Threads*: the thread count (i.e., contention severity),
2. m: the meet-in-the-middle distance in Step 2(d), and
3. T_{wait}: the time to wait in Step 2(d).

Therefore, we conduct three experiments, in each of which we fix two of these parameters, adjust the remaining one, and measure the lock abort rate. We retry for each aborted lock request until it succeeds, so the abort rate also reflects the amount of the retry traffic. We set the fixed parameters to *#Threads* = 96, $m = 4$, and $T_{\text{wait}} = 15\,\mu\text{s}$, respectively, as is described in §3. We use the same mixed-size microbenchmark as in §4.2. The abort rate is calculated as the number of lock aborts divided by the number of all lock requests. Figure 12 shows the results.

***#Threads*.** As the number of threads increases from 3 to 192, the RNIC suffers from an increased IOPS pressure and therefore delivers higher latencies, causing the lock abort rate to increase from 10^{-5} level to 10^{-2} level. However, the overall throughput (i.e., successful locks) does not drop with the increase in the abort rate after reaching the maximum. This shows that Citron's lock protocol causes acceptable numbers of lock aborts and retries under both low and high contention.

***m*.** When m is small, clients need to notify many ancestors of *node* in Step 2(d) with RDMA masked-FAA, resulting in low throughput and a high possibility of lock aborting. When m is large, the burden to notify *node*'s ancestors is low, but the networking cost to detect conflicts at *node*'s descendants suffers from exponential growth. We adopt $m = 4$ to balance throughput, abort rate, and network traffic.

T_{wait}. Increasing the wait time in Step 2(d) reduces the chance that lower clients exceed the time limit but makes higher clients wait longer and degrades throughput. We adopt T_{wait} =

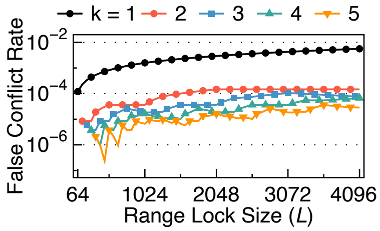


Figure 11: False conflict rate under different k settings. *Log scale.*

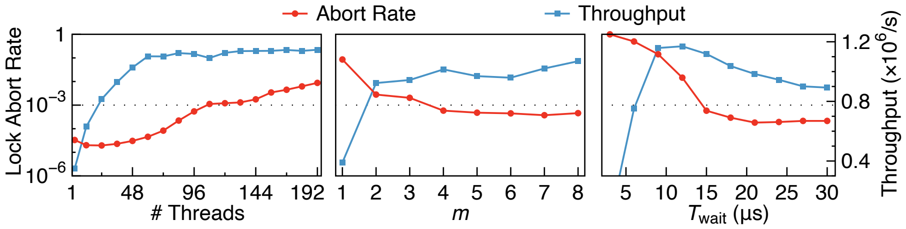


Figure 12: Lock abort rate and throughput of Citron under different workloads and configurations. *Lock abort rate is log scale.*

15 µs to balance between throughput and abort rate. Further increasing T_{wait} contributes little to reducing the abort rate since there are always occasional long-lasting RDMA verbs due to unpredictable hardware-level issues of the RNIC.

To sum up, with our configuration, the lock abort rate of Citron is acceptably low and has minimal negative effects on the overall performance.

5 Related Work

Lock management. Locks have been a major research topic ever since the outset of concurrent programming. A wealth of previous studies aim for efficient locking within a single machine [7, 8, 14, 15, 33, 34, 45, 52].

With the advent of RDMA, studies above have become less valuable in distributed systems as they fall short in alleviating the CPU bottleneck. Such a situation led to the birth of decentralized [10, 54, 72, 79] and hardware-offloaded [28, 80] lock managers. Among them, DrTM [72] uses RDMA CAS to grant writer locks and reader leases. DSLR [79] uses RDMA FAA to implement the starvation-free Lamport's bakery algorithm [38]. NetLock [80] offloads lock management to a programmable switch, achieving both high performance and the benefits of centralized lock management.

While most existing studies focus on mutexes, Citron aims at range locks and supports locking disjoint parts of the same shared storage for finer-grained concurrency.

Range locks. Range locks are widely adopted in key-value stores [18, 61], file systems [3, 9, 35, 47], and memory management systems [21, 36]. These systems usually use carefully designed *dynamic* tree data structures to manage range locks, including the range tree in RocksDB [18], the interval tree in Lustre [47], the red-black tree in BeeGFS [3], and the maple tree in Oracle Linux UEK [21]. Kogan et al. also propose using a lock-free linked list to maintain range locks [36] since the number of cores is limited and the list cannot be too long.

Citron targets distributed range lock management where far more clients exist than within a single machine. Citron avoids the CPU bottleneck by using only one-sided RDMA on the critical paths of range lock operations.

Lock conflict resolution. Allowing more types of communication aside from direct one-sided RDMA between the clients and the server brings different lock conflict resolution means. For example, Sherman [70] proposes a hierarchical lock scheme that maintains a local lock table within each client machine to avoid unnecessary remote retries and enable lock handing-over, which is also applicable to Citron. Thakur et al. proposes maintaining a lock table entry in the lock server for each client, thus enabling a lock holder to read the whole lock table and wake up conflicting clients when it releases the lock [69]. Other prior research [12, 63, 67] also discusses work delegation among clients to eliminate conflicts.

One-sided RDMA systems. In addition to decentralized lock management, existing studies employ one-sided RDMA for various purposes, including file I/O [2, 46, 77, 78], transaction processing [16, 64, 71–73], and memory disaggregation [1, 6, 19, 40, 50]. A recent study, RedN [66], even proves the Turing-completeness of one-sided RDMA and shows its efficacy in RNIC-offloading multiple functionalities.

Citron shares the same goals with most one-sided RDMA systems: eliminating server-side CPU bottlenecks and improving performance. However, Citron is the first to develop an efficient distributed range lock manager with one-sided RDMA and to outperform the state-of-the-art.

6 Conclusion

We present Citron, a distributed range lock manager that relies only on one-sided RDMA to acquire and release locks. Citron employs a lock protocol that operates a segment tree and efficiently coordinates conflicting range lock requests. Citron together offers a fast path optimization and supports dynamic scaling as the size of its managed range changes. Our evaluation shows that Citron significantly outperforms existing distributed range lock managers.

Acknowledgment

We sincerely thank our shepherd Youjip Won for helping us improve the paper. We are also grateful to the reviewers of this paper for their helpful comments and 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. 61832011 & 62022051), and Huawei (Grant No. YBN2019125112).

References

[1] 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 2020*, pages 1–16, Heraklion Greece, April 2020. ACM.

[2] Thomas E Anderson, Simon Peter, Marco Canini, Jongyul Kim, Dejan Kostic, Youngjin Kwon, Waleed Reda, Henry N Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, page 18. USENIX, November 2020.

[3] BeeGFS - The Leading Parallel Cluster File System. https://www.beegfs.io/c/.

[4] John Louis Bentley and Derick Wood. An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, July 1980.

[5] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, March 2016.

[6] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with RDMA and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, July 2018.

[7] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High performance locks for multi-level NUMA systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '15)*, pages 215–226, San Francisco CA USA, January 2015. ACM.

[8] Milind Chabbi and John Mellor-Crummey. Contention-conscious, locality-preserving locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*, pages 1–14, Barcelona Spain, February 2016. ACM.

[9] Qi Chen, Shaonan Ma, Kang Chen, Teng Ma, Xin Liu, Dexun Chen, Yongwei Wu, and Zuoning Chen. SeqDLM: A Sequencer-based Distributed Lock Manager for Efficient Shared File Access In a Parallel File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '22)*, page 11, Dallas TX USA, November 2022. IEEE/ACM.

[10] Yeounoh Chung and Erfan Zamanian. Using RDMA for Lock Management. *arXiv:1507.03274 [cs]*, July 2015.

[11] Giuseppe Congiu, Sai Narasimhamurthy, Tim Süß, and André Brinkmann. Improving Collective I/O Performance Using Non-volatile Memory Devices. In *2016 IEEE International Conference on Cluster Computing (CLUSTER '16)*, pages 120–129, September 2016.

[12] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48, Farminton Pennsylvania, November 2013. ACM.

[13] A. Devulapalli and P. Wyckoff. Distributed Queue-based Locking using Advanced Network Features. In *2005 International Conference on Parallel Processing (ICPP '05)*, pages 408–415, June 2005.

[14] Dave Dice and Alex Kogan. Compact NUMA-aware Locks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, Dresden Germany, March 2019. ACM.

[15] David Dice, Virendra J Marathe, and Nir Shavit. Lock cohorting: A general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*, page 10, February 2012.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle WA USA, April 2014. USENIX.

[17] etcd Authors. Etcd. https://etcd.io/, 2022.

[18] Facebook Open Source. RocksDB | A persistent key-value store. http://rocksdb.org/.

[19] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient Memory Disaggregation with InfiniSwap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*, page 21, March 2017.

[20] A.B. Hastings. Distributed lock management in a transaction processing environment. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems (SRDS '90)*, pages 22–31, October 1990.

[21] Liam Howlett. Introducing the Maple Tree. https://lwn.net/Articles/884840/, February 2022.

[22] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas F. Wenisch. A Top-Down Approach to Achieving Performance Predictability in Database Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 745–758, Chicago Illinois USA, May 2017. ACM.

[23] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC '10)*, page 14, Boston MA USA, June 2010. USENIX.

[24] Intel Corportation. Intel® Data Direct I/O Technology. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html, 2012.

[25] IO500 Foundation. IO500. https://io500.org/pages/running.

[26] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*, pages 1–12, November 2012.

[27] Tianyang Jiang, Guangyan Zhang, Zhiyue Li, and Weimin Zheng. Aurogon: Taming Aborts in All Phases for Distributed In-Memory Transactions. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*, pages 217–232, Santa Clara CA USA, February 2022. USENIX.

[28] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soule, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, page 16, Renton WA USA, April 2018. USENIX.

[29] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock Immunity: Enabling Systems To Defend Against Deadlocks. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, page 14, San Diego, California, USA, December 2008. USENIX.

[30] Anuj Kalia and David Andersen. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, page 17, Boston MA USA, February 2019. USENIX.

[31] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 185–201, Savannah GA USA, November 2016. USENIX.

[32] Qiao Kang, Scot Breitenfeld, Kaiyuan Hou, Wei-keng Liao, Robert Ross, and Suren Byna. Optimizing Performance of Parallel I/O Accesses to Non-contiguous Blocks in Multiple Array Variables. In *2021 IEEE International Conference on Big Data*, pages 98–108, December 2021.

[33] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, pages 586–599, Huntsville Ontario Canada, October 2019. ACM.

[34] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, page 15, July 2017.

[35] June-Hyung Kim, Jangwoong Kim, Hyeongu Kang, Chang-Gyu Lee, Sungyong Park, and Youngjae Kim. pNOVA: Optimizing Shared File I/O Operations of NVM File System on Manycore Servers. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19)*, pages 1–7, Hangzhou, China, 2019. ACM Press.

[36] Alex Kogan, Dave Dice, and Shady Issa. Scalable range locks for scalable address spaces and beyond. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, pages 1–15, Heraklion Greece, April 2020. ACM.

[37] Stavros G. Kolliopoulos and George Steiner. Partially-ordered knapsack and applications to scheduling. In Rolf Möhring and Rajeev Raman, editors, *Algorithms — ESA 2002*, pages 612–624, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[38] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[39] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write optimization of log-structured flash file system for parallel I/O on manycore servers. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 21–32, Haifa Israel, May 2019. ACM.

[40] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. Hydra: Resilient and Highly Available Remote Memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*, page 19, Santa Clara, CA, February 2022.

[41] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data*, pages 355–370, San Francisco California USA, June 2016. ACM.

[42] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. LocoFS: A loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Denver Colorado, November 2017. ACM.

[43] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant Clock Synchronization for Datacenters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, page 17. USENIX, November 2020.

[44] Zhen Liang, Johann Lombardi, Mohamad Chaarawi, and Michael Hennecke. DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory. In Dhabaleswar K. Panda, editor, *Supercomputing Frontiers*, Lecture Notes in Computer Science, pages 40–54, Cham, 2020. Springer International Publishing.

[45] Ran Liu, Heng Zhang, and Haibo Chen. Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, page 13, June 2014.

[46] Youyou Lu, Jiwu Shu, Tao Li, and Youmin Chen. Octopus: An RDMA-enabled Distributed Persistent Memory File System. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, page 15, Santa Clara, CA, July 2017. USENIX.

[47] Lustre® Filesystem. https://www.lustre.org/.

[48] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. INFINIFS: Efficient metadata service for Large-Scale distributed filesystems. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST '22)*, Santa Clara CA USA, February 2022. USENIX.

[49] Xiaosong Ma, M. Winslett, Jonghyun Lee, and Shengke Yu. Improving MPI-IO output performance with active buffering plus threads. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.

[50] Hasan Al Maruf and Mosharaf Chowdhury. Effectively Prefetching Remote Memory with Leap. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*, page 16. USENIX, July 2020.

[51] Mellanox Technologies. Mellanox Connect-IB® Firmware Release Notes. https://network.nvidia.com/pdf/firmware/ConnectIB-FW-10_16_1200-release_notes.pdf, 2017.

[52] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[53] Ali Najafi and Michael Wei. Graham: Synchronizing Clocks by Leveraging Local Clock Properties. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*, page 15, Renton WA USA, April 2022. USENIX.

[54] S. Narravula, A. Marnidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 583–590, May 2007.

[55] NASA Advanced Supercomputing (NAS) Division. NAS Parallel Benchmarks. https://www.nas.nasa.gov/software/npb.html.

[56] NVIDIA Corporation. Advanced Transport. https://docs.mellanox.com/display/MLNXOFEDv531001/Advanced+Transport.

[57] NVIDIA Corporation. NVIDIA ConnectX-5 InfiniBand Adapter Cards Datasheet. https://nvdam.widen.net/s/pkxbnmbgkh/networking-infiniband-datasheet-connectx-5-2069273.

[58] NVIDIA Corporation. NVIDIA ConnectX-6 Datasheet. https://nvdam.widen.net/s/5j7xtzqfxd/connectx-6-infiniband-datasheet-1987500-r2.

[59] NVIDIA Corporation. NVIDIA ConnectX-7 Datasheet. https://nvdam.widen.net/s/m6pt7j5rlb/networking-datasheet-infiniband-connectx-7-ds---1779005.

[60] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, and Abhinav Sharma. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, page 16. USENIX, February 2021.

[61] Percona LLC. PerconaFT. https://github.com/percona/PerconaFT, March 2022.

[62] Scott Peterson. Adaptive Distributed NVMe-oF Namespaces. https://www.snia.org/educational-library/adaptive-distributed-nvme-namespaces-2020, September 2020.

[63] Darko Petrović, Thomas Ropars, and André Schiper. On the Performance of Delegation over Cache-Coherent Shared Memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, pages 1–10, Goa India, January 2015. ACM.

[64] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118, Portland Oregon USA, June 2015. ACM.

[65] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 51–63, Chicago Illinois USA, May 2017. ACM.

[66] Waleed Reda, Marco Canini, Dejan Kostic, and Simon Peter. RDMA is Turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)*, page 15, Renton WA USA, April 2022. USENIX.

[67] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Ffwd: Delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 342–358, Shanghai China, October 2017. ACM.

[68] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework For File System Benchmarking. *USENIX ;login:*, 41(1):6–12, 2016.

[69] Rajeev Thakur, Robert Ross, and Robert Latham. Implementing Byte-Range Locks Using MPI One-Sided Communication. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3666, pages 119–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[70] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1033–1048, Philadelphia PA USA, June 2022. ACM.

[71] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, page 20, Carlsbad CA USA, October 2018. USENIX.

[72] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*, pages 87–104, Monterey California, October 2015. ACM.

[73] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM. In *Proceedings of the 2021 USENIX Annual Technical Conference*, page 16. USENIX, July 2021.

[74] John William Joseph Williams. Algorithm 232 - Heapsort. *Communications of the ACM*, 7(6):347–348, June 1964.

[75] Parkson Wong and Rob F. Van der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4, 2003.

[76] Cong Yan and Alvin Cheung. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment*, 9(5):444–455, January 2016.

[77] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memories and RDMA-Capable Networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, page 15, Boston MA USA, February 2019. USENIX.

[78] Jian Yang, Joseph Izraelevitz, and Steven Swanson. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *Proceedings of the 17th USENIX Symposium on Networked System Design and Implementation (NSDI '20)*, page 17, Santa Clara CA USA, February 2020. USENIX.

[79] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*, pages 1571–1586, Houston TX USA, May 2018. ACM.

[80] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*, pages 126–138, Virtual Event USA, July 2020. ACM.

Patronus: High-Performance and Protective Remote Memory

Bin Yan, Youyou Lu, Qing Wang, Minhui Xie, and Jiwu Shu*

Department of Computer Science and Technology, Tsinghua University

Abstract

RDMA-enabled remote memory (RM) systems are gaining popularity with improved memory utilization and elasticity. However, since it is commonly believed that fine-grained RDMA permission management is impractical, existing RM systems forgo memory protection, an indispensable property in a real-world deployment. In this paper, we propose PATRONUS, an RM system that can simultaneously offer protection and high performance. PATRONUS introduces a fast permission management mechanism by exploiting advanced RDMA hardware features with a set of elaborate software techniques. Moreover, to retain the high performance under exception scenarios (e.g., client failures, illegal access), PATRONUS attaches microsecond-scaled leases to permission and reserves spare RDMA resources for fast recovery. We evaluate PATRONUS over two one-sided data structures and two function-as-a-service (FaaS) applications. The experiment shows that the protection only brings 2.4 % to 27.7 % overhead among all the workloads and our system performs at most ×5.2 than the best competitor.

1 Introduction

Remote memory (RM) architecture, which decouples CPU and memory into two independent resource pools (i.e., compute nodes and memory nodes), is changing the landscape of modern data centers by providing many benefits, such as high memory utilization and efficient memory sharing [2, 12, 44]. This trend is sparked by the widely-deployed RDMA network, which allows compute nodes to access remote memory (at memory nodes) in a one-sided and low-latency manner. There are myriad efforts to make RM systems practical on multiple fronts, such as proposing easy-to-use programmable models [1, 43, 46], designing efficient remote indexes [50, 59], and deploying popular applications [38].

However, there is still an obstacle to cross on the way to practical RM systems: *remote memory protection*. Existing RM systems expose all RM resources or coarse-grained memory regions to compute nodes without carefully considering protection [2, 12, 15, 25, 31, 32, 36, 39, 41]. This inevitably induces several anomalies. First, buggy or malicious code in clients[1] can generate illegal one-sided access to the RM, introducing data corruption or privacy breaches. Second, even if the clients are well-behaved, concurrent memory reallocations can turn the in-flight one-sided access illegal (§3.1).

It is non-trivial to simultaneously achieve protection and high performance in RM systems. First, considering the high throughput of RDMA networks (e.g., ~70Mops/s in 100Gbps ConnectX-5 RDMA NIC), clients will frequently acquire/revoke permission upon memory allocation/deallocation. But the common RDMA protection mechanism, i.e., (re)-registering memory region (MR) to targeted memory areas, suffers high latency due to the overhead from OS kernel and RNIC (~1 ms for 256 MB; see Figure 1). Even worse, RM systems typically only have weak computing power at memory nodes [48, 55, 59], which limits the rate of acquiring/revoking permission, thus bottlenecking the system performance. Second, on the exception path of RM systems, i.e., clients fail or access illegal RM addresses, retaining high performance with a protection guarantee is challenging. Specifically, when a client fails, it may hold exclusive access permission to some memory areas. If the failed client's permission cannot be revoked rapidly, the progress of the whole RM system will be negatively impacted. When a client accesses illegal RM addresses, RDMA NICs (RNICs) at memory nodes will turn the associated queue pair (QP) into an error state, disabling subsequent RM access. Recovering the faulted QP needs a millisecond-scaled process and thus produces latency spikes for RM applications.

In this paper, we propose PATRONUS, a *protective* RM system that can provide high performance. In the control path, memory nodes perform memory (de)-allocation and bytewise memory protection for clients using weak computing power (i.e., ≤ 4 CPU cores). In the data path, clients at compute nodes access RM with permission via one-sided RDMA verbs. PATRONUS attains efficiency on both normal and exception paths. This is achieved by combining advanced RDMA hardware features and careful software design.

To enable fast permission management with weak computing power on memory side, PATRONUS first exploits *memory window (MW)* [42], an advanced RDMA hardware feature allowing RNICs to regulate the access (thus supporting one-sided RDMA) while minimizing the overhead of interaction with RNICs. Different from MR, an MW operation communicates with RNIC asynchronously and in userspace. With permission bits modified by hardware, it enjoys low latency (1.1 µs; see Figure 1). However, simply using MW cannot meet the performance requirements at peak load. Thus, we introduce a set of software techniques (e.g., MW handover and delayed unbinding; §5.4) to reduce the number of MW operations, saving the computing cycles of memory nodes.

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

[1] Clients are processes in compute nodes accessing RM.

To react fast to client failures, we equip MWs with *microsecond-scaled* leases, so that the permission will be automatically reclaimed by memory nodes on timeout. However, the fine-grained leases introduce the overhead of frequent extension to memory nodes. We reduce the extension overhead by delegating the management of lease metadata to the client with one-sided verbs while retaining the protection guarantee.

To mitigate the negative effect of illegal access (i.e., QP faults), instead of recovering the faulted QP in the foreground, we switch to another intact QP as a substitution. To avoid QP creation in the critical path, PATRONUS prepares a small number of spare QPs.

We evaluate PATRONUS thoroughly over microbenchmarks and two sets of realistic applications, i.e., the remote one-sided data structures (ODS) and the function-as-a-service (FaaS) platform. Among all the workloads evaluated, the protection only brings 2.4 % to 27.7 % overhead, and PATRONUS performs up to ×5.2 better than all the competitors. On the exception path, we reduce the interruption from faulted QP by 92 %. The lease semantics ensures the progress of the system under client crashes, evaluated under the case of ODSs.

Contributions. The main contributions are:

- An analysis of the deficiency of existing protection mechanisms and the performance goals for a protective RM system (§3).
- The design and implementation of PATRONUS, a protective RM system that retains high performance on both normal and exception path (§5).
- The thorough evaluations over microbenchmarks and realistic workloads to demonstrate the high performance of PATRONUS (§7).

2 Background

2.1 RDMA and Access Protection

RDMA is a high bandwidth (e.g., 200 Gbps) and low latency (~2 µs) networking technology widely adopted in today's data centers [12, 13, 20]. RDMA provides two types of verbs to the application, namely *one-sided verbs* and *two-sided verbs*. The one-sided verbs offer a remote memory abstraction; it allows *direct access* to the remote memory while bypassing remote CPUs. The two-sided verbs offer a *message-passing* interface similar to the well-known Linux socket. The two types of verbs make different trade-offs: the one-sided verbs are efficient for saving computation resources, but they risk data corruption for the lack of remote CPU regulation; the two-sided verbs are vice versa. The one-sided verbs are more prevalent due to their higher efficiency (i.e., ×1.7 throughput) [52].

Access protection. RDMA provides basic mechanisms for regulating RDMA verbs, e.g., *queue pair (QP)* and *memory region (MR)*. QP is the communication endpoint on which the client posts RDMA requests (via `ibv_post_send`); it offers channel-wise restrictions on the access type (i.e., readable or writable). MR represents a memory area registered to the RNIC for remote access; it restricts both the access type and the accessible range of memory.

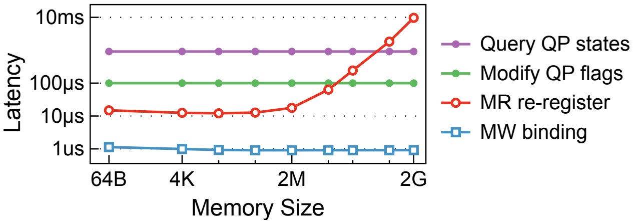


Figure 1: Median latency of protection-related operations in RDMA.

The MR/QP operations, in the RDMA control path, have orders of magnitude higher latency than the microsecond-scaled RDMA data path (Figure 1, [53]). Specifically, modifying QP flags includes a transition of QP states in the RNIC, taking ~100 µs per operation. The MR registration is synchronous and requires kernel involvement (e.g., context switches, page pre-faulting, and page pinning); it yields a non-scalable performance [3] with ~1 ms latency for a 256 MB area. Due to the inferior performance, most RM systems only involve QP constructions and MR registrations on bootstrap [12, 34, 53].

2.2 The Remote Memory Architecture

RDMA is the key enabler to the remote memory (RM) architecture for its ultra-low latency in interconnecting. RM is getting prevalent in the decade because it addresses the problem of memory usage imbalance in traditional data centers [6, 14]. With RM, the CPU and memory are assembled into two separate components, i.e., the compute nodes (CN) and the memory nodes (MN). The compute nodes gather a mass of CPU cores (10s - 100s), while the memory nodes typically have weak and limited computing power [1, 59]. The scarce computation power on MNs catalyzes a range of *RM-native applications* that mainly leverage one-sided verbs, such as KV stores [25, 36, 48], transactional systems [12, 52, 55], and data structures [4, 35, 50, 51, 58, 59]. These various workloads coexist in the cluster and share the remote memory.

3 Motivation

3.1 The Call for Stray Protection in RM

The efficiency of RDMA one-sided access comes at a price: its direct nature saves the overhead of remote CPU but in turn escapes the *protection* against *stray access*. The stray access is the illegal one towards an area of memory unowned by the process. Next, we show two causes for the stray access.

Causes of stray access. (*i*) Buggy and malicious codes. A careless bug or a piece of malicious code can generate stray access to the RM by setting an overflowed address in the RDMA request. This is a *space* anomaly that can occur when RM exposes a larger range of memory space than allowed. (*ii*) Race to memory management. The memory deallocation and reallocation make all the unaware one-sided access towards the address stray. This is a *time* anomaly that can occur when RM exposes a longer duration of permission than the application logically allowed.

In response to the causes, a protective system needs to expose only the range of memory that is allowed to access, and invalidate the permission timely after access is finished.

Cases for protection and requirements. We observe two trends making in-RM protection more urgent, posing requirements for a protective system. (*i*) The RM architecture catalyzes a wide range of remote *one-sided data structures (ODS)* [4,35,50,51,58,59], which involve frequent memory (de)-allocations and floods of concurrent access, bringing a high risk of memory management race at runtime. Moreover, the access to the ODS is typically shared and fine-grained (e.g., at a granularity of buckets in the hash table), which requires fine-grained protection for proper access isolation. (*ii*) In the function-as-a-service (FaaS) platform, functions submitted from different users leverage the shared RM for performant (intermediate) data storage [26,53], which asks for access isolation to avoid data tampering or leaks. Functions have a short lifetime (~µs), scale out quickly (to ~millions), and access RM on demand [9, 14, 27, 33], which requires a low-latency and high-throughput protection management to meet performance needs in the critical path.

To conclude, a protective system needs to offer high-performance protection management in fine granularity.

3.2 Goals for the Protective RM System

Considering that the RM system is an infrastructure to determine the overall performance of workloads, it should remain efficient in a variety of situations. Besides offering fast permission management on the normal path, it should be able to retain performance even under client failures or illegal access. Next, we elaborate on the performance goals.

Goal#1: manage protection fast. Existing workloads can introduce a mass of permission requests to the system in the peak case, demanding high throughput in permission management. For example, the bulk load to a remote hash table [59] brings a flood of concurrent permission acquisitions. Querying to the hash table involves multiple access to disjoint memory areas (e.g., the bucket and the KV block), introducing multiple permission acquisitions from one query. Therefore, we expect a protective system to offer *high-throughput* protection management to avoid introducing bottlenecks.

Goal#2: react fast to client failure. A client can affect the progress of the whole system if it crashes with exclusive permission held. This is common because clients are deemed error-prone in the distributed system, and access to the metadata should be exclusive in many workloads. Therefore, we expect that a protective system can react fast to client failures.

Goal#3: retain performance under illegal access. The illegal access turns the QP into an error state, in which the QP rejects any incoming RDMA requests, causing a serious interruption in application running. The interruption will further affect other innocent clients on the same QP (sharing QPs is very common under QP virtualization and is widely adopted for mitigating the scalability problem [12,49,53]). Therefore, we expect a protective system to retain performance in the appearance of illegal access.

Name	Goal#1	Goal#2	Goal#3	For RM
Two-sided	-	✓	✓	✗
MR	✗	✗	✗	✓
QP	✗	✗	✗	✓
MW	✓	✗	✗	✓
MW + *SW* (Patronus)	✓✓ (§5.4)	✓ (§5.3)	✓ (§5.5)	✓

Table 1: Deficiency of existing solutions. Goals are elaborated in §3.2. Only MW with *software (SW)* co-design meets all the goals (Patronus).

3.3 Deficiency of Existing Solutions

Existing solutions are all deficient for a protective RM system (Table 1). The two-sided verbs do not work well in the RM architecture where computation power is scarce on the memory node. Other existing solutions that regulate one-sided access (i.e., MR and QP, §2.1) can not simultaneously meet the three goals. In this section, we revisit these mechanisms and examine their applicability.

For protection management (G#1), the QP-based solution is coarse-grained and the MR-based solution is slow. The QP-based solution is channel-wise; it is unable to offer byte-wise protection as workloads require. Therefore, its use is very limited [3]. The MR-based solution has a high latency (~20 µs per 2 MB[2], Figure 1) and does not scale; due to the performance issue of MR, existing systems do not utilize its protection at runtime. For example, Octopus [34] registers MRs on bootstrap and never manipulates them later; FaRM [12] uses a large memory region of 2 GB, which essentially leaves the whole region unprotected.

For detecting client failures (G#2), MR is not aware of any failures on the remote side. Although QP does reveal remote failures (by issuing RDMA operations as heartbeats), it does not detect failures that leave QPs intact (e.g., clients using virtualized QPs as communication channels and clients getting hanged).

Finally, in the appearance of illegal access (G#3), whether violating the permission restricted by QP or MR, the QP will run into an error state, requiring an expensive bootstrap procedure to recover the QP (~1 ms, §7.2.5).

We conclude that *pure* hardware solutions are insufficient to achieve all the goals.

[2] We use 2 MB huge pages, a widely-adopted solution to reduce RNIC's page translation cache misses [12,52].

4 Approach Overview

4.1 Opportunity: Memory Window (MW)

The *memory window (MW)* [42] is an advanced RDMA feature widely supported in commodity RNICs. It acts as a supplementary layer over MR to deliver flexible protection management at runtime.

Interface. The MW needs to be allocated before use. It supports two types of operations, i.e., *bind* and *unbind*[3]. Binding an MW over a memory area exposes the access permission while unbinding the MW invalidates it. Note that we can bind an MW multiple times after it is allocated; each time the previous permission will be invalidated and the new permission will be granted (also called *rebind* in this paper).

Binding (rebinding) an MW takes the address and size of the memory area and the access type (read or write) as parameters. The MW exposes the memory area by generating an *rkey* (a 32 bit integer) as the permission token to the client, like in the case of MR. MW is *byte-granularity* in that it works for unaligned memory areas of any size. Binding/unbinding the MW uses the same `ibv_post_send` interface as RDMA verbs, which communicates with the RNIC in userspace asynchronously and allows requests batching.

Latency: MW vs. MR. MW binding contrasts with MR registration in two ways. First, MW binding has a *constant* latency with memory areas of any size, unlike MR registration taking proportional overhead to the size. Second, MW binding has much lower latency, i.e., 1.1 µs in median (Figure 1).

The reason for the performance difference is that MW binding communicates to the RNIC asynchronously in userspace, while MR registration is synchronous and requires kernel involvement[4], introducing the additional overhead of context switches, page pre-faulting, and page pinning.

4.2 Solutions

Although MW accelerates permission modifications, it does not introduce new features beyond MR. Therefore, MW also has limited functionality as MR. We observe that to achieve the goals, direct adoption of MW is not enough, and *software co-design* must be involved. Next, we show how software techniques are developed to fill the gap.

Can software further contribute to the overall performance? (G#1) Although MW already acts as a low-latency mechanism to manage permission, the overhead of MWs can still burden the memory nodes where CPUs are limited. We observe that software co-design can exploit the true potential of hardware for efficient permission management.

Solution: save MW operations without sacrificing protection semantics. Instead of improving performance by lowering the protection guarantee, we reduce overhead in a way the protection assurance is not sacrificed. This is possible by leveraging the characteristics we find in management. For example, by noticing that permission requests come in a batch, we can leverage the pairs of *opposite* operations to reduce the number of MW operations effectively by half (called *MW handover*). By noticing that some memory areas will not be re-used immediately after being freed, we delay the unbinding of the MWs to save operations. Finally, by exploiting the potential of address contiguity, we can combine multiple MWs into one. They are elaborated in §5.4.

How to react fast to client failure? (G#2) Like MR, MW itself is not aware of any failures from the CN side. Extra techniques must be developed to detect and handle the failure.

Solution: borrow the idea of leases. MW only offers space-wise protection. We introduce the *lease semantics* (i.e., expire on timeout, [41]) to MW from the software to enable time-wise protection. In doing so, the system can resume progress by expiring any exclusive permission on timeout, no matter whether the permission is held by a crashed or a slow client.

The lease management metadata seems too crucial to be exposed. Nevertheless, we notice the *byte-granularity* property of MW, which allows us to expose only the necessary part of metadata to the client. With this help, we are able to offload part of the lease management overhead to the client without risking metadata tampering (*CN-collaborated extension*, §5.3). It saves CPU cycles for the memory nodes.

How to retain performance under illegal access? (G#3) Like MR, MW protects against data corruption but *does not* protect the QP from running into an error state, which seriously interrupts application running.

Solution: conceal the interruption rather than prevent it. We notice that illegal access is unable to prevent because the memory node invalidates the permission (i.e., unbinds the MW) without notifying the client. Therefore, we try to conceal the interruption caused by the faulted QP instead of preventing it. We prepare *spare QPs* to stand in for the faulted ones at runtime so that the recovery overhead can be concealed in the background. In doing so, we leveraged a special property of MWs: they can remain valid across all QPs[5]. Therefore, the granted permission remains valid even if the underlying QP has changed. They are elaborated in §5.5.

5 Patronus: The Protective RM system

Motivated by how stray access is common and necessary to be prevented (§3.1), we design a protective RM system in response, called Patronus, to offer complete protection with sufficient performance for existing workloads (§3.2).

Different from previous systems where the whole remote memory is exposed to the clients [12, 34], the basic idea of Patronus is leaving clients with *no initial permission* and demanding permission acquisition before allowing clients to issue remote access.

[3] To distinguish, we use the verb *bind* for MW and *register* for MR.

[4] Although MR supports on-demand paging (ODP), which can remove page pinning, it is notorious for causing high latency (>=10 ms) on normal RDMA access upon remote page faults [22].

[5] Precisely, MWs work across all QPs under the same *protection domain (PD)*.

Category	API	Parameters	Return	Description
Control Path	`allocate`	*size, time, ex/shr*	*Perm*	Allocate memory and acquire permission
	`acquire`	*addr, size, time, r/w, ex/shr*	*Perm*	Acquire permission over $\langle addr,\ size \rangle$
	`extend`	*Perm, time*	*Success*	Extend the permission lease
	`revoke`	*Perm*	*Success*	Revoke the permission
Data Path	`read/write/` `CAS/FAA`	*Perm, addr, size, buffer*	*Success*	Issue remote access (support batching)

Table 2: The PATRONUS APIs. In the parameters, *r/w* specifies read/write permission; *ex/shr* specifies exclusive/shared access mode; *time* specifies the expected lifetime of the permission lease. *Perm* is an opaque object containing the remote address, the `rkey` as the permission token, and the expiration time. *Success* denotes whether the call succeeded.

Failure model and leases. PATRONUS considers two kinds of failures for the client, i.e., fail stop and fail slow, both addressed by leases. First, the lease ensures availability on client crashes (fail stop). The *orphaned* exclusive permission held by a crashed client precludes other clients from accessing the memory, resulting in unavailability. The lease resumes system progress by expiring the permission on timeout. Second, lease accelerates memory reclamation with slow clients (fail slow). A slow client (e.g., due to network traffic) hinders *actual* memory reclamation, because the active permission it holds makes the memory area potentially accessible and thus not reclaimable. The lease forcibly invalidates permission on timeout to allow reclamation on time. We assume loosely-synchronized clocks for the lease to work, like similar work [19]. PATRONUS does not handle the failure of memory nodes, where orthogonal work (e.g., erase coding) can be applied [30, 57].

5.1 The Interface

PATRONUS provides control path APIs to acquire new permission, extend the permission lease, and revoke permission. The data path APIs accept the permission as a parameter and are translated into one-sided verbs for remote access (Table 2).

Permission starts in two cases. (*i*) Client allocates remote memory via the `allocate` call. (*ii*) Client attempts to access a known remote area for the first time, in which case the client needs to get the permission via the `acquire` call. Both calls will issue an RPC to the memory node, where the memory node starts new permission by binding MWs to the allocated/specified memory, and responds with a `Perm` object to the client. `Perm`, needed by all the data path API, contains the permission token (`rkey`s of the MWs), the expiration time, and the remote address. Note that re-access to the same memory can re-use the previous `Perm` as long as it has not expired.

In the parameters, the client specifies the access mode (read-/write), ownership (shared/exclusive), and expected lifetime for the permission lease. For acquisitions that conflict in the ownership, PATRONUS postpones granting the latter permission until the conflict resolves.

PATRONUS allows pre-allocation to amortize the overhead; i.e., clients call `allocate` at a larger granularity and back their fine-grained allocators on the blocks. Nevertheless, it does not speed up the queries or in-place updates from other clients (which is also common), because those clients still need to call `acquire` for their own fine-grained permission.

Permission extends via the `extend` call. Extending an existing permission is more efficient than re-acquiring a new one. We assume that clients only extend the *hot* permission that is re-used frequently.

Permission ends when the client explicitly revokes the permission (via `revoke`) or when the lease expires. The `revoke` will issue an RPC. The to-expire leases are detected by periodical scans from the memory node. In both cases, the memory node unbinds the MWs to invalidate the permission.

Data path. PATRONUS purely uses one-sided verbs in the data path (`read`, `write`, `CAS`, and `FAA` calls). It supports batch execution and therefore allows the familiar IO consolidation optimizations in the application [59].

5.2 Architecture Overview

PATRONUS provides a library for the client in compute nodes (*CN-lib*) and a *manager* daemon for memory nodes, as illustrated in Figure 2. The manager manages the remote memory and permission in response to clients' RPC.

Main components. PATRONUS manager takes over the whole memory on MN. Most of the memory will be exposed for the client's use; we call them *buffers* in the memory pool. The other ($\leq 0.02\,\%$) is reserved on bootstrap to use as the *header pool*.

The header is a central structure that stores all necessary metadata for a permission. Each individual permission (possibly over the same memory area but owned by different clients) has an individual header. The header contains two kinds of information (Figure 3). (*i*) The resource information, i.e., the address and size of the RM buffer and the corresponding MWs. (*ii*) The lease information, such as when the permission starts and how long the permission will last. We use the address of the header as the cluster-wise permission *identifier* in the RPC between clients and the manager.

Permission management. The start of permission is triggered by the client's `allocate` or `acquire` calls. In response, the manager allocates a header for the new permission and then binds *two* MWs to expose both the buffer and the header to the client (❶ in Figure 2). The header is additionally exposed

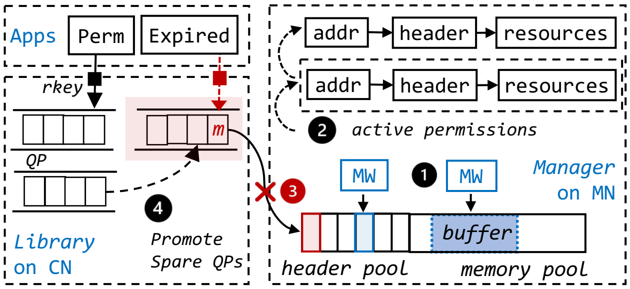


Figure 2: The architecture of PATRONUS. We assume that CNs own a mass of CPU cores (10s - 100s), while MNs use several weak cores to operate the manager.

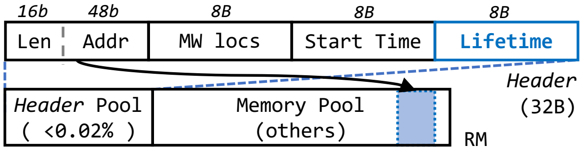


Figure 3: The format of the 32 B *header*. `MW locs` denotes the locations of the MWs. Blue indicates the area exposed to the client, i.e., the *Lifetime* variable in the header and the buffer in the RM.

so that the client can facilitate permission management, a technique called *CN-collaborated extension* (§5.3).

The remote memory is managed by slab allocators with different object sizes, similar to FaRM [12]. Permission has the same granularity as memory management: clients must start permission over the whole object, but not a part of it. The manager uses a per-slab hash table to store all the active permission, with addresses as the keys; in this way, permission can be queried efficiently. To detect any to-expire permission, the manager periodically polls the hash table to collect the timeout ones (❷ in Figure 2). The polling overhead is minor because the number of active permissions in the system is typically small.

To invalidate permission, the memory node unbinds the MWs, which in turn invalidates the permission token (`rkey`), causing RNIC to forcibly reject the RDMA requests with that `rkey` (❸ in Figure 2).

5.3 CN-collaborated Extension

Considering that unexpected permission expiration seriously interrupts application running, PATRONUS allows extending the permission on runtime. The naive approach is using an RPC to notify the manager of the extension. However, the communication brings significant overhead to the memory node where computation power is scarce. To mitigate this overhead, we propose to utilize the collaboration from the CNs for extension while handling careless and malicious clients correctly.

Collaboration from CN. The metadata in the header seems too crucial to be exposed. Nevertheless, we notice that MW is *byte-granularity*; therefore, it can be used to expose only the necessary part of metadata to the clients without risking metadata tampering.

The basic idea is to expose the *lifetime* variable in the header (Figure 3) so that the client can update it in a *one-sided* way. In turn, the manager will encounter extended permissions on polling for the timeout ones; the manager skips them.

Regulate the extension. The CN collaboration can introduce the starvation problem in the system without proper regulation. Specifically, (*i*) the client is able to set *lifetime* to a large value to own it infinitely. (*ii*) The client is also able to extend continuously, starving other clients.

In response, we propose two regulations for the extension. First, we require that any permission can not live beyond a predefined maximal lifetime (empirically set to several milliseconds). Any aged permission will be detected by the manager and be forcibly invalidated.

Second, to avoid starving other clients, we need an efficient way to notify the owner that a permission is no longer extendable. In PATRONUS, the notification is implemented by setting the lifetime to *zero* and requiring the CN-lib to update the lifetime with `RDMA_CAS` instead of `RDMA_WRITE`. In this way, the zeroed lifetime causes `RDMA_CAS` to fail and thus the clients are notified. Note that the permission is arbitrated on the memory node, which means that the manager can always reclaim the permission by forcibly invalidating the MWs if it suspects any anomalies, without negotiating with the client.

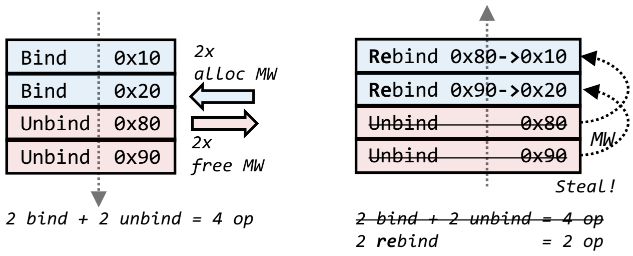
Trade-off analysis. With collaboration, the overhead of an RPC (the naive approach) is reduced to one inbound one-sided access. The collaboration benefits the performance because (*i*) one-sided verbs are more efficient than two-sided ones, and (*ii*) inbound verbs are more efficient than outbound ones [25]. The benefit will enlarge if extensions occur multiple times.

Exposing the lifetime variable introduces the overhead of using one more MW. Nevertheless, we deem the overhead minor compared to the naive approach (taking one RPC), because the additional MW operations can be batched together in the `ibv_post_send` API, communicating with the RNIC once. Furthermore, as we show in §5.4, this extra MW overhead can be reduced most time.

Polling: the alternative to lease. An alternative approach to the lease semantics is *QP polling*, where MNs track whether CNs are still alive by periodically issuing RDMA operations to each QP as heartbeats. Polling has several deficiencies compared to leases. First, it does not handle fail slow of clients. Second, polling can not distinguish clients sharing the same QP (while QP sharing is common [12, 49, 53]). In terms of overhead, polling and leases both pay one RDMA operation for keepalive; however, leases allow to save one `revoke` call by letting the permission expires itself, potentially yielding better performance.

5.4 Reduction of Permission Overhead

In this section, we introduce our techniques for reducing the permission overhead *without* sacrificing protection assurance.



(*a*) Before

(*b*) w/ MW Handovers

Figure 4: The *MW handover* technique saves half of MW operations by stealing (reusing) MWs from the unbinding requests to the binding requests. 0x*dd* denotes the *addr*.

We elaborate on the characteristics we find in protection management and how we leverage them to reduce MW operations.

#1: Leverage pairs of opposite operations. Every active permission ends eventually. Therefore, among all the MW operations in the system, about half of them are binding while the others are unbinding. Following this fact, we can combine every pair of opposite MW operations into one *rebinding* operation, a technique we call *MW handover* (Figure 4). Rebinding an MW, which takes the new $\langle addr, size \rangle$ as parameters, generates new rkeys and invalidates the old rkeys (§4.1). To adopt this technique, the manager collects opposite operations with the best effort: the to-expire permission will be polled and clients' requests will be scanned before performing the handover.

Trade-off analysis. The benefit of this technique comes with no price because handover is only performed in a speculative way. First, the manager never waits for future requests; no latency is introduced. Second, client requests on the memory nodes are naturally batched by the RNIC, which is a hardware approach and does not introduce extra batching overhead. The memory node, accordingly, always handles requests in a batch, leaving room for performing handover.

#2: Delay unbinding if memory is not re-used. We observe that if a memory area is not re-used after deallocation, we can delay unbinding the MW because stray access to this area does not introduce data corruption. The header is a good candidate for doing so: we reserved more-than-enough headers in the system, so it is easy enough not to re-use the just-deallocated header in the near future. If the available memory buffers are adequate in the system, we also delay unbinding the buffer MW; however, if it is not the case, we unbind the buffer MW and reclaim (thus re-use) the buffer promptly.

Trade-off analysis. This technique wastes available headers and MWs but in a minor way. The waste of headers is negligible because we reserve adequate headers in the pool for the extreme case. We carefully encode the header so that the pool occupies no more than 0.02 % of a regular memory node (§5.6 for details). The waste of MWs is trivial because the RNIC (Mellanox CX-5 in our case) allows ~16 million MWs, far from possibly being used up.

#3: Exploit the potential of contiguity. We observe that if two addresses are contiguous and share the same protection lifetime, we can combine the two MWs into one. At first glance, the situations of this case are rare because addresses are generally *not* contiguous and memory buffers seldom share the same protection lifetime. We exploit this potential in handling the allocate RPC: we can allocate an extra 32 B to place the header right before the buffer; thus, the header and the buffer are contiguous (this case is not revealed in the figures for brevity). While doing so, we carefully place the to-expose variable (i.e., the *lifetime* variable in Figure 3) at the tail of the header to make the to-expose areas contiguous.

Trade-off analysis. The benefit comes with no price. At first glance, allocating an extra header introduces a lot of 32 B holes. However, these holes are not wasted, because they can be re-used as headers again when the permission over the same buffer is re-acquired. This situation is very common; for example, inserts to the remote hash table involve allocations of KV blocks. These KV blocks will be re-accessed when being read or modified. In this case, the following permission acquisitions can re-use the holes as headers again. On deallocation, the frontal 32 B will be reclaimed altogether; thus they are not leaked.

5.5 Isolation from Illegal Access

Although MW can prevent illegal access from corrupting the memory, it does not handle the consequence of it: the illegal access will turn the underlying QP into an error state. The faulted QP requires an expensive procedure for recovery (~1 ms), seriously interrupting application running.

We observe that this interruption is not preventable by the software. This is because process scheduling and network traffic can introduce a nondeterministic delay to the one-sided request. During the delay, the permission may have expired. Therefore, we propose to *conceal* the interruption rather than prevent it.

Conceal the interruption. We prepare spare QPs to conceal the interruption caused by QP failure. Specifically, each client is assigned a virtual QP number, which maps to a physical QP initially. On QP failure, we transparently promote one of the spare QP by altering the virtual-to-physical mapping (❹ in Figure 2). In this way, the client can resume its execution immediately. A special property from MWs enables continuous execution, i.e., the MWs are able to remain valid across all QPs. This wide validity allows the previous permission to remain valid in the new QP. Therefore, the client does not need to re-acquire permission when QP switches.

A small number of spare QPs are sufficient to hide the foreground interruption as the manager performs QP recovery in the background. We assume a low illegal access rate (less than 1-10s per second) compared with the speed of QP recovery (~1 Kops).

Trade-off analysis. The spare QPs introduce no overhead for the normal path. The reason is that the spare QPs, while inactive, will not contend for the rare RNIC resources (e.g., the limited cache [37]).

The spare QPs consume host memory but in a negligible way. To adopt this technique, considering that we use the peer-to-peer RC (reliable connection) type of QP, each memory node needs to prepare $O(C)$ spare QPs for connection, where C is the number of compute nodes. It does not cost much, because even for a large cluster with one thousand CNs, preparing 3 QPs for each CN only consumes ~1 MB host memory (each QP takes ~375 B; [40]).

5.6 Implementation Details

MW pool. The allocation of MWs, unlike binding, has a much higher latency (1 µs vs. 100 µs). We maintain an MW pool to offload the allocation off the critical path.

Header encoding and overhead. The header only takes up 32 B after our effort on data encoding. We leverage the *tagged pointer* [59] to steal the higher 16 bit for the buffer size (*Len*), which is able to present 0-64 KB. For the case where larger buffers are common, we use a scale factor for *Len*, e.g., 64 or 4096. Since we need to locate two MWs (i.e., header and buffer) for each permission, we store *two* 4 B MW indexes to locate MWs in the MW array. *Start time* and *lifetime* are encoded in microseconds; 8 B is ample to encode any time in theory. In the extreme case where 1 million permissions are simultaneously present in the system (clients own very few active permissions in general), the headers only occupy 32 MB in total (< 0.02 % with 128 GB memory). In conclusion, the memory consumption is negligible.

Handling double invalidation. Without careful management, double invalidation of permissions may occur in the system, caused by the famous *ABA problem*. Specifically, the ABA problem comes when an obsolete RPC tries to locate the permission header that has already been re-used. To address this problem, the *start time* is additionally attached with the permission identifier in each RPC. The manager filters out any RPCs whose start time does not match.

6 The Cases for PATRONUS

In this section, we demonstrate the benefits of PATRONUS through case studies. We explain the way to adopt PATRONUS to these cases individually.

6.1 One-sided Data Structure

We mainly focus on two one-sided data structures (ODS), i.e., the start-of-the-art RACE hashing [59] and a concurrent queue [17]. Other ODSs, such as the hashing-based ones [51], the tree-based ones [50, 58], and the skip list [35], are similar.

The RACE hashing is an RDMA-conscious extendible hash table. It purely uses one-sided verbs and leverages `RDMA_CAS` for the lock-free remote concurrency control. The concurrent queue follows the design in [17]; it is implemented as a lock-free linked list of segments, with each segment containing multiple entries.

Necessity for protection. Inserting (removing) elements to (from) the data structure involves memory allocations. Since the remote data structure is shared by multiple clients, the race of memory reallocation, especially invoked from other clients, turns any concurrent one-sided requests into stray access. Specifically, RACE hashing uses copy-on-write (CoW): updating a new value involves freeing the old KV block $\langle K, V_{old} \rangle$. A seriously *delayed* client, e.g., due to network traffic or scheduling, may post one-sided access to V_{old}, the already unowned memory. Similar situations apply to any one-sided data structures involving memory management. Note that this race is hard to address from the design of data structures because the delay is nondeterministic.

Necessity for performance. The one-sided data structures are the essential building block of applications in remote memory; their efficiency determines the performance of the system. The data structures typically support millions of operations per second with microsecond-scaled latency, asking for high-performance protection at the same level.

Adoption of PATRONUS. For RACE hashing, we take insertion as a concrete example. In the vanilla implementation, insertion takes four steps in the common path. (*i*) Allocate a KV block and write the KV to the block. (*ii*) Read the bucket in the subtable. (*iii*) Link the KV block into the bucket via CAS. (*iv*) Re-read the bucket to detect duplicity. To adopt PATRONUS, we use our `allocate` API for KV block allocation (and the permission) and use one `acquire` for the permission to access the subtable. Among the four RDMA operations (one write, two reads, one CAS), two PATRONUS operations are introduced (one `allocate` and one `acquire`). Note that the subtables, as the metadata, are (re)-accessed frequently; therefore, the active permission to the subtable can be re-used several times, possibly across insertions.

For concurrent queue, it is implemented as a lock-free linked list of segments, with each segment containing multiple entries. At insertion, the client tries to fetch an index of an available entry slot from the segment via `FAA`, and fill the entry slot via `write`. If failed (i.e., the segment is full), the client allocates a new segment and links it to the back of the list via `CAS`. The concurrent queue also contains a metadata block maintaining the (possibly stale) head and tail of the linked list. With PATRONUS, each new segment introduces one `allocate` for allocation and one `acquire` for the access permission to the segment. Each client also maintains a prolonged permission to the metadata block.

6.2 Function as a Service

The FaaS is a cloud computing paradigm where the applications are developed and served at the unit of functions. Each function runs in a virtualized environment for isolation and performance fairness. We consider that the FaaS platform equips RM as an external medium for data storage.

Necessary for protection. In the FaaS system, functions submitted by different users access shared remote memory

CPU	Xeon Gold 6240M @2.6 GHz, 32 logical cores, with hyperthreading enabled
RAM	186 GB 2666 MHz DDR4
NIC	Mellanox MT27800 ConnectX-5 Family
OS	18.04.5 LTS, Linux 4.15.0-153

Table 3: Experimental cluster configuration. The evaluation was carried out on a 4-node cluster.

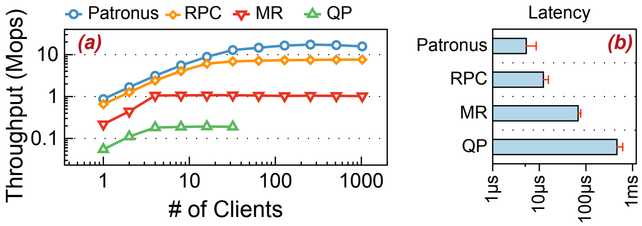


Figure 5: The throughput *(a)* and latency *(b)* of random access of RM with PATRONUS and other protection techniques. QP does not scale beyond 32 clients per machine.

simultaneously. It asks for the isolation of remote access in addition to the existing isolation of local memory and storage. With PATRONUS, future FaaS systems can offer sandboxed remote memory to functions with flexible control over which function is allowed to access which piece of remote memory.

Necessity for performance. First, functions in FaaS have a low bootstrap latency. The state-of-the-art FaaS systems [9,14,27,33] enable microsecond-scaled ultra-low bootstrap latency in the case of hot starts, emphasizing the need for low-latency permission management. Second, functions scale out quickly to millions of instances [33]. Even if each function accesses remote memory once, it introduces a flood of permission acquisitions, asking for high throughput permission management to meet performance needs. Finally, functions access remote memory on demand. Functions are spawned dynamically in response to user requests, and remote access from functions can not be known in advance. It involves permission management in the critical path, precluding the optimization of pre-acquiring permission.

Adoption of PATRONUS. Functions access remote memory on bootstrap, do some calculations, and exit. With PATRONUS, for each data on RM, one `acquire` call is introduced on function bootstrap and one `revoke` is introduced on exit. Re-access to the same data shares the same permission from one `acquire`. Specifically, for the image processing and data analysis workloads we evaluated in the experiment, functions call `acquire` for permission to access the input image and in-memory database respectively. We assume cascadingly invoked functions are executed in the same container so that they can share permission (communicate) with local memory (a technique called *sequence function chain* [8, 16]).

7 Evaluation

Compared mechanisms. We adopt **PATRONUS** to enable protection in various workloads and compare it against three

Name	(Abbr)	# of MW	# of RPC
Baseline		2 + 2	2
Delay Unbind	(+ Delay)	2 + **1**	2
Use Contiguity	(+ Cont)	**1** + 1	2
MW Handover	(+ HO)	1 + **0** †	2
Lease Expire	(+ Expr)	1 + 0 †	**1**

Table 4: A summary of techniques for reducing the permission management overhead (§5.4). The **# of MW** column reports *binding* + *unbinding* operations. † means at probability.

mechanisms used in existing RM systems. (*i*) Re-registration of memory region (**MR**), representing the mechanism adopted by FaRM [12] and Octopus [34]. (*ii*) Modification of QP flag (**QP**), used by uPaxos [3]. (*iii*) Using **RPC** in the data path (no permission acquisitions needed), used by AIFM [43] and Redy [56]. Finally, **Unprot** stands for the vanilla implementation of workloads without any protection.

Experimental setup. We perform the evaluations on a cluster with 4 nodes. Table 3 summarizes the configuration. One node acts as the memory node with limited use of 4 CPU cores. The others are compute nodes with 32 cores. We bind each client thread to a core; for more than 32 clients, we spawn coroutines in each thread to simulate a larger deployment. On reporting latency, we disable coroutines to avoid the schedule variance. The number of clients reported is per machine.

7.1 Overall Performance

Experimental setting. We performed an experiment to reveal the overall data path performance of PATRONUS and compared mechanisms. In the experiment, each client randomly accesses 64 B within a large memory region. The client will re-access the same address three times while using the same permission, representing the common use cases with space locality [12, 52]. The effective access throughput and latency are reported in Figure 5.

Result. Among these techniques, PATRONUS performs the best and only `RPC` can keep pace with it. The performance gap will be enlarged significantly for a larger IO size because `RPC` pays extra overhead of memory copy for each access, but the MW overhead that PATRONUS pays is irrelevant to the size. The performances of `MR` and `QP` are not comparable to PATRONUS. The MR registration is expensive, because it is synchronous and incurs kernel involvement (§2.1). The latter requires modification to the QP flag, which includes the complex QP state management overhead in the RNIC. The QP-based solution also precludes sharing QPs among clients; therefore, we can not evaluate it with more clients.

7.2 Effect of Software Co-design

In this section, we evaluate the effect of the software co-design that makes PATRONUS achieve all the goals.

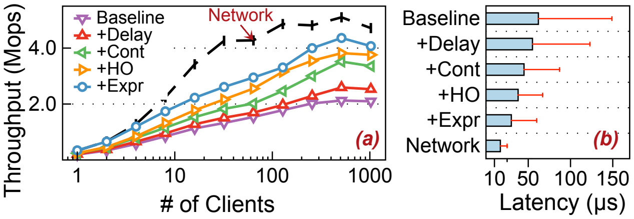


Figure 6: Throughput *(a)* and latency *(b)* of permission acquisition under different optimizations.

7.2.1 Performance of Permission Management (G#1)

Experimental setting. We evaluate the performance of permission management by breaking down the techniques we adopt. Table 4 summarizes the technique. Besides the three techniques described in §5.4, we additionally consider the lease semantics as the final technique, which effectively eliminates the overhead of `revoke` RPC.

To evaluate the performance, we saturate the system with enough clients to acquire (and revoke) permission over 64 B areas with random addresses. We report the throughput and latency of permission acquisitions in Figure 6.

Result. The combination of all the techniques effectively leads to a performance close to the network bound (bare RPC performance). The eventual throughput is more than 1 Mops per core, which is only achievable with our effort in software co-design, considering that additional overhead besides MWs also exists in the system, such as memory management, RPC, and lease management. In theory, we reduce the overhead of managing a full permission lifecycle to one MW operation and one RPC (the last line in Table 4), which doubles the performance as the baseline and, we believe, exploits the true potential of the hardware.

7.2.2 CN-collaborated Extension (G#2)

In this section, we demonstrate the necessity of the extension API and the effectiveness of our CN-collaborated extension technique (§5.3).

Experimental setting. We evaluate three cases: no extension API, the naive RPC-based extension, and our CN-collaborated extension technique. Without extensions, the unexpected permission expiration requires another `acquire` call to get the permission again (denoted as `Re-acquire`). The RPC-based implementation allows extension but in a naive way, i.e., uses an RPC to notify the memory nodes (`+ Extend`). Finally, our technique (`+ CN Extend`) offloads the management overhead to the CN. In the evaluation permission extends eight times.

Result. Figure 7 reports the throughput and latency. Without the extension, the `Re-acquire` brings both extra MW operations and RPC overhead, which bottlenecks the system seriously and gives only 202 Kops. The RPC-based extension implementation, although saves unnecessary MW operations, still introduces the RPC overhead to bottleneck the system. The CN-collaborated technique reduces both the MW and RPC overhead, effectively producing a ×6 performance gain.

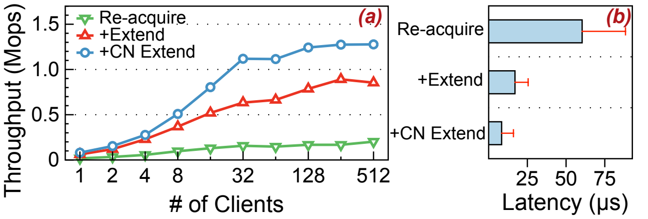


Figure 7: Comparison of not allowing extension (Re-acquire), using RPC for extension (Extend), and our CN-collaborated extension technique (CN Extend). Reporting throughput *(a)* and latency *(b)*.

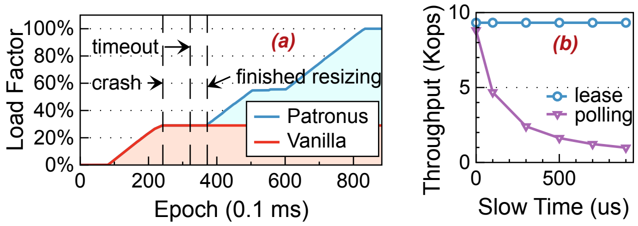


Figure 8: *(a)* The load factor of RACE hashing under client crash for vanilla implementation and PATRONUS. *(b)* The comparison of polling and leases with clients of different fail-slow degrees.

7.2.3 Effect of Lease Semantics (G#2)

We evaluate how the lease semantics enables the system to resume progress when the client crashes while holding exclusive permission. We use resizing in RACE hashing [59] as a case.

Experimental setting. In the experiment, clients are concurrently accessing the hash table while resizing occurs. RACE hashing does not allow cascaded resizing, so the resizing client accesses the metadata (i.e., the resizing subtable) in an exclusive way. The resizing client crashes at epoch 240, leaving the orphaned exclusive permission (or an orphaned lock in the vanilla design) in the system, potentially causing deadlocks. We compare PATRONUS against the vanilla design.

Result. Figure 8 *(a)* shows the load factor of the hash table while clients are concurrently loading data into the table and the resizing client crashes. In the vanilla design, the orphaned lock results in deadlock and prevents the following insertions into the table (red line in the figure). With PATRONUS, the exclusive permission to the metadata can be re-granted to the other concurrent clients after the permission expires. The other clients resume progress and load the table full.

7.2.4 Compare QP polling to leases (G#2)

Experimental setting. QP polling (`polling`) is an alternative approach to leases (`lease`) where memory nodes periodically issue RDMA operations to each QP as heartbeats. On heartbeat timeout, which we set to the same value of lease time (100 μs), the memory node suspects that the compute node has crashed and reclaims the permission. We report the throughput of exclusive permission acquisitions under the anomalies where clients fail slow (get hanged due to network traffic or

Name	w/o	w/
Failure Reported	769 µs	
Promote QP	-	78 µs
Notify QP Failure	8 µs	-
Recover QP	1004 µs	-
Summary	1012 µs	78 µs (8 %)

Table 5: Latency breakdown of handling QP faults with (w/) or without (w/o) the spare QPs technique.

scheduling); a zero slow time denotes the case of normal clients.

Result. Figure 8 *(b)* shows the throughput with normal and slow clients. With normal clients (*zero* slow time), `lease` performs slightly better than `polling` (6 % better) because it allows the lease to expire itself, saving one `revoke` call than `polling`. With slow clients, `lease` is able to retain performance by timely expiration while `polling` does not detect it and degrades performance seriously.

7.2.5 Spare QPs for Concealing Interruption (G#3)

In this section, we evaluate how the spare QPs can conceal the interruption caused by QP faults with illegal access. We prepare spare QPs and trigger illegal access (an out-of-bound write) deliberately.

Result. We break down the latency with and without the technique in Table 5. After illegal access triggers QP faults, the case without spare QPs needs to go through the QP rebootstrap procedure, introducing significant overhead (Recover QP, 1004 µs). Since QP recovery needs effort from both sides, the client needs to notify the manager (Notify QP Failure, 8 µs). On the other hand, for the case with spare QPs, only the promotion of spare QPs is required (Promote QPs, 78 µs), which involves a handy resource swap in the software. Therefore, it reduces the interrupted time to 8 %.

The illegal request takes much longer for the RNIC to complete (Failure Reported, 769 µs), measured between posting the request and the error being notified. Unfortunately, this procedure purely happens in the RNIC firmware and is confidential; we are unable to analyze and optimize it. Nevertheless, we expect that this period can be significantly shortened by simple modifications to the firmware for future RNICs.

7.3 Case Study: One-sided Data Structures

Next, we reveal the performance of PATRONUS under realistic workloads. In this section, we focus on two remote one-sided data structures, i.e., RACE hashing [59] and the concurrent queue (adopted from [17, 24]). We omit the QP-based mechanism in the figures because it has much worse performance and does not allow scaling beyond 32 clients per CN (not allow clients to share QP).

7.3.1 Hash Table

Experimental setting. We adopt PATRONUS to RACE hashing [59], the state-of-the-art one-sided extendable hash table. Since RACE hashing is not open-sourced, we implement RACE hashing following the original paper, with all the optimizations described in the paper enabled. We verified that the performance of our version is on par with the one reported in the paper. In the evaluation, we set the size of KV blocks to 4 KB. The key follows Zipfian distribution with skewness parameter 0.99. We also consider memory allocation in the critical path as the extended version of the paper does [59], with a pre-allocation factor of four to amortize allocation overhead. The lease time is set to 100 µs. The detailed adoption of PATRONUS is described in §6.1.

Result. Figure 9 shows the throughput under read-only, 50% read-write and write-only workloads (*a*-*c*) and the read-only latency (*d*) respectively. PATRONUS performs the best and has a reasonable price for memory protection. The protection only introduces 4 µs to the median latency (+29 %), which is an acceptable price for most use cases. The MR-based mechanism is not scalable under all workloads. It is because insert and query to RACE hashing involve a lot of memory access, generating a flood of permission requests in turn. The MR-based mechanism is unable to meet the performance requirements. The P99 latency of MR degrades severely because of the synchronous API it exposes. The RPC mechanism gets better, but it is still inferior due to the memory copy overhead.

7.3.2 Concurrent Queue

Experimental setting. The concurrent queue is implemented as a lock-free linked list of segments; each segment contains 1024 entries. The lease time is set to 100 µs. The implementation and adoption of PATRONUS are described in §6.1.

Result. Figure 10 reports the throughput and latency of inserts with the variety of producers. The performance of PATRONUS is very close to the theoretical upper bound without protection. The reason for the high efficiency is that the overhead of one PATRONUS operation can be amortized into multiple insert operations. Nevertheless, the MR-based solution is still insufficient in terms of throughput because its overhead is too high to amortize (17.5 % throughput as PATRONUS). The RPC-based solution also gets inferior performance because it uses two-sided verbs in its data path, introducing overhead to the limited CPU cores on memory nodes. We also vary segment size from 64 to 1024 to reveal the effect; PATRONUS is ×5.18 to ×1.78 better than MR (not shown for space limits).

7.4 Case Study: Function as a Service

Description. To evaluate how PATRONUS performs with the FaaS platform, we adopt ServerlessBench [54], a thorough benchmark with representative realistic serverless workloads. We consider two typical applications in TC4 of ServerlessBench, i.e., image processing and data analysis. The former is one of the most popular workloads in the cloud [5], which comprises five functions in the chain to extract metadata and generate the thumbnail of the input image. The data analysis application is a workflow that analyses the salary of employees, triggered by data alteration in the database.

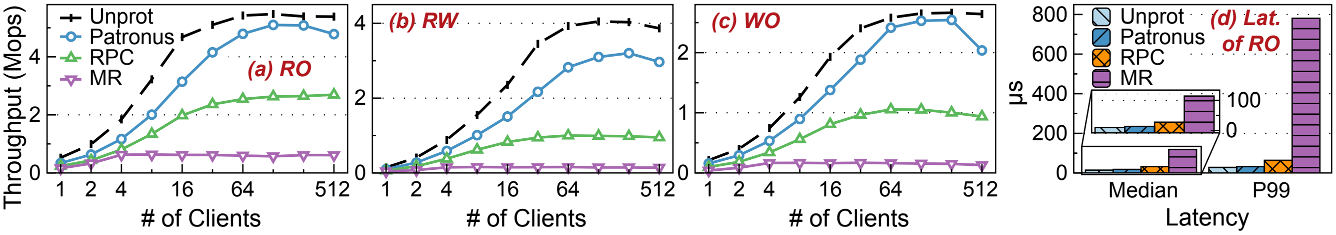


Figure 9: Performance of RACE hashing. *(a-c)* The throughput under read-only (RO), 50%-mixed read-write (RW), and write-only (WO) workloads. *(d)* The latency under RO workload.

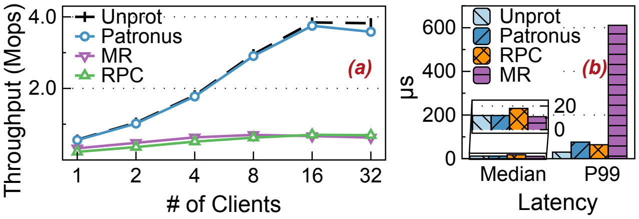


Figure 10: Throughput of producers for the one-sided concurrent queue.

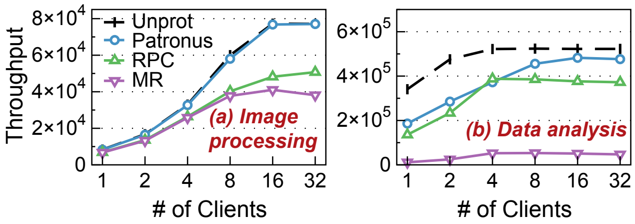


Figure 11: Performance of two serverless applications.

Experimental setting. In the evaluation, we launch enough functions to saturate the system. We assume hot starts for all the functions, excluding the overhead of disk IO and container bootstrap. The lease time is set to multiple times of function lifetime so that lease extensions are rare. The adoption of PATRONUS is described in §6.2.

Result. For the image processing application (Figure 11 *(a)*), PATRONUS has a performance close to the unprotected case. The reason is that generating the thumbnail is a CPU-intensive task, and thus the bottleneck shifts from the protection overhead to the CPU computation. Besides, MW has a constant overhead over the memory size; therefore, the protection performance remains constant even with larger images. On the contrary, the overhead of MR and RPC is so high that they still bottleneck the system even in this CPU-intensive case.

For the data analysis workload (Figure 11 *(b)*), it is more IO-intensive than the previous workload and therefore a wider performance gap is shown between PATRONUS and the unprotected case. Nevertheless, PATRONUS still performs the best and the gap is shortened with more concurrent clients (≥ 8) in the system.

8 Related Work

The development of fast networks, especially the emergence of RDMA, leads to a wide discussion on resource disaggregation [18, 21, 23, 44, 45]. Among them, remote memory is one of the most typical forms of disaggregation, and it has gained much research interest in the last decade [2, 15, 31, 32]. To our best knowledge, no prior RM system has provided efficient protective interfaces with commodity RNICs.

Performance-oriented RM. A wide range of research on RM focuses on optimizing performance with customized hardware. Kona [11] eliminates the virtual memory overhead with a new architecture. Other work [4, 7, 10] extends the RDMA interface for richer semantics. StRoM [47] adopts the idea of near-data processing by performing task offload to the smart remote memory. On the contrary, PATRONUS focuses on the less-discussed access protection issue, which is neglected by these systems. PATRONUS runs on unmodified commodity hardware, allowing a lower cost and a wider deployment.

Protective interfaces. Some RM systems provide access protections with customized hardware, such as programmable switches [29], FPGA [22], and architectural modifications [28]. PATRONUS is designed for commodity hardware, serving as a ready-to-use solution for existing data centers. On the other hand, transactional RM systems [12, 52, 55] also provide protection for data consistency with the transaction interface. However, the transaction semantics is overkilled for most cases (e.g., data structures) because it introduces the expensive overhead of transaction logging and distributed commit protocol.

9 Conclusion

In this paper, we designed, implemented, and evaluated PATRONUS, a protective remote memory system. PATRONUS achieves high performance under all situations by hardware and software co-design. Deployed to realistic applications, it performs ×5.2 better than all the competitors and introduces acceptable overhead ($\leq 27.7\,\%$).

Acknowledgments

We sincerely thank our shepherd Hyungon Moon and the anonymous reviewers for their valuable feedback. This work is funded by the National Natural Science Foundation of China (Grant No. 61832011, 62022051) and the National Key R&D Program of China (Grant No. 2021YFB0300500).

References

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, Boston, MA, July 2018. USENIX Association.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 121–127, New York, NY, USA, 2017. Association for Computing Machinery.

[3] 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.

[4] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 120–126, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[6] 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*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Emmanuel Amaro, Zhihong Luo, Amy Ousterhout, Arvind Krishnamurthy, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Remote memory calls. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 38–44, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Apache OpenWhisk. `http://openwhisk.apache.org/`. Accessed: 2022-09-01.

[9] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, July 2018. USENIX Association.

[10] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. Prism: Rethinking the rdma interface for distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 228–242, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 79–92, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[13] 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, page 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[16] Fn Protect. `https://fnproject.io`. Accessed: 2022-09-01.

[17] Folly UnboundedQueue. https://github.com/facebook/folly/blob/main/folly/concurrency/UnboundedQueue.h. Accessed: 2022-09-01.

[18] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 249–264, Savannah, GA, November 2016. USENIX Association.

[19] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, Carlsbad, CA, July 2022. USENIX Association.

[20] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Zhiyuan Guo, Zachary Blanco, Mohammad Shahrad, Zerui Wei, Bili Dong, Jinmou Li, Ishaan Pota, Harry Xu, and Yiying Zhang. Resource-centric serverless computing. *arXiv preprint arXiv:2206.13444*, 2022.

[22] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 417–433, New York, NY, USA, 2022. Association for Computing Machinery.

[23] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, page 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. *SIGCOMM Comput. Commun. Rev.*, 44(4):295–306, aug 2014.

[26] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, and Ion Stoica. Jiffy: Elastic far-memory for stateful serverless analytics. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 697–713, New York, NY, USA, 2022. Association for Computing Machinery.

[27] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[28] Vamsee Reddy Kommareddy, Clayton Hughes, Simon David Hammond, and Amro Awad. Deact: Architecture-aware virtual memory support for fabric attached memory systems. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 453–466. IEEE, 2021.

[29] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. Mind: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 488–504, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. Hydra : Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 181–198, Santa Clara, CA, February 2022. USENIX Association.

[31] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *SIGARCH Comput. Archit. News*, 37(3):267–278, jun 2009.

[32] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, 2012.

[33] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient microservices on SmartNIC-Accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, Renton, WA, July 2019. USENIX Association.

[34] 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.

[35] Teng Ma, Dongbiao He, and Ning Liu. Hybridskiplist: A case study of designing distributed data structure with hybrid rdma. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 68–73, 2021.

[36] 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.

[37] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a feather flock together: Scaling rdma rpcs with flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 212–227, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, July 2015. USENIX Association.

[39] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, July 2015. USENIX Association.

[40] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 97–108, New York, NY, USA, 2019. Association for Computing Machinery.

[41] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, jan 2010.

[42] RDMA Memory Window. `https://docs.nvidia.com/networking/pages/viewpage.action?pageId=25138102`. Accessed: 2022-09-01.

[43] 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. USENIX Association, November 2020.

[44] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, October 2018. USENIX Association.

[45] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. Towards a fully disaggregated and programmable data center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '22, page 18–28, New York, NY, USA, 2022. Association for Computing Machinery.

[46] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 323–337, New York, NY, USA, 2017. Association for Computing Machinery.

[47] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: Smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48. USENIX Association, July 2020.

[49] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 306–324, New York, NY, USA, 2017. Association for Computing Machinery.

[50] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1033–1048, New York, NY, USA, 2022. Association for Computing Machinery.

[51] Tinggang Wang, Shuo Yang, Hideaki Kimura, Garret Swart, and Spyros Blanas. Efficient usage of one-sided rdma for linear probing. In *Eleventh International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (AMDS'20)*, 2020.

[52] 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.

[53] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. KRCORE: A microsecond-scale RDMA control plane for elastic computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 121–136, Carlsbad, CA, July 2022. USENIX Association.

[54] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[55] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 51–68, Santa Clara, CA, February 2022. USENIX Association.

[56] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. Redy: Remote dynamic memory cache. *Proc. VLDB Endow.*, 15(4):766–779, dec 2021.

[57] 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 *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

[58] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 741–758, New York, NY, USA, 2019. Association for Computing Machinery.

[59] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-Conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 15–29. USENIX Association, July 2021.

More Than Capacity: Performance-Oriented Evolution of Pangu in Alibaba

Qiang Li[◇], *Qiao Xiang*[†], *Yuxin Wang*[†◇], *Haohao Song*[†◇], *Ridi Wen*[†◇],
Wenhui Yao[◇], *Yuanyuan Dong*[◇], *Shuqi Zhao*[◇], *Shuo Huang*[◇], *Zhaosheng Zhu*[◇],
Huayong Wang[◇], *Shanyang Liu*[◇], *Lulu Chen*[◇], *Zhiwu Wu*[◇], *Haonan Qiu*[◇], *Derui Liu*[◇],
Gexiao Tian[◇], *Chao Han*[◇], *Shaozong Liu*[◇], *Yaohui Wu*[◇], *Zicheng Luo*[◇],
Yuchao Shao[◇], *Junping Wu*[◇], *Zheng Cao*[◇], *Zhongjie Wu*[◇], *Jiaji Zhu*[◇], *Jinbo Wu*[◇],
Jiwu Shu[†], *Jiesheng Wu*[◇],
[◇]*Alibaba Group*, [†]*Xiamen University*

Abstract

This paper presents how the Pangu storage system continuously evolves with hardware technologies and the business model to provide high-performance, reliable storage services with a 100-μs level of I/O latency. Pangu's evolution includes two phases. In the first phase, Pangu embraced the emergence of solid-state drive (SSD) storage and remote direct memory access (RDMA) network technologies by innovating its file system and designing a user-space storage operating system. As a result, Pangu substantially reduced its I/O latency while providing high throughput and IOPS. In the second phase, Pangu evolved from a volume-oriented storage provider to a performance-oriented one. To adapt to this business model change, Pangu upgraded its infrastructure with storage servers of much higher SSD volume and RDMA bandwidth from 25 Gbps to 100 Gbps. It introduced a series of key designs, including traffic amplification reduction, remote direct cache access, and CPU computation offloading, to ensure Pangu fully harvests the performance improvement brought by hardware upgrades. Other than technology innovations, we also shared our operating experiences during Pangu's evolution, and discussed important lessons learned from them.

1 Introduction

Since Alibaba started developing and deploying the Pangu storage system in 2009, Pangu has been serving as a unified storage platform for Alibaba Group and Alibaba Cloud. It provides a scalable, high-performance, and reliable storage service for Alibaba's core businesses (*e.g.*, Taobao, Tmall, AntFin, and Alimama). Many cloud services, such as Elastic Block Storage (EBS) [1], Object Storage Service (OSS) [2], Network-Attached Storage (NAS) [3], PolarDB [4], and MaxCompute [5], are built on top of Pangu. After over a decade, Pangu has become a global storage system with a volume of exabytes and manages trillions of files.

Pangu 1.0: volume-oriented storage service provision. The development and deployment of Pangu go through two generations. Pangu 1.0 spanned from 2009 to 2015. It was designed on an infrastructure composed of servers with commodity CPUs and hard disk drives (HDD), which have a *ms*-level I/O latency, and a Gbps-level datacenter network. Pangu 1.0 designed a distributed kernel-space file system based on Linux Ext4 [6] and kernel-space TCP [7], and gradually added support to multiple file types (*e.g.*, TempFile, LogFile, and random access file) as needed by different storage services. This period overlaps with the early stage of cloud computing. Although Pangu 1.0's performance (*i.e.*, throughput and I/O latency) reached the limit of HDD and Gbps-level networks, clients' primary focus was to get large volumes of space to store their data, rather than high performance.

New hardware technologies require new designs. Since 2015, we started to design and develop Pangu 2.0 to embrace the emerging SSD and RDMA technologies. The goal of Pangu 2.0 is to *provide high-performance storage services with a 100μs-level I/O latency*. Although SSD and RDMA can achieve high-performance, low-latency I/O in storage and network, we observe that (1) multiple file types used in Pangu 1.0, in particular file types that allow random access, perform poorly on SSD, which can achieve high throughput and IOPS on sequential operations; (2) the kernel-space software stack cannot keep up with the high IOPS and low I/O latency of SSD and RDMA, due to data copy and frequent interrupts; and (3) the paradigm shift from server-centric datacenter architectures to resource-disaggregated datacenter architectures poses additional challenges to achieving low I/O latency.

Phase one of Pangu 2.0: embracing SSD and RDMA by file system refactoring and user-space storage operating system. To achieve high-performance and low-latency I/O, in this phase, Pangu 2.0 first proposed new designs in key components of its file system. To simplify the development and management of the overall system, it designed a unified, append-only persistence layer. It also introduced a self-contained chunk layout to reduce the I/O latency of file write operations. Second, Pangu 2.0 designed a user-space storage operating system (USSOS). USSOS uses a run-to-completion thread model to realize efficient collaboration between the user-space storage stack and the user-space network stack. It also proposes a user-space scheduling mechanism for efficient CPU and memory resource allocation. Third, Pangu 2.0 deployed mechanisms to provide SLA guarantees under dynamic environments. With these innovations, Pangu 2.0 achieved a ms-level P999 I/O latency in phase one.

Phase two of Pangu 2.0: adapting to a performance-oriented business model with infrastructure updates and breaking through network/memory/CPU bottlenecks. Since 2018, Pangu gradually changed its business model from volume-oriented to performance-oriented. It is because enterprises are increasingly moving their businesses to Alibaba Cloud and they have stringent requirements on storage performance and latency. This shift became faster after the COVID-19 pandemic broke out in 2020. To adapt to this business model change and the fast expansion of clientele, Pangu 2.0 needed to keep upgrading the infrastructure.

Scaling the infrastructure with original servers and switches along a Clos-based topology (*e.g.*, FatTree [8]) is not economical, including a higher total cost of ownership (*e.g.*, rack space, power, cooling, and labor) and a higher environmental cost (*e.g.*, a higher carbon emission rate). As such, Pangu developed in-house high-volume storage servers (96 TB SSD per server) and upgrades network bandwidth from 25 Gbps to 100 Gbps.

To fully harvest the performance improvement brought by these upgrades, Pangu 2.0 proposed a series of techniques to cope with the performance bottleneck at network, memory, and CPU and fully utilize its massive resources. Specifically, Pangu 2.0 optimized network bandwidth by reducing the network traffic amplification ratio and dynamically adjusting the priorities of different traffic. It coped with the memory bottleneck by proposing remote direct cache access (RDCA). It addressed the CPU bottleneck by eliminating the data tax of data (de)serialization and introducing *CPU wait* instruction to synchronize hyper-threading.

High performance in production. By the end of phase one, Pangu 2.0 successfully supported the elastic SSD block storage service with a 100μs-level I/O latency and 1M IOPS. During Alibaba's Double 11 Festival in 2018, Pangu 2.0 supported the Alibaba database service with a latency of 280 μs. For the OTS storage service [9], with the same hardware, its I/O latency in Pangu 2.0 is lower than that in Pangu 1.0 by an order of magnitude. For write-intensive services (*e.g.*, EBS cloud drive), the P999 I/O latency is less than 1 ms. For read-intensive services (*e.g.*, online search), the P999 I/O latency is less than 11 ms.

In phase two, by upgrading the network from 2×25 Gbps to 2×100 Gbps and breaking through the bottlenecks of network, memory and CPU, the normalized effective throughput per Taishan storage server increases by 6.1×.

2 Background

2.1 Overview of Pangu

Pangu is a large-scale distributed storage system. It consists of Pangu Core, Pangu Service, and Pangu Monitoring System (Figure 1). Pangu Core consists of clients, masters, and chunkservers, and provides an append-only persistence semantic. The clients provide SDK to Pangu cloud storage services (*e.g.*, EBS and OSS), and are responsible for receiving file requests from these services and communicating with the masters and the chunkservers to fulfill these requests. Similar to other distributed file systems (*e.g.*, Tectonic [10] and Colossus [11]), the clients in Pangu are heavyweight and play a key role in the replica management, SLA guarantee, and data consistency management of Pangu.

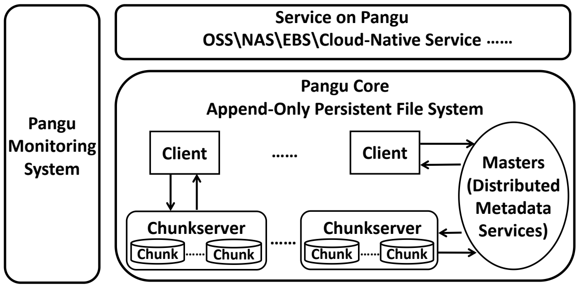


Figure 1: The architecture of Pangu.

The masters manage all the metadata in Pangu and use a Raft-based protocol to maintain metadata consistency [12]. For better horizontal scalability and extensibility (*e.g.*, hundreds of billions of files), the Pangu masters decompose the monolithic metadata service into two separate services: the namespace service and the stream meta service, where a stream is an abstraction of a group of chunks. Both services first partition metadata by directory tree to achieve metadata locality, then further partition these groups using hashing to achieve good load balancing [13]. While the namespace service provides information on files (*e.g.*, directory tree and the namespace), the stream meta service provides the mapping from files to chunks (*i.e.*, the locations of chunks). Chunkservers store data in chunks and are equipped with a customized user-space storage file system (USSFS). The USSFS provides high-performance, append-only storage engines for different hardware (*e.g.*, SMRSTORE for HM-SMR drives [14]). In the early days (*i.e.*, phase one of Pangu 2.0), each file is stored in chunkservers with three replicas, and later a garbage collection worker (GCWorker) performs garbage collection and stores the file using erasure coding (EC). In phase two of Pangu 2.0, we gradually replace the 3-way replication with EC in key businesses (*e.g.*, EBS) to reduce the traffic amplification in Pangu (§4.1.2).

On top of the Pangu Core, the Pangu Service provides traditional cloud storage services (*e.g.*, EBS, OSS, and NAS) and cloud-native storage services through a cloud-native-oriented file system (*i.e.*, Fisc [15]). The Pangu Monitoring System (*e.g.*, Perseus [16]) provides real-time monitoring and AI-assisted root cause analysis services to both the Pangu Core and the Pangu Service. The infrastructure of Pangu Core, the Pangu Service, and the Pangu Monitoring System are interconnected using high-speed networks [17, 18].

2.2 Design Goals of Pangu 2.0

To cope with the emergence of hardware technologies and the shift of business model, Pangu 2.0 aims to achieve the following goals:

- **Low latency**: Pangu 2.0 aims to leverage the low latency characteristics of SSD and RDMA, to reach an average 100μs-level I/O latency in a computation-storage disaggregated architecture, and provide a ms-level P999 SLA even under environment dynamics such as network traffic jitters and server failures.
- **High throughput**: Pangu 2.0 aims to reach an effective throughput on storage servers that approaches their capacity.
- **Unified high-performance support to all services**: Pangu 2.0 aims to provide unified high-performance support to all services running on top of it, such as online search, data streaming analytics, EBS, OSS, and database.

2.3 Related Work

Many distributed storage systems have been designed and deployed [10, 19–21]. Some are open-sourced ones (*e.g.*, HDFS [20] and Ceph [22]), and some are proprietary ones used by different industry organizations (*e.g.*, GFS [19], Tectonic [10], and AWS [23]).

Pangu is a proprietary storage system of Alibaba Group. It provides storage infrastructure support for Alibaba's core businesses and Alibaba Cloud. In the past few years, we have shared our experiences in different aspects of Pangu, such as the large-scale deployment of RDMA [17], the key-value engine for scale-out cloud storage services [24], the co-design of network and storage software stack for the EBS storage service [18], and some key designs of the namespace metadata service [13]. This paper focuses on introducing our experience in evolving Pangu to provide unified low-latency, high-throughput storage services to support all Alibaba's businesses and Alibaba Cloud, in response to the emergence of hardware technologies and the shift of Pangu's business model.

3 Phase One: Embracing SSD and RDMA

In this section, we introduce how Pangu embraces the emergence of SSD and RDMA to provide high-performance, reliable storage services with low I/O latency. Compared with HDD and TCP, SSD and RDMA technologies substantially reduce the I/O latency in storage and network, respectively. However, integrating these two technologies into Pangu, a large, distributed storage system with a mature architecture, is not without challenges. To this end, Pangu introduces a series of new designs in key components of its file system (§3.1) and develops a user-space storage operating system (§3.2) to achieve a high-throughput, high IOPS performance with a 100μs-level I/O latency. It also deploys novel mechanisms to provide such SLA guarantees under dynamic environments, *e.g.*, straggler and transient/permanent failures (§3.3).

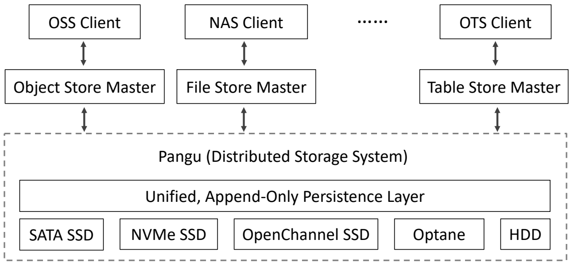


Figure 2: Various businesses are based on the unified persistence layer.

3.1 Append-Only File System

As shown in Figure 1, the core base layer of Pangu consists of the masters, chunkservers, and clients. Pangu first introduces a unified, append-only persistence layer with an append-only interface called FlatLogFile to simplify its architecture (§3.1.1). FlatLogFile is friendly to SSDs with high throughput and low latency. Then Pangu adopts a variety of designs to improve its performance. Specifically, the Pangu client is heavyweight to meet the requirements of different storage services (§3.1.2). Based on FlatLogFile, Pangu adopts the append-only chunks and uses a self-contained chunk layout to manage chunks on chunkservers (§3.1.3). Pangu implements distributed metadata management on the master to realize efficient metadata operation (§3.1.4).

3.1.1 Unified, Append-Only Persistence Layer

The persistence layer of Pangu provides interfaces to all Pangu's storage services (*e.g.*, EBS, OSS, and NAS). In the early development of Pangu, the persistence layer provides different interfaces to different storage services. For example, it provides the LogFile interface to low-latency NAS services, and the TempFile interface to high-throughput Maxcompute data analytics services. However, this design brings substantial development and management complexities. Specifically, for every storage service, Pangu developers must design and implement a new interface. That is a complex, labor-intensive and error-prone process.

To simplify the development and management of Pangu and make sure all storage services can achieve high-performance, low-latency I/O on SSD, motivated by the layered architecture of computer networks, Pangu introduces a unified file type called FlatLogFile (Figure 2). Specifically, FlatLogFile has an append-only semantic, and upper-layer services (*e.g.*, OSS) can equip a key-value-like mapping to update their data and a garbage collection mechanism to compress their historical data. FlatLogFile provides a simple, unified interface for storage services to perform data operations. Furthermore, Pangu developers must ensure that data operations via FlatLogFile, especially the write operations, can be executed efficiently and reliably on storage media. As such, all upgrades and changes to storage services are transparent to Pangu developers, substantially simplifying the development and management of Pangu.

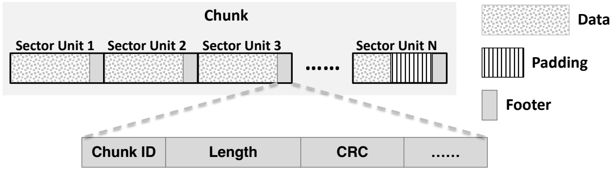


Figure 3: The self-contained chunk layout.

Under the hood, we observe that SSD can achieve high throughput and IOPS on sequential operations due to its intrinsic characteristics of the storage unit and flash transaction layer. To make sure data operations via FlatLogFile can be executed on SSD efficiently, we align the sequential operations on FlatLogFile to achieve high performance.

3.1.2 Heavyweight Client

We design Pangu's client as a heavyweight one. It is responsible for the data operations with chunkservers and the metadata information retrieval and updates with the masters. After getting the chunk information from the masters, a Pangu client is in charge of the corresponding replication protocol and EC protocol. The client is equipped with retry mechanisms (*e.g.*, backup read in §3.3) to cope with occasional jitters in Pangu (*e.g.*, network packet drop) in order to improve the I/O SLA. It also deploys probing mechanisms to periodically get the latest chunkserver status from the masters and evaluate the quality of services of chunkservers. Similar to the client of Facebook's Tectonic file system [10], the Pangu client can select appropriate write or read parameters to meet the specific requirements of different storage services (*e.g.*, EBS and OSS).

3.1.3 Append-only Chunk Management

Typical file systems (such as Ext4 [6]) store files in blocks. A file and its metadata are written to the storage media separately with two SSD write operations. Not only does it increase the latency of the file write, it also shortens the lifespan of SSDs. As such, adopting this design in Pangu 2.0 would result in high latency for file access, and lead to a high hardware replacement cost.

To address this issue, Pangu chooses to store files in chunks, which have the append-only semantic based on FlatLogFile, on chunkservers, and designs a self-contained chunk layout, where each chunk stores both data and its own metadata. As such, a chunk can be written into the storage media in one operation instead of two, substantially reducing the write latency and improving the lifespan of the storage media. Figure 3 shows the layout of the chunk. A chunk includes multiple sector units, where each sector unit includes 3 elements: data, padding, and footer. The footer stores chunk metadata, such as chunk ID, chunk length, and the CRC checksum.

The self-contained chunk layout also allows the chunkserver to recover from failures by itself. Specifically, when a client writes consecutive self-contained chunks to the storage media, the chunkserver stores a copy of the metadata of these chunks in the memory, and periodically takes checkpoints of this information to the storage media. When a failure happens and leads to some unfinished write operations, the chunkserver loads the metadata from the checkpoints and compares it with the metadata in chunks. If discrepancies happen, the chunkserver checks the CRC of the chunks to recover to the latest correct state.

3.1.4 Metadata Operation Optimization

The Pangu masters provide two metadata services: the namespace service is responsible for directory tree and file management, and the stream service is responsible for chunk information. Stream is an abstraction of a group of chunks. Chunks in the same stream belong to the same file. Both services use a distributed architecture for better scalability. They partition metadata by considering metadata locality and load balancing (*i.e.*, partition by directory tree first and then by hashing). We design multiple mechanisms to optimize the efficiency of metadata operations.

Parallel metadata processing. Both namespace and stream services adopt parallelization processing (*e.g.*, InfiniFS [13]) to meet the low-latency requirement of metadata access. Specifically, Pangu uses a hash algorithm to map highly cohesive metadata to different metadata servers. It also uses a new data structure that supports predictable directory IDs and allows clients to perform path parsing efficiently in parallel.

We also introduce several techniques to accelerate how clients retrieve chunk information from the stream service.

Big chunks with variable lengths. Pangu 2.0 chooses to use big chunks. This decision has three benefits. It reduces the number of metadata. It avoids unnecessary I/O latency caused by clients frequently requesting chunks. It also helps improve the lifespan of SSD. However, simply increasing the chunk size will increase the risk of fragmentation. As such, We introduce variable-length chunks (*e.g.*, sizes ranging from 1 MB to 2 GB). For example, the chunk size of the EBS service has a 95% quantile of 64 MB and a 99% quantile of 286.4 MB. Variable-chunk length reduces the probability of fragmentation and maintains compatibility with Pangu 1.0.

Caching chunk information at clients. Each client maintains a local metadata cache pool to reduce the number of metadata query requests. The pool is updated using an LRU-based mechanism. When an application wants to access the data, the client first queries its metadata cache. It issues a new request to the masters to get the up-to-date metadata when (1) its cache is not hit; or (2) the cache is hit, but in response to the request, the corresponding chunkserver notifies the client that its cached metadata becomes stale (*e.g.*, due to replica migration).

Chunk information request batching. We let each client aggregate multiple chunk information requests over a short interval and send them in a batch to the masters to improve the query efficiency. The masters process the batched requests in parallel, aggregate the results and send them back to the client. The client disaggregates the results and dispatches them to corresponding applications.

Speculative chunk information prefetching. We design a greedy, probabilistic prefetching mechanism to reduce the number of chunk information requests. When the masters receive a read request, they respond to the client with the metadata of the related chunk and the metadata of other chunks. When the masters receive a write request, the masters respond with more available chunks than the client requested. In this way, the client can switch chunks without requesting new chunks if it encounters write exceptions.

Data piggybacking to reduce one RTT. We are motivated by QUIC [25] and HTTP3 [26] to use data piggybacking to improve the write latency. Specifically, after a client retrieves the chunk address from the masters, it merges the chunk creation request and the data to write into one request and sends it to the chunkserver. As a result, we are able to reduce the write latency by one RTT.

3.2 Chunkserver USSOS

In Pangu, the chunkserver is responsible for carrying out all the data operations. As such, it is essential to carefully design the run-time operating system to ensure that data operations can be finished with low latency and high throughput. With the emerging high-speed network technology and storage media, sticking to the traditional design that puts data operations through kernel space is inefficient. In particular, this would incur not only frequent system interrupts, which consume CPU resources, but also unnecessary data duplication between the user space and kernel space.

To cope with these issues, we resort to the kernel-bypassing design to develop a high-performance user-space storage operation system [17] for the chunkserver, which provides a unified user-space storage software platform. Aside from realizing device management and run-to-completion thread model [17, 18] in USSOS, Pangu also realizes user-level memory management (§3.2.1), lightweight user-space scheduling strategy (§3.2.2), and a customized high-performance append-only user-space storage file system (USSFS) for SSDs (§3.2.3).

3.2.1 User-Level Memory Management

Chunkserver USSOS is built based on existing user-space technologies (*e.g.*, RDMA in the network stack, DPDK [27], and SPDK [28] in the storage stack). But we go beyond and unify these two stacks to further reduce the latency and improve the performance of data operations. First, we make use of the run-to-completion thread model. In the traditional pipeline thread model, a request is decomposed into individual stages and each stage runs on a thread. In contrast, in USSOS, the request is run on one thread from beginning to end in the run-to-completion model, reducing the overhead caused by context switch, inter-thread communication, and inter-thread synchronization. Second, the thread requests a huge-page memory space to serve as a shared memory between the network and the storage stacks. To be concrete, data received from the network can be stored in this shared huge-page memory using RDMA protocol. After sending the metadata of the huge-page memory (*e.g.*, its address and size), data can be written directly from the huge-page memory to the storage media via SPDK frame. This way, we achieve zero copy between the network and the storage stacks during the data transmission and storage procedure. Besides, through the user-level shared huge-page memory for I/O data, data transmission operations among different roles (*e.g.*, the chunkserver and the garbage collection worker) can also achieve zero copy.

3.2.2 User-Space Scheduling Mechanism

In a real production environment, we encounter performance glitches brought by problems like task scheduling. Here, we introduce three key designs to optimize CPU scheduling in USSOS to improve the performance of Pangu.

Preventing a task from blocking the subsequent ones. As explained in §3.2.1, the run-to-completion thread model helps achieve zero-copy between the user-space network and storage stacks by using shared huge-page memory. However, each chunkserver has a fixed number of working threads. A new request is dispatched to a working thread based on the hash value of the file in the request. Requests assigned to the same working thread are executed in a first-in-first-serve order. As such, given one request, if one of its tasks takes too much time (*e.g.*, table lookup, table search, traversal, memory allocation, monitoring, and statistics), it will hog resources and block subsequent tasks. This issue degrades the performance of this request and leaves other requests to starve. To solve this problem, we take different measures for different scenarios. For heavy tasks, Pangu introduces the heartbeat mechanism to monitor the execution time of tasks and set an alarm. If a task runs out of time slice, it would be put into a background thread to remove it from the critical path. For system overhead, Pangu uses TCMalloc's cache [29] to allow high-frequency operations to be completed in the cache.

Priority scheduling to guarantee high QoS. Pangu assigns different QoS objectives for different requests (*e.g.*, user requests are assigned a high-priority objective while GC requests are assigned a low one). However, a request with a low-priority objective may block a high-priority request that arrives later and is assigned to the same working thread. As a result, it is hard to guarantee that requests with higher QoS objectives can always receive higher priorities. To address this problem, USSOS creates priority queues. Then tasks can be put into the corresponding priority queue according to their QoS objectives.

Polling and event-driven switching (NAPI). USSOS adopts a switching mechanism between polling and event-driven modes to reduce the overhead of massive interrupt processing with a low CPU utilization [30]. Specifically, NIC provides a *fd* monitored by applications and notifies the applications of data arrival through the *fd* event. Applications are in the event-driven mode by default. When they receive a notification from NIC, they enter the polling mode. If they do not receive any I/O request for some time, they switch back to the event-driven mode and notify the NIC.

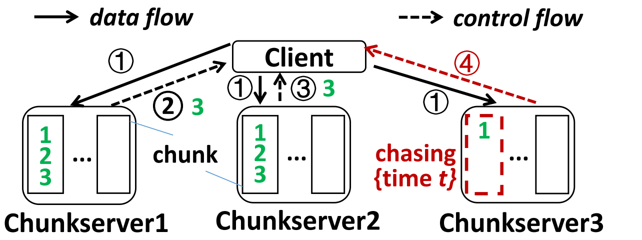


Figure 4: An illustrating example of chasing with *MaxCopy* = 3 and *MinCopy* = 2.

3.2.3 Append-Only USSFS

Previous file systems (*e.g.*, Ext4) could not make full use of the append-only FlatLogFile (§3.1.1) and SSDs with high throughput and low latency. Therefore, Pangu goes beyond and customizes the USSFS, a compact and high-performance user-space storage file system.

With the append-only semantic of FlatLogFile, USSFS supports append-only write and provides a set of chunk-based semantics, such as open, close, seal, format, read, and append, instead of standard POSIX semantics like Ext4. On this basis, it supports the append-only sequential write, which fully leverage of the sequential write-friendly feature of SSDs, and random read of successfully written data. Meanwhile, USSFS adopts different mechanisms to maximize the performance of SSDs. First, it can fully utilize the self-contained chunk layout (§3.1.3) to significantly reduce the number of data operations without using mechanisms such as page cache and journal. Second, it does not establish a hierarchical relationship between inodes and file directories like Ext4. All operations on files are recorded to log files. The corresponding metadata can be rebuilt by replaying the logs when mounting the file system. Third, we use the polling mode instead of an interrupt notification mechanism like Ext4 to maximize the performance of SSDs. Moreover, considering that the capacity of a single SSD node is tens or even hundreds of terabytes and the size of the chunk is usually 64 MB, we set the minimum space allocation granularity in USSFS as 1 MB. This choice considers both the size of memory used by space management metadata and SSD space utilization.

3.3 High Performance SLA Guarantee

Pangu introduces multiple mechanisms to provide high performance and a ms-level P999 SLA guarantee [31] in failure scenarios. *Chasing* copes with abnormal jitters (*e.g.*, network flash and packet retransmission caused by network incast). In these scenarios, exceptions occur in the cluster operating environment, but the system can recover to a normal state automatically in a short time. *Non-stop write* aims at unavailable chunks. *Backup read* reduces latency when the read request can not return within a limited time. *Blacklisting* isolates the chunkservers which provide poor service or fail.

Chasing. We design this mechanism to reduce the impact of system jitters on write latency. It allows the client to return success to the application when *MinCopy* out of *MaxCopy* replicas are successfully written in chunkservers, where $2 \times MinCopy > MaxCopy$. Figure 4 illustrates how chasing works with $MaxCopy = 3$ and $MinCopy = 2$. Suppose the application asks the client to write data $[1,2,3]$ to 3 chunk replicas. At time T, chunkserver 1 and 2 return success to the client's write operation but chunkserver 3 has not. The client returns success to the application. But it keeps the chunk in its memory and waits for another period t, an empirical *ms*-level threshold. If chunkserver 3 returns success to the client before $T+t$, the client releases the chunk from its memory. If chunkserver 3 does not finish the write, but the unfinished part is smaller than an empirical threshold k, the client issues a retry on chunkserver 3. If the unfinished part is larger than k, the client will seal this chunk at chunkserver 3 so that this chunk will not have subsequent append operation. The client then notifies the masters, who will replicate the data on a different chunk from chunkserver 1 or 2 to ensure there are a total of 3 replicas eventually.

Our analysis shows that with a careful choice of t and k, chasing substantially reduces the write tail latency without increasing the risk of data loss. Specifically, taking the case of $MaxCopy = 3$ as an example, after two replicas are successfully written to chunkservers, three replicas are in the system, whereas the third one is in the memory of the client. During $[T, T+t]$, data loss can only happen when the SSDs of the two chunkserver replicas are damaged and the last replica in the client memory also fails or the cluster powers down. Because SSDs' annual failure rate (*i.e.*, $\sim 1.5\%$ [32–37]) and servers' annual downtime rate (*i.e.*, $< 2\%$ [38, 39]) are close, the probability of data loss when two replicas are written successfully and the third replica is chasing is approximately the same as that when all three replicas are written successfully. Pangu has deployed chasing for over a decade and has not experienced any data loss caused by chasing. Recent studies also started to investigate this early-write-acknowledgment mechanism [40–42].

Non-stop write. We design this mechanism to reduce the write latency when a chunk write fails. When the failure happens, the client seals the chunk and reports the successfully written data length to the masters. It then uses a new chunk to continue writing the unfinished data. If the data written to the sealed chunk is corrupted, we use other replicas to duplicate a copy of this data to the new chunk in the background traffic. If no replica is available, the client writes this data to the new chunk again.

Backup read. To reduce the read latency under dynamic environments, the client sends additional read requests to other chunkservers as backups before receiving the response of the previously sent read request. This mechanism has two key parameters, the number and waiting time of sending backup read requests. To this end, Pangu calculates the latency of different disk types and I/O sizes and uses this information to dynamically adjust the time to send backup read requests. It also limits the number of backup read requests to control the system's load.

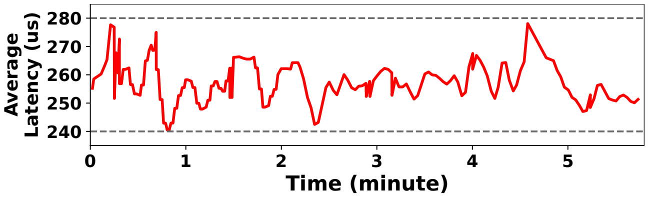


Figure 5: The average latency of database access to Pangu on Double 11 Festival in 2018.

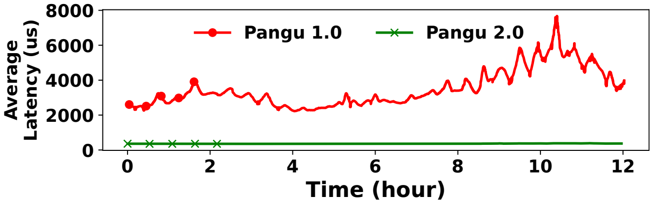


Figure 6: The average latency of OTS querying of Pangu 1.0 and Pangu 2.0 under the same stress test.

Blacklisting. To avoid sending I/O requests to chunkservers with poor service quality, Pangu introduces two blacklists, deterministic blacklist and non-deterministic blacklist. When Pangu determines that a chunkserver is unserviceable (*e.g.*, SSDs of a chunkserver are damaged), this server will be added to the deterministic blacklist. If a chunkserver can provide service, but its latency exceeds a certain threshold, it will be added to the non-deterministic blacklist with a probability that increases with its service latency. If the server's latency exceeds the median latency of all servers by several times, it is directly added to the non-deterministic blacklist with a probability of one. To release servers from these blacklists, clients send I/O probes to those servers periodically (*e.g.*, every second). If a server on the deterministic blacklist successfully returns the response of this request, it will be removed from this blacklist. For a server on the non-deterministic blacklist, Pangu decides whether to remove the server from the blacklist based on the time it takes to receive the response to this request.

Pangu limits the total number of servers on the blacklists to ensure system availability. For each server, it introduces a grace period for adding/removing it to/from the blacklist to maintain system stability. In addition, because the failed servers in the TCP and RDMA links may be different, Pangu maintains separate blacklists for TCP and RDMA links, respectively, and takes I/O probes on both links to update them.

3.4 Evaluations

Figure 5 shows the latency of database (DB) access to Pangu under the peak of 550,000 transactions per second during the Double 11 Festival in 2018. This peak value is at least one order of magnitude larger than that of non-festival days. The DB consists of millions of databases (*e.g.*, databases of Taobao merchants), each of which contains millions of e-commerce users' data. During this process, the DB needs to query orders and record transactions for those users. Under such a peak transactions rate, the access latency is less than 280 μs, which proves the high performance of Pangu 2.0.

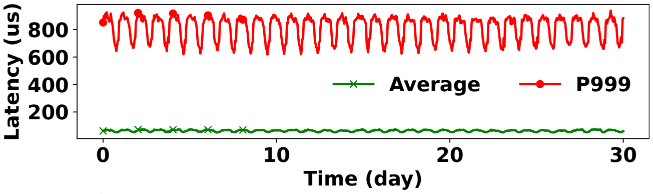


Figure 7: The write latency of EBS business.

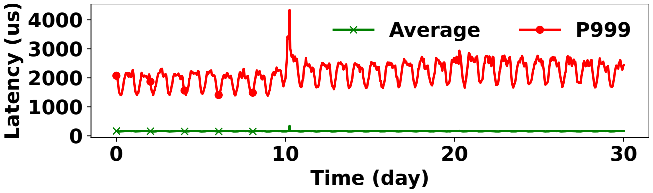


Figure 8: The read latency of online search business.

Figure 6 shows the latency of the cloud product OTS [9] querying with the same SSD and 25 Gbps network. Under the same query pressure, after upgrading from Pangu 1.0 to Pangu 2.0, the query latency is reduced by nearly an order of magnitude. This is mainly due to the low latency of read and write operations and the improved processing capability of a single thread in Pangu 2.0.

Figures 7 and 8 show the average latency and long tail latency of two clusters (online EBS and online search business) within one month. Online EBS is write-intensive and its online read/write ratio is nearly 1:10, which expects to achieve high throughput on write. As shown in Figure 7, its long tail latency is less than 1 ms. Online search business mainly services for the query and recommendation of Alibaba e-commerce (*e.g.*, search for goods on Taobao and Tmall), which expects to achieve high throughput on read. As shown in Figure 8, the long tail latency of read is less than 5 ms within one month. The results exhibit that Pangu 2.0 has a good guarantee for latency SLA.

4 Phase Two: Adapting to Performance-Oriented Business Model

Since 2018, Pangu gradually changes its role from a volume-oriented storage provider to a performance-oriented provider. This change in business model and the fast expansion of Pangu's clientele require Pangu to keep upgrading the infrastructure. However, scaling the infrastructure with original servers and switches along a Clos-based topology is not economical in many ways, including a higher financial and environmental cost (*e.g.*, a higher carbon emission rate). As such, Pangu develops its in-house storage server Taishan. A Taishan server is equipped with 2×24 core Skylake CPUs, 12×8 TB commodity SSDs, 128 GB DDR memory, and 2×dual-port 100 Gbps NICs. Although we could continue to increase its storage volume at the moment, we choose not to do so to maintain a high-level write IOPS/GB to suit the need of the performance-oriented business model. With optimizations made by SSD manufacturers (*e.g.*, caching and channel), the SSD throughput of a single Taishan server can reach more than 20 GB/s.

With such high-performance storage servers, it is natural to observe that other resources (*e.g.*, network, memory, and CPU) become the performance bottleneck of Pangu. As such, we upgrade the network of Pangu from 25 Gbps RDMA to 100 Gbps. However, contrary to common perception, it is *non-trivial* to provide high-performance, low-latency I/O in such an upgraded infrastructure. That is because new hardware also comes with new technical challenges in large-scale deployment. To this end, we propose and deploy a series of novel techniques to optimize the operation of Pangu's massive network (§4.1), memory (§4.2), and CPU resources (§4.3).

4.1 Network Bottleneck

Pangu optimizes the network in two aspects: network bandwidth expansion (§4.1.1) and traffic optimization (§4.1.2).

4.1.1 Bandwidth Expansion

To match the throughput of all SSDs on a single storage node, Pangu upgrades the network bandwidth from 25 Gbps to 100 Gbps to increase its network capability. The success of network bandwidth expansion depends on the improving of software and hardware. For hardware, Pangu adopts high-performance NIC/RNIC, optical modules (QSFP28 DAC, QSFP28 AOC, QSFP28 [43]), single-mode/multi-mode fiber, and high-performance switches. For the network software stack, Pangu first adopts lossless RDMA and proposes various mechanisms to achieve large-scale RDMA deployment, such as shutting down NIC ports or temporialy switching from RDMA to TCP for a short time (*e.g.*, several seconds) when there are too many pause frames on the RDMA network [17]. However, these mechanisms cannot handle other issues of the pause frame based flow control (*e.g.*, deadlocks [44] and head-of-line blocking [45, 46]). As such, Pangu upgrades to lossy RDMA, in which pause frames are disabled, to avoid these problems and improve performance. More details about this bandwidth expansion (*e.g.*, how we address the interoperability issue of heterogeneous network hardware and software) can be found in our early paper [17].

4.1.2 Traffic Optimization

Other than increasing the network capability, we also tackle the network bottleneck by reducing the traffic amplification ratio. Specifically, the traffic amplification ratio is computed as the amount of data transmitted through the network divided by the actual file size. Take the workflow of the EBS service as an example (Figure 9). First, the EBS client sends a file (1x) to the Pangu client (step (a)). Second, the Pangu client transfers the file to 3 storage nodes to write 3 replicas (3x). Third, the garbage collection worker (GCWorker) reads the file (1x) and performs GC on it. For ease of exposition, we ignore the file size change before and after GC. In the end, the file is written back to storage nodes in the form of EC(8,3) (1.375x), which provides at least the same level of fault tolerance as 3-replica but uses less storage space. As such, the traffic amplification ratio of a file write can be up to 6.375x (1x+3x+1x+1.375x). In other words, the maximum data access bandwidth of EBS in a 100-Gbps network is less than 16 Gbps. To cope with the issue of the high traffic amplification ratio limiting the service capability of Pangu, we introduce two optimizations: EC and data compression.

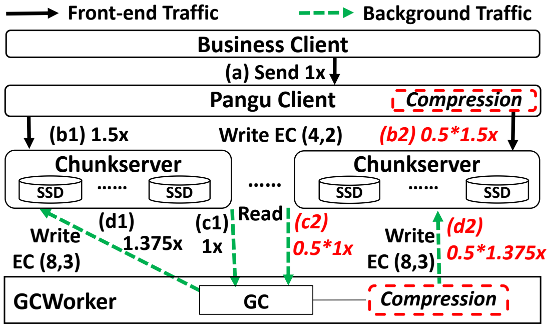


Figure 9: Pangu optimizes network traffic with 3 techniques: EC, compression, and balance between background and front-end traffic. The entire life cycle of a file we define is as follows. First, the business client sends a file (step a) to the Pangu client. Second, the Pangu client writes the file to the chunkserver in way b1 (b2 after compression). Third, the GCWorker reads the file from the chunkserver in way c1 (c2 after client compression) and performs garbage collection. Finally, the GCWorker writes the file in way d1 (d2 after GCWorker compression).

Use EC to replace 3-replica. Using EC to replace the 3-replica mechanism can substantially reduce the network traffic amplification while achieving a good level of fault tolerance. Take Figure 9 as an example. If we use EC (4,2) (step (b1)) to replace the 3-replica step, the network traffic amplification ratio can be reduced from 6.375x to 4.875x, the sum of step (a), (b1), (c1), and (d1).

Two challenges arise during this replacement. First, storing small files in EC is expensive because of the large number of zero-paddings needed to perform EC on data with a fixed length. We introduce multiple mechanisms to cope with this waste of space, including small write request aggregation and dynamic switching between EC and 3 replicas. Second, computing EC introduces a non-negligible latency. To this end, Pangu adopts Intel ISA-L [47], which reduces the latency of computing EC by 2.5 to 3 times compared with Jerasure [48].

Compressing FlatLogFile. We observe that FlatLogFile is highly redundant. As such, both the Pangu client and GCWorker compress the file before writing it to further reduce the traffic (*e.g.*, step (b2) and (d2) in Figure 9). We choose the LZ4 algorithm [49] to achieve efficient (de)compression. Empirical data in Pangu shows that the average compression rate can reach 50%. As such, in the example above, the traffic amplification ratios of step (b2), (c2) and (d2) can all be reduced by half. As a result, the traffic amplification ratio can be further reduced from 4.875x to 2.9375x, *i.e.*, the sum of step (a), (b2), (c2), and (d2).

Dynamic bandwidth allocation between front-end and background traffic. We dynamically adjust the *threshold* of usable network bandwidth for background traffic. For example, if there is sufficient empty space in the whole storage cluster, we temporally decrease the threshold to limit the bandwidth of background traffic (*e.g.*, the GC traffic), and let the frontend traffic use more bandwidth. For Taobao, Pangu sets a low threshold from daytime to midnight to cope with the large number of front-end access requests. After midnight, Pangu increases it because the front-end traffic decreases.

4.2 Memory Bottleneck

The fundamental memory bottleneck in Pangu lies in the high contention of memory bandwidth between network processes (*i.e.*, NIC performing DMA operations) and application processes (*e.g.*, data copy, data replication, and garbage collection) in the receiver host. Because NIC cannot acquire enough memory bandwidth, severe PCIe back-pressure is generated to the NIC. As a result, the NIC buffer is filled with in-flight packets. Eventually, it drops the overflowed ones, triggering the congestion control mechanism in the network and leading to overall performance degradation (*i.e.*, 30% network throughput drop, 5%-10% latency increase and 10% IOPS drop per server). This phenomenon is not unique to Pangu. Google also recently reported this issue [50].

We tackle this memory bandwidth bottleneck in three steps. First, we add more small-capacity DRAMs to the server to fully utilize the memory channels. Second, we switch background traffic from TCP to RDMA to reduce servers' memory bandwidth consumption (§4.2.2). Third, we design remote direct cache access (RDCA) to move memory out of the receiver host datapath and let senders access the receiver's cache directly (§4.2.3).

4.2.1 Adding Small-Capacity DRAMs

Because the bottleneck is memory bandwidth instead of memory capacity, we add more DRAMs with small capacity (*e.g.*, 16 GB) to servers to fully utilize the memory channels and increase the available memory bandwidth per server. We also enable non-uniform memory access (NUMA) to avoid across-sockets memory accesses being constrained by the ultra path interconnect [51].

4.2.2 Shifting Background Traffic From TCP to RDMA

In the 25-Gbps network, Pangu's background traffic was transmitted using TCP. It is to guarantee the QoS of front-end traffic because there is only one hardware queue for RDMA transmission on 25-Gbps switches.

With the network being updated to 100 Gbps, Pangu starts to transmit background traffic using RDMA to reduce the memory bandwidth consumption of network processes. It is because TCP needs at least four more memory copies than RDMA. By switching to RDMA, the memory bandwidth spent by background traffic is reduced by about 75%. To guarantee the QoS of front-end traffic, we design a host rate control mechanism similar to Linux *tc* [52,53] to control the rate of the background traffic being injected into the network.

4.2.3 Remote Direct Cache Access

In addition to increasing the available memory bandwidth and decreasing unnecessary memory bandwidth consumption, we propose the remote direct cache access (RDCA) architecture to let senders bypass the receiver's memory and access its cache directly. It is supported by an important observation in Pangu's production workload: the timespan data spent in memory after leaving NIC is very short (*i.e.*, hundreds of μs on average). Assuming a 200 μs average post-NIC timespan, for a dual-port 100 Gbps NIC, we only need 5 MB to temporally store the data leaving NIC. Although other cache accessing technologies (*e.g.*, DCA [54,55] and DDIO [56]) have been proposed, they suffer from the leaky DMA problem (*i.e.*, frequent cache eviction triggered by new arrival messages [57,58]). In contrast, RDCA goes beyond substantially to show that we can recycle a small area of LLC to support NIC operations at line rate with three components:

Cache-resident buffer pool. The pool uses a shared receiver queue (SRQ) for receiving small messages and a READ buffer equipped with a window-based rate control mechanism for receiving large messages, such that the memory buffer needed for RDMA operations can fit into the cache.

Swift cache recycle. In order to support 100-Gbps NIC operating at line rate with as few LLC as possible, we design the swift cache recycle mechanism to reduce data's post-NIC timespan by (1) processing data in parallel along a pipeline, and (2) optimizing processing using hardware offloading and lightweight (de)serialization.

Cache-pressure-aware escape mechanism. To deal with occasional jitters (*e.g.*, SSD slow write and application exceptions), the escape mechanism monitors the usage of reserved LLC and takes corresponding actions, including (1) replacing the cache buffer of straggler data by adding a new buffer to the cache-resident buffer pool, such that the size of the usable cache in RDCA to accommodate newly arriving requests remains unchanged; (2) actively copying the data of slow-running applications to memory if too many replacements happened, such that other applications can use the RDCA buffer pool and the pool does not take up too much cache; and (3) let the NIC mark explicit congestion notification (ECN) in congestion notification packets to indicate congestion if copying to memory fails or is insufficient in releasing the cache pressure.

We leverage Intel's DDIO [56] to implement RDCA on commodity hardware. Results of extensive evaluation in some clusters of Pangu show that for typical storage workloads, RDCA consumes a 12MB LLC cache (20% of the total cache) per server, decreases the average memory bandwidth consumption by ∼89% and improves network throughput by 9%. We find that RDCA is also effective in non-storage workloads, *e.g.*, it reduces the average latency of collective communications in latency-sensitive HPC applications by up to 35.1%. RDCA is rolled out in Pangu at the end of 2022.

4.3 CPU Bottleneck

Even with optimizations to break the network and memory bottleneck, the throughput of Pangu in a 100-Gbps network can still reach only 80% of its theoretical value. It is because operations such as data serialization and deserialization, data compression and data CRC computation consume many CPU resources, making CPU another bottleneck of Pangu. To this end, Pangu introduces a hybrid design to reduce (de)serialization operations (§4.3.1), a special hardware instruction *CPU Wait* to make full use of the CPU (§4.3.2), and a hardware/software co-design [59] to offload CRC computing and data compression to hardware (§4.3.3).

4.3.1 Hybrid RPCs

Serializing and deserializing RPC requests using Protobuf [60] in Pangu costs about 30% of CPU overhead. We observe that this overhead mostly happens in the data path over a small number of RPC types. As such, we take a hybrid design to handle this issue. We switch our data path operations to use a raw structure similar to FlatBuffer [61], to send and receive data directly without serialization. Pangu continues to use Protobuf for control operations due to its flexibility and complexity. As a result, network throughput for each CPU core increased by about 59%.

4.3.2 Supporting Hyper-Threading Using CPU Wait

Pangu initially did not use hyper-threading (HT) [62], but started to adopt it as the CPU core resources become scarce. However, HT has two main performance issues. First, two HTs on one physical core need to switch contexts. Second, one HT affects the execution of the other HT when they execute tasks simultaneously, resulting in increased latency on both tasks. For example, when a network idle-polling thread is running on one HT, and a compression thread running on the other HT from the same physical core is compressing 4 KB data with the LZ4 algorithm [49] and lzbeach [63] at the same time, the latency of data compression increases by 25%, compared with the case that the compression thread exclusively occupies the physical core.

To solve these two issues, Pangu introduces the *CPU wait* instruction. It consists of *monitor* and *mwait*. Pangu needs less than 5 ms to call them. Revisit the example above. After introducing CPU wait, the network idle-polling thread will *mwait* at the monitored memory address and does not wake up until the memory address is written by other threads. During the *mwait* process, the HT it runs on enters an idle sleep state, one of the C-States except for C0 [64, 65], without interfering with the other HT. In addition, the time of waking up HTs with system calls is of ms-level. As such, Pangu can fully utilize the CPU cores with high performance. In this example, the network throughput increases by 31.6%, compared with the case where *CPU wait* is not used.

4.3.3 Hardware and Software Co-design

For high performance, Pangu offloads some tasks from CPU to programmable hardware. First, data compression is offloaded to FPGA-based computational storage drives [66]. At a 3-GB/s throughput, hardware-based FPGA compression can save about 10 physical cores (Intel(R) Xeon(R) Platinum 2.5 Ghz). Second, CRC computation is offloaded to RDMA-capable NICs, which calculates the CRC of each data block. CPU then aggregates these CRC and performs a lightweight check [18]. This design ensures low CPU overhead and high application-level data integrity. At the same throughput, compared with using software, using hardware for CRC computation can save 30% of CPU overhead.

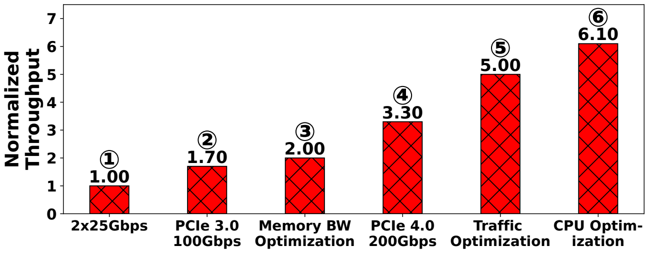


Figure 10: The normalized effective throughput per storage node of Pangu in the evolution process.

4.4 Evaluations

Figure 10 shows the normalized effective throughput per storage node (NETPSN) on the evolution roadmap of the storage node on EBS and high-performance ESSD [67].

Stage 1. To better illustrate the development of the performance of Pangu 2.0, we initialize NETPSN as 1 when Pangu introduces 2 × 25-Gbps network and 4-TB SSDs.

Stage 2. As SSD's capacity increases to 8 TB with higher performance, Pangu introduces a 100-Gbps network. The network bandwidth is 100 Gbps because the server supports PCIe gen3. Though the network bandwidth doubles, NETPSN increases from 1 to 1.7 instead of 2. It is because the memory bandwidth for read and write reaches 90 GB/s, close to the maximal memory bandwidth of the server (105 GB/s).

Stage 3. Insufficient memory bandwidth leads to packet drops at NICs, resulting in a large throughput drop. With the memory bandwidth optimizations in §4.2, we successfully increase NETPSN to 2.

Stage 4. After introducing PCIe gen4, the network throughput increases to 2×100 Gbps. The network bandwidth doubles, but NETPSN does not, *i.e.*, increased to 3.3 rather than 4, due to the traffic amplification problem (about 6x).

Stage 5. After the Pangu client adopts EC(4, 2), data compression, and other optimizations, the traffic amplification ratio reduces by half (∼3x) (§4.1). However, NETPSN increases to 5 instead of 6.6. The main reason is that data compression and other operations consume many CPU resources.

Stage 6. In order to solve the CPU bottleneck, Pangu offloads tasks (*e.g.*, data compression and CRC) from CPU to hardware (§4.3), eventually increasing NETPSN to 6.1.

These results show that by breaking the bottlenecks of network, memory, and CPU, Pangu achieves high performance and adapts to the new performance-oriented business model.

5 Operation Experiences

After introducing the design innovation in Pangu 2.0, we next introduce the basic operation cycle of Pangu, focusing on the Pangu monitoring system, and share our experiences in addressing several important issues during Pangu's operation.

5.1 Pangu's Operation Cycle

Pangu's basic operation cycle consists of five stages: planning, development, testing, deployment, and monitoring. Before entering development, new hardware and software solutions go through a rigorous planning phase (*i.e.*, feasibility analysis, benefit/cost analysis, and social and regulation studies). We conduct extensive tests on the solutions under various scenarios between development and deployment. In particular, Pangu copes with the interoperability issue among heterogeneous hardware from different vendors using an in-house, template-based admission test. New solutions are rolled out in Pangu's production environment cluster by cluster. After they are online, the Pangu monitoring system watches their behaviors closely via fine-grained monitoring, thorough root-cause analysis, fast response, and post-mortem documentation.

Fine-grained monitoring and intelligent diagnosis. In Pangu 2.0, we improve the Pangu monitoring system with two key designs to keep up with the high-performance requirement (*i.e.*, 100 μs-level I/O latency). First, we increase the time granularity of monitoring from 15 seconds to 1 second and extend the Log Service [68] to design an on-demand tracing system. Compared with the coarse-grained monitoring in Pangu 1.0, this allows us to perform tracings on a per-file-operation basis to accurately capture fine-grained abnormal events (*e.g.*, memory allocation exception and log printing timeout). Second, we embrace AI to better capture the causal relationship between abnormal events and their root causes. The inferred root causes are rated by operating teams and fed back to the trained model to improve its accuracy. This design substantially improves diagnostic accuracy and reduces the required human efforts.

5.2 Case Studies

Extensive data integrity checking. Pangu extensively employs CRC to ensure data integrity, such as end-to-end CRC along the data path, monthly CRC on all replicates, CRC on random sampled replicates and CRC on one extra replicate during EC building. Among all the data integrity issues we encountered, the CPU silent error is a representative one. Specifically, we find some end-to-end CRC errors and pinpoint their root causes as the silent errors on certain CPUs. To prevent such errors, we work with Intel to deploy silent error testing tools that run in production environments when the overall workload is low, and achieve some success.

Handling SLA jitters in USSOS. During Pangu's operation, we encounter and address several issues that contribute to SLA jitters in USSOS (§3.2), such as memory allocation, periodic heavyweight tasks and increased USSOS CPU utilization. First, we find existing memory allocation mechanisms (*e.g.*, TCMalloc [29]) become too time-consuming if they enter a global memory allocation phase (*e.g.*, when a new thread requires memory) or a memory organization phase (*e.g.*, when too many RDMA queue pairs try to reserve high-order memory space). To this end, we introduce a user-space memory allocation pool and optimize RDMA drivers to use anonymous pages. Second, for periodic heavyweight tasks (*e.g.*, log printing), we move them to asynchronous threads to avoid affecting the SLA of data operations. Third, we find that the CPU utilization of USSOS significantly increases when the memory occupation is high and USSOS needs to perform memory recollection. As such, we adjust the threshold of memory recollection to reduce the chance of USSOS entering memory recollection. We also recollect memory allocated to the buffer and cache in the background.

Handling correctable machine check exceptions (MCE) in the USSOS to improve availability. Initially, USSOS (§3.2) can monitor such hardware failures, but it cannot perceive how the kernel migrates the physical memory for exception isolation. As such, errors would happen when USSOS tries to access the already migrated physical memory based on its outdated virtual-physical memory address mapping.

To this end, we add a handler to the MCE monitor daemon in USSOS. Once the number of found correctable MCEs exceeds a threshold, the user-space process related to them will pause and let the handler notify the kernel to migrate the memory. After the migration, the process resumes and updates its mapping table before accessing memory pages. This design improves the availability of Pangu with negligible performance degradation. For example, we observe <330 correctable MCEs in a 2300-server cluster in 22 days.

Heterogeneous memory bandwidths from different vendors. Pangu deploys memories from different vendors. Our tests show that under a 1:1 memory read/write ratio, the achievable bandwidths of 128 GB memory from three different vendors are 94 GB/s, 84 GB/s, and 60 GB/s, respectively (*i.e.*, a 57% difference). Such heterogeneous memory bandwidths would cause performance degradation in clusters. This observation taught us to pay more attention to the performance of memory, instead of the capacity. It is also our earliest evidence of congestions in the receiver host datapath and a direct motivation for RDCA (§4.2.3).

Coping with tail latency surge during Double 11 Festival. This festival is Alibaba's largest annual online shopping event. Guaranteeing Pangu's high performance under such high-pressure traffic requires real-time monitoring, diagnosis, and response to ensure all technical features work in harmony. On the Double 11 Festival in 2019, we noticed a surge of read tail latency in the Relational Database Service (RDS) built on top of Pangu. By analyzing the system traces, we identify the root cause as the increased simultaneous occurrence of rebalancing migration of 2 chunks and the failure of chunkserver storing the remaining unmigrated chunk. As such, clients have to try and fail to access all three replicas before requesting the

IP addresses of the latest chunkservers, increasing latency. To fix this issue, we let the clients periodically fetch chunk information of abnormal chunkservers from the masters and immediately request the latest chunk metadata if their needed chunks are on abnormal servers. This action works well in coping with the surge of tail latency back then.

However, this mechanism has its limitation. On the Double 11 Festival 2020, clients experience the surge of tail latency again. We find that the root cause is that many chunkservers are determined as abnormal due to an internal issue. Clients then receive a large amount of information about abnormal chunkservers and spend many resources processing them (*e.g.*, deserialization and address resolution), resulting in long-time I/O hang. Facing the upcoming peak traffic, we temporarily disable this mechanism and upgrade it after the festival with a series of improvements, including independent threads for abnormal server operations and limiting the number of abnormal servers requested each time.

6 Lessons

Lessons on user-space systems. We develop Pangu's chunkserver USSOS (§3.2) to keep up with the high speed of new network and storage technologies. During its development and operation, we learned three lessons. First, user-space systems are simpler to develop and operate. For example, our data shows that bug fixing takes about two months in a kernel-space file system, but a couple of weeks in USSFS (§3.2.3). Developing new features (*e.g.*, zoned namespace [69]) in the user space requires fewer developers and a shorter time. Furthermore, it is also easier to monitor and trace behaviors of user-space systems and adjust their parameters accordingly.

Second, developing user-space systems should learn from the design of the kernel space. In particular, to build a high-performance USSOS, not only we need to unify the storage and network stack, we also need to design user-space modules for memory management, CPU scheduling and hardware failure handling. The kernel space is pretty good at these functionalities. As such, user-space systems can benefit by learning from it.

Third, the performance gain of user-space systems is not exclusive to high-speed storage such as SSD. Specifically, we provide a series of mechanisms in Pangu's USSFS to accelerate the performance of HDD. For example, USSFS takes advantage of the self-contained chunk layout (§3.1.3) to save the number of metadata operations. It leverages the differences between internal and external tracks of disks to improve HDD's write efficiency.

Lesson on performance-cost tradeoff. To meet new business requirements, Pangu usually first chooses to add more hardware to improve its performance based on total cost of ownership (TCO) balance (*e.g.*, upgrading the network from 25 Gbps to 100 Gbps, increasing the number of memory channels by placing more small-volume DRAMs and upgrading servers with more powerful CPUs). Hardware expansion effectively improves Pangu's performance, but is not sustainable due to the cost incurred. As such, Pangu also spends substantial efforts, such as traffic optimization (§4.1.2), improving its resource utilization and efficiency.

Lesson on persistent memory (PMem). PMem has many advantages, such as fast data persisting, RDMA friendliness, low read latency (6 μs on PMem vs. 80 μs on SSD), low tail latency, and cache friendliness. As such, we developed a 30-μs PMem-based EBS service [1] in Pangu. However, Intel's decision to kill off its PMem business [70] forces us to rethink this service. We need to plan more thoroughly when developing new services (*e.g.*, considering substitutability, sustainability, and cost tradeoffs). But we are optimistic that new storage class memory [71] will emerge with better solutions to these issues.

Lesson on hardware offloading. The cost vs. benefits tradeoff is a fundamental issue for hardware offloading (§4.3.3), and has been an ever-lasting debate topic in Pangu since 2018. The entire development of hardware offloading compression takes a 20-person team two years, during which we resolve many issues such as the FPGA hardware cost, the integrity of compressed data, and the co-existence with other functions in hardware. In the end, the outcome benefits outweigh this cost substantially. Hardware offloading significantly reduces the compression's average and tail latency, effectively reducing the network traffic within a low latency. As a result, we can improve the service provision capability of our infrastructure by $\sim$50%. We rolled out hardware compression in Pangu in 2020 at a slow pace, first in internal services (*e.g.*, log/monitoring services) and then gradually expanding to core external services (*e.g.*, EBS). To prevent potential bugs in hardware from harming data integrity, we perform data decompression and CRC checking on hardware and conduct routine spot software CRC checking. Since 2022, all 200 Gbps clusters in Pangu enable hardware compression by default, and incidents happen less and less often.

7 Conclusion

We introduce how we embrace the emerging hardware technologies and adapt to the shift of business model to evolve the Pangu to provide high-performance, reliable storage services with a 100μs-level I/O latency. We also share our experiences operating Pangu 2.0 to shed light on future research in large-scale, high-performance storage systems.

Acknowledgments. We are extremely grateful for our shepherd, Peter Macko, and the anonymous FAST'23 reviewers for their wonderful feedback. Qiao Xiang, Haohao Song, Yuxin Wang, and Ridi Wen are supported in part by the National Key R&D Program of China 2022YFB2901502, Alibaba Innovative Research Award, NSFC Award 62172345, Open Research Projects of Zhejiang Lab 2022QA0AB05, MOE China Award 2021FNA02008, NSF-Fujian-China 2022J01004, and IKKEM Award HRTP-2022-34.

References

[1] Alibaba Group. Alibaba Cloud Products & EBS. https://www.aliyun.com/product/disk. Accessed Sept 18, 2022.

[2] Alibaba Group. Alibaba Cloud Products & OSS Services. https://www.alibabacloud.com/zh/product/object-storage-service. Accessed Sept 18, 2022.

[3] Alibaba Group. Alibaba Cloud Products & NAS Services. https://www.alibabacloud.com/zh/product/nas. Accessed Sept 18, 2022.

[4] Alibaba Group. Alibaba Cloud Products & PolarDB. https://www.alibabacloud.com/zh/product/polardb. Accessed Sept 18, 2022.

[5] Alibaba Group. Alibaba Cloud Products & MaxCompute. https://www.alibabacloud.com/zh/product/maxcompute. Accessed Sept 18, 2022.

[6] 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*, pages 21–33. Citeseer, 2007.

[7] Wright Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. Technical report, 1997.

[8] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM'08*, pages 63–74. ACM, 2008.

[9] Alibaba Group. Alibaba Cloud Products & OTS. https://www.alibabacloud.com/zh/product/table-store. Accessed Sept 18, 2022.

[10] 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 *FAST'21*, pages 217–231. USENIX Association, 2021.

[11] Colossus under the Hood: A Peek into Google's Scalable Storage System. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system. Accessed Sept 18, 2022.

[12] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *ATC'14*, pages 305–319. USENIX Association, 2014.

[13] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems. In *FAST'22*, pages 313–328. USENIX Association, 2022.

[14] Su Zhou, Erci Xu, Hao Wu, Yu Du, Jiacheng Cui, Wanyu Fu, Chang Liu, Yingni Wang, Wenbo Wang, Shouqu Sun, Xianfei Wang, Bo Feng, Biyun Zhu, Xin Tong, Weikang Kong, Linyan Liu, Zhongjie Wu, Jinbo Wu, Qingchao Luo, and Jiesheng Wu. Deployed System: SMRSTORE: A Storage Engine for Cloud Object Storage on HM-SMR Drives. In *FAST'23*. USENIX Association, 2023.

[15] Qiang Li, Lulu Chen, Xiaoliang Wang, Shuo Huang, Qiao Xiang, Yuanyuan Dong, Wenhui Yao, Minfei Huang, Puyuan Yang, Shanyang Liu, et al. Fisc: A Large-Scale Cloud-Native-Oriented File System. In *FAST'23*. USENIX Association, 2023.

[16] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, and Jiesheng Wu. Deployed System: Perseus: A Fail-Slow Detection Framework for Cloud Storage Systems. In *FAST'23*. USENIX Association, 2023.

[17] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When Cloud Storage Meets RDMA. In *NSDI'21*, pages 519–533. USENIX Association, 2021.

[18] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From Luna to Solar: the Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *SIGCOMM'22*, pages 753–766. ACM, 2022.

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP'03*, pages 29–43. ACM, 2003.

[20] Dhruba Borthakur. HDFS Architecture Guide . https://hadoop.apache.org/docs/r1.2.1/hdfs_design.htmll. Accessed Sept 15, 2022.

[21] Microsoft. Azure Storage. https://azure.microsoft.com/en-us/products/category/storage/. Accessed Sept 18, 2022.

[22] 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 *OSDI'06*, pages 307–320. USENIX Association, 2006.

[23] AWS. Cloud Storage on AWS. https://aws.amazon.com/products/storage/. Accessed Sept 18, 2022.

[24] Zhu Pang, Qingda Lu, Shuo Chen, Rui Wang, Yikang Xu, and Jiesheng Wu. ArkDB: A Key-Value Engine for Scalable Cloud Storage Services. In *SIGMOD'21*, pages 2570–2583. ACM, 2021.

[25] Chromium Contributor. Jim Roskind. QUIC: Design Document and Specification Rationale. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit. Accessed Jan 3, 2023.

[26] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM'17*, pages 183–196. ACM, 2017.

[27] Intel. Data Plane Development Kit. https://dpdk.org/. Accessed Aug 25, 2022.

[28] SPDK. Storage Performance Development Kit. https://www.spdk.io/. Accessed Aug 25, 2022.

[29] Sanjay Ghemawat and Paul Menage. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html. Accessed Aug 25, 2022.

[30] J. Yang, D. B. Minturn, and F. Hady. When Poll Is Better than Interrupt. In *FAST'12*, pages 1–7. USENIX Association, 2012.

[31] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[32] Backblaze. The SSD Edition: 2022 Drive Stats Midyear Review. https://www.backblaze.com/blog/ssd-drive-stats-mid-2022-review/. Accessed Sept 18, 2022.

[33] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? and Why? In *SYSTOR'16*, pages 1–11. ACM, 2016.

[34] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *FAST'16*, pages 263–276. USENIX Association, 2016.

[35] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *FAST'16*, pages 67–80. USENIX Association, 2016.

[36] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and Actions: What We Learned from 10K SSD-Related Storage System Failures. In *ATC'19*, pages 961–976. USENIX Association, 2019.

[37] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the Robustness of SSDs under Power Fault. In *FAST'13*, pages 271–284. USENIX Association, 2013.

[38] Alibaba Group. Alibaba Cloud Products & ECS. https://www.aliyun.com/product/ecs. Accessed Sept 18, 2022.

[39] Amazon. Amazon EC2. https://aws.amazon.com/cn/ec2/?nc2=h_ql_prod_fs_ec2. Accessed Sept 18, 2022.

[40] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *OSDI'16*, pages 401–417. USENIX Association, 2016.

[41] Takayuki Fukatani, Hieu Hanh Le, and Haruo Yokota. Delayed Parity Update for Bridging the Gap between Replication and Erasure Coding in Server-Based Storage. In *ADMS@ VLDB*, pages 1–9, 2021.

[42] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. Hydra: Resilient and Highly Available Remote Memory. In *FAST'22*, pages 181–198. USENIX Association, 2022.

[43] QSFP. https://community.fs.com/search?key_word=QSFP. Accessed Sept 18, 2022.

[44] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *SIGCOMM'16*, pages 202–215. ACM, 2016.

[45] Mark J. Karol, S. Jamaloddin Golestani, and David Lee. Prevention of Deadlocks and Livelocks in Lossless Backpressured Packet Networks. *IEEE/ACM Trans. Netw.*, 11(6):923–934, 2003.

[46] Brent E. Stephens, Alan L. Cox, Ankit Singla, John B. Carter, Colin Dixon, and Wes Felter. Practical DCB for Improved Data Center Networks. In *INFOCOM'14*, pages 1824–1832. IEEE, 2014.

[47] Intel(R) Intelligent Storage Acceleration Library. https://github.com/intel/isa-l. Accessed Sept 18, 2022.

[48] Jerasure: A Library in C Facilitating Erasure Coding for Storage. https://jerasure.org. Accessed Sept 18, 2022.

[49] LZ4 - Extremely fast compression. https://github.com/lz4/lz4. Accessed Sept 18, 2022.

[50] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, et al. Understanding Host Interconnect Congestion. In *HotNets'22*, pages 198–204. ACM, 2022.

[51] Intel. Intel Stratix 10 FPGAs & SoC FPGA. https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10.html. Accessed Sept 18, 2022.

[52] Bert Hubert et al. Linux Advanced Routing & Traffic Control HOWTO. *Netherlabs BV*, 1:99–107, 2002.

[53] Alok Kumar, Sushant Jain, Uday Naik, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *SIGCOMM'15*, pages 1–14. ACM, 2015.

[54] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *ISCA'05*, pages 50–59. IEEE, 2005.

[55] Amit Kumar, Ram Huggahalli, and Srihari Makineni. Characterization of Direct Cache Access on Multi-Core Systems and 10GbE. In *HPCA'09*, pages 341–352. IEEE, 2009.

[56] Intel® Data Direct I/O Technology (Intel® DDIO): A Primer. https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html Accessed Aug 28, 2022.

[57] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *NSDI'18*, pages 283–297. USENIX Association, 2018.

[58] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *NSDI'19*, pages 361–378. USENIX Association, 2019.

[59] Giovanni De Michell and Rajesh K Gupta. Hardware/Software Co-Design. *Proceedings of the IEEE*, 85(3):349–365, 1997.

[60] Protocol Buffers are a Language-Neutral, Platform-Neutral Extensible Mechanism for Serializing Structured Data. https://developers.google.com/protocol-buffers/. Accessed Sept 18, 2022.

[61] FlatBuffers is an Efficient cross Platform Serialization Library. https://google.github.io/flatbuffers/. Accessed Sept 18, 2022.

[62] William Magro, Paul Petersen, and Sanjiv Shah. Hyper-Threading Technology: Impact on Compute-Intensive Workloads. *Intel Technology Journal*, 6(1):1–9, 2002.

[63] Lzbench, an in-Memory Benchmark of Various Compressors. https://openbenchmarking.org/test/pts/lzbench. Accessed Sept 18, 2022.

[64] Intel. Intel C-states. https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top/reference/energy-analysis-metrics-reference/c-state.html. Accessed Sept 18, 2022.

[65] Intel. C-states. https://www.thomas-krenn.com/en/wiki/Processor_P-states_and_C-states. Accessed Sept 18, 2022.

[66] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *FAST'20*, pages 29–41. USENIX Association, 2020.

[67] Alibaba Group. Alibaba Cloud Products & ESSD. https://www.aliyun.com/storage/storage/essd/. Accessed Sept 18, 2022.

[68] Alibaba Group. Alibaba Cloud Products & SLS. https://www.aliyun.com/product/sls. Accessed Sept 18, 2022.

[69] Matias Bjørling. From Open-Channel SSDs to Zoned Namespaces. In *Proc. Linux Storage Filesyst. Conf.(Vault)*, pages 1–18, 2019.

[70] Intel. Intel Reports Second-Quarter 2022 Financial Results. https://download.intel.com/newsroom/2022/corporate/Intel-CEO-CFO-2Q22-earnings-statements.pdf. Accessed Sept 18, 2022.

[71] Chung H Lam. Storage Class Memory. In *ICSICT'10*, pages 1080–1083. IEEE, 2010.

λ-IO: A Unified IO Stack for Computational Storage

Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu*

Tsinghua University

Abstract

The emerging computational storage device offers an opportunity for in-storage computing. It alleviates the overhead of data movement between the host and the device, and thus accelerates data-intensive applications. In this paper, we present λ-IO, a unified IO stack managing both computation and storage resources across the host and the device. We propose a set of designs – interface, runtime, and scheduling – to tackle three critical issues. We implement λ-IO in full-stack software and hardware environment, and evaluate it with synthetic and real applications against Linux IO, showing up to 5.12× performance improvement.

1 Introduction

Data-intensive applications (e.g., data mining) suffer from the overhead of loading sizable data from the storage device for processing. Emerging computational storage devices offer the opportunity to alleviate the bother and thus gain increasing attention in recent years [1–3]. They encapsulate in-device computation resources, which enables the applications to offload the computation down into the devices for in-storage computing (ISC) [4, 5], therefore mitigating the overhead of data transfer between the host and the device.

As both the host and the device can conduct computation, we observe from experiments that blindly pushing computation tasks to the devices is not always optimal; instead, applications may be faster on either side regarding to various application features (e.g., computation complexity) and the ever-changing status of the host and the device (e.g., cache ratio, bandwidth consumption). Since the operating system has delivered a sophisticated IO stack with mature functionalities and interfaces, we explore a fundamental question in the paper – *how to build a unfied IO stack managing both computation and storage resources across the host and the device?* For answering the question, we first identify three critical issues to achieve the goal.

1) Interface. IO stack has been evolving the read and write interfaces [6, 7], which are widely used by existing applications. We are committed to endowing them with the computation capability of the host and the device at a minimum change, i.e., unified interfaces for both sides that are generic and rely on no specific file systems.

2) Runtime. IO stack contains components like the page cache, file system, and device driver for storage management, but has no computation runtime. Thus, we need to build a unified runtime that enables executing the same piece of computation task on either the host or the device. eBPF [8, 9] is the preference for constructing such runtime: it proposes a hardware-independent intermediate bytecode format, so that the program can be compiled once and run on various ISAs. However, eBPF is still inapplicable to ISC due to its overstrict static verification (§2.3).

3) Scheduling. IO stack features rich scheduling mechanisms for reads and writes, but computation makes the dispatching mechanism more complicated. As application performance varies over features and system status as mentioned above, we aim at detecting and dispatching requests to the faster side effectively and efficiently.

In this paper, we propose λ-IO, a unified IO stack that comes with three key designs to tackle the aforementioned issues. First, λ-IO extends IO stack with λ extension across the host and the device to support ISC. Besides normal IOs, the extended interfaces enable an application to submit λ requests to customize, load and invoke computational logic (λ function) during reading and writing a file. With the extended interfaces, developers can still use their familiar programming style to access and process file data, hiding the details of how computation tasks are executed and scheduled.

Second, λ-IO proposes unified λ runtime that crosses the host-and-device boundary with two counterparts. At the core of λ runtime is sBPF (s stands for storage). For managing and executing λ functions atop heterogeneous hardware of both sides, λ-IO extends eBPF to sBPF, which supports pointer access and dynamic-length loop (loops are bounded at runtime to a specified threshold). In contrast to static verification of

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

eBPF, sBPF brings in dynamic verification with extra information from λ-IO.

Third, λ-IO adopts dynamic request dispatching to designate requests to the faster side effectively and efficiently. For effectiveness, we model the execution time of a λ request in both sides, so that the λ-IO finds the faster side. For efficiency, λ-IO profiles partial requests periodically to determine variables in the execution time model.

We implement λ-IO in the full-stack software and hardware environment (§4). We modify Linux kernel to integrate λ extension and build an NVMe device with λ extension support on a real hardware platform. We compare λ-IO and vanilla Linux IO on synthetic applications, showing 5.12× improvement. We also port a real application, Spark SQL [10], to λ-IO and evaluate its end-to-end performance with TPC-H [11], showing up to 2.15× improvement. We open source λ-IO at `https://github.com/thustorage/lambda-io`.

In summary, we make the following contributions.

- We present λ-IO, a unified IO stack managing both computation and storage resources across the host and the device.
- We propose a set of designs, unified interfaces, λ runtime, and dynamic request dispatching, to exploit the benefits of both sides.
- We implement λ-IO in the full-stack software and hardware environment, and evaluate it with synthetic and real applications against vanilla Linux IO, showing significant performance improvement.

2 Background and Motivation

2.1 In-Storage Computing and IO Stack

In-storage computing (ISC) originates in the disk era [4, 12], aiming to ship computation closer to storage. It mitigates the data movement overhead and exploits the in-device internal bandwidth. ISC gets revitalized with the advent of SSDs [5, 13], as an SSD typically has more powerful computation resources and higher internal bandwidth. Besides the researches for specific domain acceleration [14–22], general frameworks [23–27] allow programmers to define and offload their own computational logic. They mostly focus on providing manipulation interfaces in the userspace and accelerating computation in the device, but do not fully exploit the host-side computation and storage resources.

IO stack is a fundamental part of the operating system for managing storage devices, including the device driver, the block layer, and the file system. Although userspace IO libraries like SPDK [28] arise for their kernel-bypass low latency, the IO stack is still indispensable in most scenarios for three reasons. **1) Compatibility**. A sea of applications rely on POSIX file interfaces to access storage data. Programmers are also accustomed to leveraging APIs of the IO stack. **2) Functionality**. The IO stack offers abundant modules and functionalities, including the file system and the page cache. In contrast, a userspace IO library only supports data transfer with the raw storage device. The application has to build its own file system and data cache. **3) Sharing**. The IO stack has well-tested resource allocation and security mechanisms, so that users and applications can share the whole device. Sharing is also hard to be implemented in the userspace and absent from userspace IO libraries.

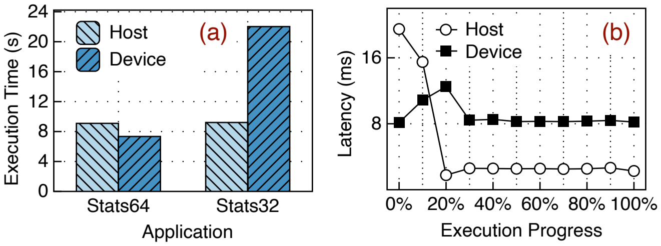


Figure 1: Performance Comparison of Host and Device. *Stats64 and Stats32 are two applications. They view the file data as 64-bit/32-bit integers and calculates the sum, maximum, and minimum. Refer to §5.2 for detailed settings.*

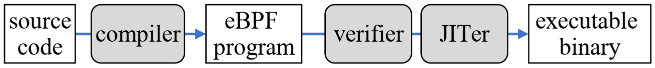


Figure 2: The Progress to Run eBPF/sBPF.

As ISC becomes increasing popular and the IO stack has unique advantages, we explore how to incorporate ISC to the IO stack in this paper.

2.2 Host-Device Coordination

We analyze the need for host-device coordination through application experiments. The key observation is that either the host or the device may be faster to run an application. 1) Different applications have different features and thus prefer different sides. We run Stats64 and Stats32 either in the host or the device (detailed settings in §5.2). As reported in Figure 1(a), Stats64 runs faster in the device while Stats32 runs faster in the host. 2) Even one application also favors different sides during the execution progress. We measure the request latency on both sides when running Stats64 with the warmed page cache (detailed settings in §5.4.1). As reported in Figure 1(b), earlier requests are faster in the device as the host does not cache data. The host outperforms significantly when it has the data in cache after 20%.

We conclude that blindly pushing down all the computational logic to the device may deliver suboptimal application performance. Many factors affect the execution time, such as computation complexity, data size, and cache. Thus, one should take factors into consideration and dispatch computation to their preferred side for better performance.

2.3 eBPF and its Limitations

eBPF (extended Berkeley Packet Filter) [8] is an in-kernel virtual machine. It enables the user to run a piece of logic inside the kernel without modifying the kernel source code or loading a kernel module [9]. Figure 2 shows the entire progress to run eBPF in three steps. 1) The user compiles the source code to an eBPF program in eBPF bytecode and loads the program via a specific syscall. 2) The in-kernel static verifier checks the program to ensure safety. 3) The just-in-time compiler (JITer) [8] translates the eBPF program to executable binary in native hardware for later execution. eBPF is used in a wide variety of domains such as network filtering, tracing, and profiling [9].

eBPF receives greater attention from in-storage computing researches recently [27, 29–34]. It becomes the preference for constructing the ISC runtime, because eBPF provides a hardware-independent bytecode format. The source code can be compiled to an eBPF program only once but runs on general CPUs (e.g. x86, ARM) and specific hardware (e.g. FPGA [35], ASIC [36]). It enables inserting user-defined logic to the devices [31, 35, 37] apart from the kernel.

However, we find that eBPF is inapplicable to general ISC because of its overstrict static verifier. eBPF lacks two critical features, pointer access and dynamic-length loop, which are widely employed in data processing code. We explain limitations with an example. `compute` function in Listing 1 is the typical computation code to sum values, but fails to pass the eBPF static verifier for two reasons. 1) Pointer access. The eBPF verifier checks that the program does not access arbitrary kernel addresses. As `input` and `output` are memory pointers, the verifier does not know their boundaries and prohibits pointer arithmetic and dereference. 2) Dynamic-length loop. eBPF checks each loop by simulating the iteration of the loop during the static verification. It supports a bounded loop [38] when the loop boundary is bounded so that the loop finishes in bounded time in the simulation. However, `length_i` is unknown during the static verification and is dynamically determined before execution. In the verifier's view, the loop is unbounded and the program does not complete in limited time. Therefore, it rejects the program.

We notice that XDP [37], a programmable network packet processing framework based on eBPF, proposed an explicit checking mechanism for pointer access. With the network packet content in the memory buffer [`data, data_end`), before dereferencing a pointer `data+offset`, the program is required to explicitly check `data+offset` < `data_end` in the code. However, XDP requires `offset` to be a known constant. So this explicit checking mechanism does not work for general ISC, where `offset` can be a variable.

In a nutshell, eBPF needs extension to embrace ISC without compromising safety.

3 Design

This section presents the design of λ-IO. We begin with an overview and describe a range of key techniques in detail.

Listing 1: Sample Pseudo-Code of Summing a File

```c
// sum.c
ssize_t compute(void *output, void *input, size_t length_i) {
    int sum = 0;
    for (int i = 0; i < length_i / sizeof(int); i++) {
        sum += ((int *)input)[i];
    }
    *output = sum;
    return sizeof(int); // return the output size
}

// main.c
int vanilla_io_sum(int fd, int file_size) {
    char buf[BUF_SIZE];
    int sum = 0, sum_s;
    for (int i = 0; i < file_size; i += BUF_SIZE) {
        pread(fd, buf, BUF_SIZE, i);
        compute(&sum_s, buf, BUF_SIZE);
        sum += sum_s;
    }
    return sum;
}

int lambda_io_sum(int fd, int file_size) {
    int λ_id = load_λ("sum.sbpf");
    char buf[sizeof(int)];
    int sum = 0;
    for (int i = 0; i < file_size; i += BUF_SIZE) {
        pread_λ(fd, buf,  BUF_SIZE, i, λ_id);
        sum += *(int *)buf;
    }
    return sum;
}

int main() {
    int fd = open("path/to/file");
    vanilla_io_sum(fd, FILE_SIZE);
    lambda_io_sum(fd, FILE_SIZE);
    close(fd);
    return 0;
}
```

3.1 Overall Architecture

Figure 3 sketches the overall architecture of λ-IO. λ-IO extends the vanilla IO stack with λ extension to support offloading computation in the host kernel and the device. λ extension consists of three parts, λ-IO APIs, λ runtime (λ-kernel runtime and λ-device runtime), and request dispatcher.

λ-IO APIs (§3.2) introduce additional programming interfaces for applications. Besides normal IOs to a file, an application can submit λ requests to customize the computational logic (λ function), load the λ function to λ-IO, and invoke the λ function during reading and writing a file.

λ runtime (§3.3) crosses the host-and-device boundary with two counterparts, λ-kernel runtime and λ-device runtime. They provide identical interfaces of computation and data to execute a λ request. λ-IO extends eBPF to sBPF to support ISC, so that the λ function needs to be compiled and loaded only once, but runs in both sides.

Request dispatcher (§3.4) aims to designate λ requests to the λ-kernel runtime or the λ-device runtime effectively and efficiently. λ-IO profiles and monitors the status of the host and the device periodically. It estimates the request execution time of a λ request using our proposed model, and dispatches

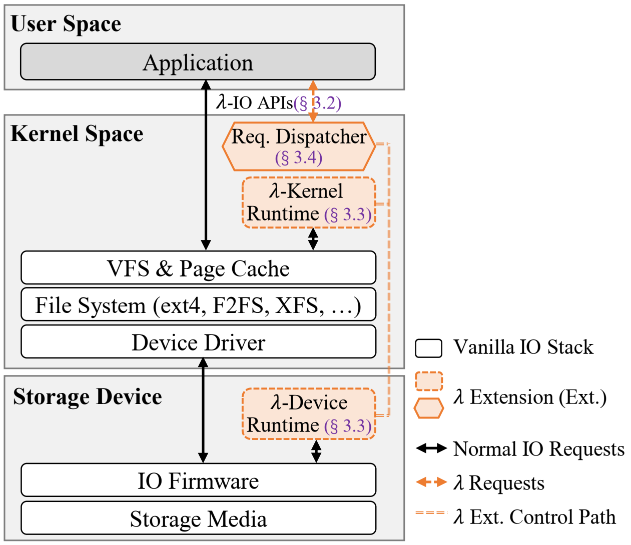


Figure 3: Overall Architecture of λ-IO.

Op	Interface
λ	`ssize_t compute(void *output, void *input, size_t length_i)`
load	`λ_id = load_λ(λ_path)`
open	`fd = open(file_path)`
close	`close(fd)`

read	`pread(fd, buf, length, offset)`
	`pread_λ(fd, buf, length, offset, λ_id)`

write	`pwrite(fd, buf, length, offset)`
	`pwrite_λ(fd, buf, length, offset, λ_id)`

* We omit some well-known types of parameters and return values.

Table 1: λ-IO APIs.

the request to the faster side.

3.2 λ-IO APIs and Workflow

λ-IO inherits the vanilla IO fundamental interfaces to open, close, read, and write a file, as listed in Table 1. Additionally, λ-IO introduces λ *load*, *read* and *write* interfaces, for an application to submit a λ request to offload computation during data transfer.

λ. The first line of Table 1 shows the interface to program the computational logic. Parameters of `input` and `output` point to input and output buffers, along with `length_i` to indicate the size of the input buffer. Thus, the computational logic in the function body can access data through pointers as it does in normal memory computing. It consumes data from the input buffer and produces data to the output buffer.

It is notable that the λ function body need not worry about the specific value of the two pointers, no matter the computational logic runs in the λ-kernel or the λ-device runtime. The λ runtime prepares memory buffers and sets proper values before executing the λ function.

Load. `load_λ` is the interface to load a λ function. The user compiles the λ function source code to an sBPF program, and loads it via `load_λ`. `load_λ` returns `λ_id` as the handle to be used in later λ read and write calls.

The user needs to compile and load a λ function only once, although λ-IO has two runtimes in the host kernel and the device. Receiving a loading call, λ-IO parses the sBPF program file, and transfers the bytecode to both the λ-kernel runtime and the λ-device runtime. Afterward, each runtime invokes the sBPF verifier and JITer to translate the bytecode to an executable binary in the native ISA for later execution.

Open and close. Two APIs remain intact. All λ extension APIs accept the identical `fd` as vanilla ones. Thus, the application can use normal and λ extension APIs simultaneously.

Read. Compared to normal read, λ read (`pread_λ`) adds a parameter `λ_id` to indicate the invoked λ funtion. λ-IO loads file data of the specified range as the `input`, performs computation of `λ_id`, and finally copies the `output` to the application-allocated buffer (`buf`).

To provide better intuition, we walk through how `pread_λ` works with an example of summing values in a file (Listing 1). `vanilla_io_sum` finishes the job via vanilla IO. The application repeatedly reads file data into a buffer (`buf`). Then it runs the computational logic (`compute`) to sum values. `lambda_io_sum` shows how to program the same logic using λ read (`pread_λ`). It has almost the same skeleton as `vanilla_io_sum`. The application loads the compiled sBPF program in Line 24, and gets a handle `λ_id`. When the application calls `pread_λ`, it additionally passes `λ_id`.

The λ runtime performs `pread_λ` in four steps. ❶ It loads the file data of the specified range into an input memory buffer, and sets the `input` and `length_i` parameters of the λ function. ❷ It allocates an output memory buffer and sets it as `output` of the λ function. The output memory buffer is as large as the input by default. ❸ It triggers the λ function, which sums data in the `input` and stores the result in the `output`. ❹ It copies data in `output` to the user-allocated `buf`, along with the output size represented by the return value of `compute`. In this way, the application only needs to allocate a buffer to receive the execution result, instead of the file data.

Write. λ write (`pwrite_λ`) works similarly to λ read in the reversed direction. λ-IO uses data in `buf` as the `input`, runs the function of `λ_id`, writes the `output` to the file at the `offset` and returns the output size.

The λ runtime performs `pwrite_λ` in four steps. ❶ It copies data of `buf` to a memory buffer in λ extension runtime, sets it as the `input` of λ. ❷ It allocates an output memory buffer and sets it as `output` of λ. ❸ It triggers the λ function, and stores the output data to the file from the position of `offset`. ❹ It returns the output size to the application.

3.3 Cross-Platform λ Runtime

The λ runtime plays a central role in executing λ requests of `load_λ`, `pread_λ`, and `pwrite_λ`. To execute λ requests in both the host kernel and the device, the λ runtime crosses the boundary of the host and the device with two instances, λ-kernel runtime and λ-device runtime.

To achieve the cross-platform runtime design, we identify two critical aspects of challenges, computation and data. For computation, both λ-kernel and λ-device runtimes have to store and run the same λ functions, although they are on top of different computation hardware platforms. For data, λ functions should be able to access consistent file data in λ-kernel and λ-device runtimes, along with other userspace applications. We describe how to tackle two challenges separately.

3.3.1 Computation: Extending eBPF to sBPF

At the core of the λ runtime is sBPF. As we state in §2.3, eBPF faces two critical limitations, pointer access and dynamic-length loop. We aim to tackle limitations efficiently. Our ***key idea*** is to bring in dynamic verifications with extra information from λ-IO, so as to check the pointer access and loop during running. sBPF inherits the bytecode format of eBPF, but extends the verifier and JITer.

For the pointer access, sBPF focuses on two memory buffer pointers, `input` and `output`. The sBPF verifier tracks all the pointer variables derived from `input` and `output`, by adding or subtracting an offset. For each dereference of such pointers (e.g. line 5 in Listing 1), the sBPF JITer inserts additional native code for checking the pointer during running. sBPF dynamically checks whether the dereferenced pointer falls within the given region $[\texttt{input}, \texttt{input} + \texttt{length\_i})$ or $[\texttt{output}, \texttt{output} + \texttt{length\_i})$, and returns an error when encountering an out-of-boundary pointer dereference. Afterward, the λ runtime terminates the execution of the `pread_λ` or `pwrite_λ` request with an error code to be handled by the application. In this way, sBPF ensures that pointer accesses to the input and output buffers are valid.

One may notice that `length_i` is passed by the application. For safety, λ-IO checks whether `length_i` is valid through kernel safety verifications (e.g. by `rw_verify_area`), before passing `length_i` to the λ function.

λ-IO does not introduce eBPF helper functions such as `bpf_probe_read` [39] to check pointer access. The root cause is that each eBPF helper function call is translated to a function call in the native binary by the JITer. If a program accesses `input` and `output` buffers via helper functions, the calls incur heavy overhead.

Note that sBPF only handles two pointers of `input` and `output` specially, as they are allocated and managed by λ runtime. Other pointers are still verified by the static verifier.

For the dynamic-length loop, sBPF applies a dynamic count. We observe that, the program has at least a jump-back instruction with a negative offset to implement a loop. Therefore, sBPF limits the number of executing jump-back instructions. The sBPF verifier allows loops during the static verification. The sBPF JITer allocates a counter and inserts extra native code beside each jump-back instruction. Once a jump-back instruction is executed, the counter increases. If the counter reaches the preset loop threshold, the program terminates and returns an error. As we limit the number of jump-backs, the program completes in bounded time.

We further describe how to choose the loop threshold value. For an ISC program, the number of loops is typically proportional to the input buffer size. In practice, the threshold can be set to the same order of magnitude as the maximum input buffer size of programs. In this way, λ-IO permits normal programs and aborts buggy or malicious programs.

Safety. sBPF changes the logic of pointer access and dynamic-length loop, but does not introduce extra safety risks compared to eBPF. In eBPF, the Linux kernel and the eBPF toolchain (e.g., verifier, and JITer) are trusted, while the user-written source code is untrusted. In sBPF, both the kernel and the eBPF toolchain are extended so that the source code running with sBPF can behave differently in two aspects:

1) Pointer accesses to the `input`/`output` buffer. The λ runtime allocates the input/output buffer and checks that the user has permission to access the file range (λ-IO strictly follows the ACL checking of the file system when accessing file data). Once a λ request finishes, the input/output buffer is reclaimed as well. Except for the input/output buffer, pointer accesses to other memory space are verified by eBPF's default static verifier. As a result, sBPF does not introduce extra risks of memory leak compared to eBPF.

2) Dynamic-length loops. sBPF does not allow infinite loops during execution; instead, sBPF allows loops to pass the verifier's checking, but lets the program abort if a loop repeats too many times.

3.3.2 Data: Consistent File Access

As we state in §3.2, the λ function accesses file data via the `input` pointer in `pread_λ` while via the `output` pointer in `pwrite_λ`. Given that there are a sea of file systems, such as ext4, F2FS and XFS, we introduce how λ-IO access file data consistently without relying on any specified file systems for compatibility. Our ***key idea*** is to leverage generic interfaces and components, e.g. existing syscalls, VFS, and page cache in the kernel, and to let the host control consistency.

λ-kernel runtime. Considering compatibility to a host of underlying file systems, we place the λ-kernel runtime atop VFS and page cache, as shown in Figure 3. Regardless of the specific file system, VFS provides the unified file abstraction and access interfaces. For a λ request, the λ-kernel runtime accesses file via `kernel_read` and `kernel_write`.

To optimize read performance, we propose kernel-version mmap, `kernel_mmap` to avoid the heavy data copy during `kernel_read`. λ-IO maps the pages located in the requested range to a kernel virtual address region via `vm_map_ram`. Afterward, λ-IO passes the mapped virtual address as the `input` pointer, so that the λ function can access the file directly.

λ-device runtime. Unlike the host end, the device is agnostic to the host-side file semantics. The device first has to know the exact storage locations of a range in the file. So the host is responsible for extracting necessary metadata and pushing it down to the device. Fortunately, Linux offers `FIEMAP` and `FIBMAP` ioctl interfaces [40, 41], to retrieve the file extent metadata of storage locations by given offset and length.

For λ read, λ-IO gets extent metadata of the specified range inside the file, and pushes the extent down to the device. The λ-device runtime loads the data from given storage locations to a continuous device memory buffer, and passes the buffer address to the λ function running in the device.

λ-IO does the similar for λ write inside the allocated scope of a file. In the case that a λ write appends the output file, λ-IO preallocates device storage space by `fallocate` [42] and then retrieves the file extent metadata of storage locations. Additionally, λ-IO resets the `unwritten` flag as previous works do [1, 25, 43]. In this way, the host can read data written by the λ write on the device, instead of zeros.

Consistency. As data may be modified in the host userspace, the host kernel, and the device, λ-IO has to guarantee data consistency among three places.

One is the host-side consistency between the userspace and the kernel. Both the userspace and the kernel rely on the identical VFS and page cache to access file data. As the λ-kernel runtime reads and writes file through `kernel_read` and `kernel_write`, these calls go through the same path as userspace calls. In this way, consistency is guaranteed by the VFS, page cache, and underlying file systems.

The other is the consistency between the host and the device. λ-IO maintains the consistency in the host. During executing a λ read/write request in the device, λ-IO acquires a read/write lock on the file, to guarantee consistency against other normal IO and λ requests. Additionally, λ-IO manipulates the host-side page cache before dispatching λ requests to the λ-device runtime. Before dispatching a λ read request to the device, λ-IO flushes dirty cache within the overlapped range, so that the device can view the up-to-date version. Before dispatching a λ write request to the device, λ-IO invalidates the host-side page cache within the overlapped range, so that the host can retrieve new data written on the device.

We propose λ-kernel not for performance gaining over vanilla IO, but for two strengths. 1) λ-kernel, together with λ-device, offers unified interfaces and runtime across the host and the device. On basis of this, the computational logic can be run and dispatched on both sides. 2) With the host-side components in the kernel, λ-IO can better exploit the capabilities of the vanilla Linux IO stack, e.g. compatibility, functionality, and sharing, as we mention in §2.1. In this way, λ-IO is friendly to programmers, since a sea of applications use POSIX file interfaces of the vanilla IO stack.

3.4 Dynamic Request Dispatching

λ-IO executes λ requests in both sides, but a request can be faster in either side, as we state in §2.2. The design goal of the request dispatcher is to designate λ requests to the faster side effectively and efficiently.

For effectiveness, we model the execution time of a λ request in both sides, so that the dispatcher can predict the execution time and find the faster side. For efficiency, the requester dispatcher should introduce low extra overhead to the execution. To this end, λ-IO periodically profiles partial requests to determine variables in the model, rather than profiles each λ request. We describe them in detail below.

3.4.1 Modeling Execution Time

We first consider λ read. We start by defining a few symbols, the loaded data size from the storage media D, the bandwidth between the storage media and the device controller buffer B_s for a request, the bandwidth between the device and the host B_d for a request, and the host-side computing bandwidth B_h for a request.

We notice that multiple requests may be submitted and executed concurrently. Therefore, these bandwidth variables correspond to a request, instead of all the concurrent requests. When multiple requests are executed concurrently, the data transfer and computation bandwidth of each request is affected by other requests and thus less than the aggregated.

The execution time in the host t_h is composed of transferring data from the storage media to the device controller, then to the host, and the host computing, as shown in Equation 1.

If the computation happens in the device, the transferred data size between the host and the device becomes αD, where α is the ratio between the output and input. The computing bandwidth in the device is βB_h, where β is the ratio between the device and the host computing bandwidth. We have the execution time t_d in Equation 1.

$$\begin{cases} t_h = \frac{D}{B_s} + \frac{D}{B_d} + \frac{D}{B_h} & \textit{if in the host} \\ t_d = \frac{D}{B_s} + \frac{D}{\beta B_h} + \frac{\alpha D}{B_d} & \textit{if in the device} \end{cases} \tag{1}$$

Similarly, given the input data size D, we have the following to calculate the execution time for a λ write

$$\begin{cases} t_h = \frac{D}{B_h} + \frac{\alpha D}{B_d} + \frac{\alpha D}{B_s} & \textit{if in the host} \\ t_d = \frac{D}{B_d} + \frac{D}{\beta B_h} + \frac{\alpha D}{B_s} & \textit{if in the device} \end{cases} \tag{2}$$

We further discuss the impact of the host-side cache on λ read. The cache reduces data transfer from the device and saves significant data movement overhead. Given the cache ratio c, Equation 1 becomes

$$\begin{cases} t_h = \frac{(1-c)D}{B_s} + \frac{(1-c)D}{B_d} + \frac{D}{B_h} & \textit{if in the host} \\ t_d = \frac{D}{B_s} + \frac{\alpha D}{B_d} + \frac{D}{\beta B_h} & \textit{if in the device} \end{cases} \tag{3}$$

The higher the ratio of the cached file data, the more we tend to dispatch the λ read request to the host.

3.4.2 Periodical Profiling and Dispatching

To estimate the execution time using the above equations, λ-IO has to determine variable values in the equations. We divide seven variables into two groups, one's exact values obtained directly and one estimated by profiling.

The first group includes the input data size D and the host-side cache ratio c. D is the `length_i` in Table 1. To get c, the request dispatcher walks the page cache tree to count the number of cached pages n_c within the given range. Afterward, it calculates c by $n_c \times PAGE\_SIZE /$ `length_i`.

The second group includes the remaining five variables, B_s, B_d, B_h, α, and β. We call them profiling variables. λ-IO does not know their exact values for a λ request in advance, so it estimates their values through periodical profiling. It is notable that B_s is the bandwidth for a request, but is not necessarily equal to the physical bandwidth between the storage media and the device controller buffer. B_s changes over time, like when multiple requests share the device, and so are the other variables. Thus we call these variables in the second group profiling variables and describe how to profile them.

We notice that profiling variables vary on different files and λ functions. Therefore, λ-IO profiles their values for each (`file_path`, `λ_id`) pair separately. The dispatching supports collocated applications as (`file_path`, `λ_id`) pairs may come from different applications running simultaneously.

To determine these variables efficiently, λ-IO profiles partial requests periodically rather than profiles each request. For a given (`file_path`, `λ_id`) pair, λ-IO sets a *profiling period*, like n requests. For the beginning k requests in a period (we refer to k as *profiling length*), λ-IO submits them to both the λ-kernel and λ-device runtimes. Each runtime measures the values of profiling variables of a request during execution. After k requests complete, λ-IO calculates the average of each variable and uses it as the estimated value.

For the following requests in the same period, λ-IO calculates the estimated execution time in both sides using the equations and estimated variable values. Then it dispatches the request to the faster side. When a new period comes, λ-IO repeats the profiling to estimate new values for the variables.

To achieve a balance between effectiveness and efficiency, λ-IO should be careful to set values of two profiling parameters, i.e. profiling period and profiling length. With a smaller profiling period and a bigger profiling length, the dispatcher can keep track of the status and find the faster side more effectively, but will induce higher overhead. In contrast, a bigger profiling period and a smaller profiling length lead to lower overhead, but the dispatcher may miss real-time changes in the status and make suboptimal decisions.

4 Implementation

We implement λ-IO in the full-stack software and hardware environment. We modify the Linux kernel to integrate λ extension in the host and build an NVMe device with λ extension support on a real hardware platform. We introduce the implementation details in this section.

Host. We implement the request dispatcher and the λ-kernel runtime in the host kernel with a kernel module. The kernel module creates a `procfs` [44] file to receive λ extension calls. We modify the eBPF verifier and the x86 JITer.

Device. We implement the device-side components on a real hardware platform, Daisy OpenSSD [45]. It is representative of computational storage devices, as one of the latest iterations of the OpenSSD project [46, 47]. OpenSSD is widely recognized and used by ISC researches from both industry and academia [18–21, 27, 48, 49]. The device connects to the host via PCIe and operates as an NVMe drive. We migrate OpenExpress [50, 51], an open-source NVMe controller, to the platform. We refactor the NVMe firmware thoroughly, modify the eBPF verifier and the ARM eBPF JITer from the kernel, and construct the λ-device runtime atop them.

Communication. We use standard NVMe over PCIe to communicate between the host and the device. λ-IO delivers normal IO requests through existing NVM IO commands, and customizes three *Vendor Specific Commands* [52, 53] for λ load, read, and write. λ-IO creates one standalone NVMe command for one λ request separately, delivering all necessary file extent metadata and indicating the λ function it triggers.

5 Evaluation

We evaluate λ-IO to answer the following questions:

- How does λ-IO perform on typical in-storage computing applications compared to existing approaches? (§5.2)
- How does λ-IO perform when collocated applications run concurrently? (§5.3)
- How do workload characteristics and dispatching configurations affect the performance? (§5.4)
- How is the sBPF overhead compared to eBPF? (§5.5)
- Can the end-to-end performance of a real application benefit from λ-IO? (§5.6)

5.1 Experimental Setup

Testbed. Table 2 shows our detailed testbed configurations. We implement λ-IO on the host and the device as stated in §4. The host CPU has 4 physical cores and 8 hyperthreads. The device SoC has 4 ARM cores. The 2GB SoC memory acts as the device controller memory. We equip the device with two 32GB DRAM DIMMs to act as the backend storage media, as the hardware board manufacturer provides no NAND flash modules. The data transfer bandwidth between the storage media and the device controller memory is 3.52GB/s. The bandwidth between the device controller memory and the host is 3.22GB/s. The bandwidth for ARM cores to access the device controller memory is 5.09GB/s. The performance is comparable to real SSDs and hardware platforms in recent works [19, 24, 48, 54].

Host	CPU	Intel Core i7-7700 @3.6GHz (4C8T)
	Memory	16GB
	OS	Ubuntu 20.04.2 LTS
	Kernel	Linux 5.10.21
Device	SoC	Xilinx Zynq Ultrascale+ ZU17EG
	Memory	2GB
	Storage	64GB

Table 2: Testbed Configurations.

App.	Description	ISC LoC	
		λ-IO	INSIDER [25]
Stats64	Read the file data as 64-bit integers and calculate the sum, maximum, and minimum.	30	240
Stats32	Read the file data as 32-bit integers and calculate the sum, maximum, and minimum.	30	240
KNN	Read vectors in the file and calculate distances between a target vector.	32	99
Grep	Read rows in the file and match a target string.	35	254
Bitmap	Decompress a bitmap and write to the file.	20	188

Table 3: Information of Synthetic Applications.

Workloads. We choose five synthetic applications Stats64, Stats32, KNN, Grep, and Bitmap with their information listed in Table 3. The first four applications read data from the source file and process it. The last application, Bitmap, decompresses a bitmap passed by `buf` and writes the output to the file. They fit the in-storage computing scenario well and are widely studied and evaluated in previous works [4, 13, 14, 24, 25, 27, 55]. The program code follows the skeleton of Listing 1 and costs little extra overhead to implement.

The input dataset file of four read applications is 16GB in size. The output file of Bitmap is also 16GB in size. As the output data also requires persistence, we synchronize the file after writing. We format the storage device to ext4 to store files. Applications run with 8 host threads and read/write file data to an 8MB buffer in each iteration. We set loop threshold (§3.3) to 16 million, profiling period (§3.4) n to 200, and profiling length k to 5. We keep the above configurations throughout experiments unless otherwise specified.

Porting overhead. As shown in Table 3, the LoC of implementing the same offloaded computational logic for λ-IO is significantly smaller than INSIDER [25]. We count the ISC LoC of INSIDER from its code repository [56]. The porting overhead of INSIDER is large, like other FPGA-based ISC frameworks. This is because INSIDER leverages HLS [57] to implement the computational logic on FPGA. Although HLS reduces the programming overhead, it still requires the programmer to be an expert in FPGA details [58] and write much hardware-specific code to optimize the performance.

Comparing targets. We adopt six IO modes in our evaluation, three vinalla IO modes and three λ-IO modes.

- *(**B**) Buffer IO*: uses the default IO mode of vanilla IO `pread/pwrite` to access file data. It enables the page cache. The computational logic is directly compiled to the native ISA, and so are Direct IO and Mmap.
- *(**I**) Direct IO*: similar to Buffer IO but opens the dataset file with `O_DIRECT`. It bypasses the kernel page cache.
- *(**M**) Mmap*: maps the dataset file to the userspace virtual address of the process. It eliminates data copy between the kernel and the userspace.
- *(**K**) λ-IO Kernel*: integrates the computational logic using λ-IO APIs and executes all the requests in the kernel runtime.
- *(**D**) λ-IO Device*: integrates the computational logic using λ-IO APIs and executes all the requests in the device runtime.
- *(**λ**) λ-IO*: integrates the computational logic using λ-IO APIs and leaves the λ-IO request scheduler to dispatch between the kernel and the device dynamically.

The first three IO modes (***B***, ***I***, ***M***) correspond to common approaches that use vanilla IO. (***D***) λ-IO Device corresponds to existing device-only ISC frameworks, e.g. Biscuit, INSIDER, MetalFS, BlockNDP [25–27, 55].

Warmup. We run experiments in two caching settings to examine the performance of λ-IO, 1) without warmup: drop the page cache before each execution, 2) with warmup: read the input/output file sequentially via Buffer IO, to warm up the page cache before each execution.

With warmup, we simulate the host-side cache state in practice and evaluate the performance of λ-IO in this scenario. After the warmup, partial data resides in the page cache while the rest is in the device if the input/output file is larger than the page cache. In real-time data analysis systems such as OLAP databases, the application continues producing data and storing it. Thus, the host-side cache always stores the latest part of the data. Meanwhile, the analytical application runs periodically to process the data in the recent period, where partial data resides in the cache while others have been flushed to the device.

5.2 Single Application

We compare the execution time of applications running singly in six IO modes. Figure 4 reports the results.

For further analysis, we break down execution time to three parts, IO, computation, and other. ❶ IO time means the time of reading/writing the file. In (B) Buffer IO and (I) Direct IO, it is the time of `pread/pwrite`. In (M) Mmap, the read time is included in computation, because the application retrieves data implicitly via page faults. In (K) λ-IO Kernel, it is the time of reading/writing data via the VFS. In (D) λ-IO Device, it is the time of transferring data between the storage media and the device controller memory. In (λ) λ-IO, it is the sum of IO time of requests, whether they execute in the host or the device. ❷ Computation time contains the calculation part of the application, along with memory access time. ❸ Other time is the remaining part excluding IO and computation.

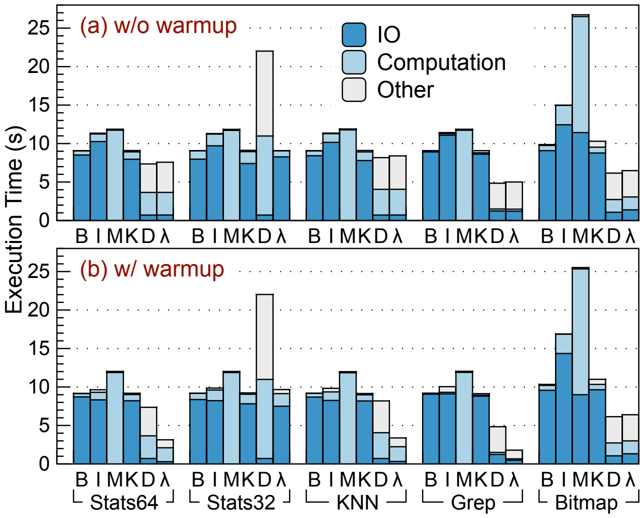


Figure 4: Performance of Applications Running Alone. ***B**: Buffer IO, **I**: Direct IO, **M**: Mmap, **K**: λ-IO Kernel, **D**: λ-IO Device, **λ**: λ-IO.*

1) λ-IO Device vs. vanilla IO. We choose Buffer IO to compare with λ-IO Device first, as it is the default operation mode of vanilla IO. We analyze the performance without warmup. In-storage computing improves the overall performance of Stats64, KNN, Grep, and Bitmap by 23.24%, 10.82%, 87.13%, and 60.15% against Buffer IO respectively. This is because applications running in the host are bounded by IO, which takes up more than 92.04% of execution time. For a normal read, data is transferred from the storage media to the device controller memory and then to the host software stack, as a normal SSD does. In contrast, IO time in λ-IO Device occupies less than 25.29%, because loading data inside the device avoids transferring to the host and going through the software stack. After the data is read to the device controller memory, device processors enjoy higher internal bandwidth to access data. Moreover, the computational logic outputs a smaller amount of data than the input file data, reducing the data transfer overhead to the host. Write (Bitmap) works similarly, except for the reversed direction.

The performance with warmup is close to that without warmup. This is because the host memory is 16GB in size, and the page cache capacity is just smaller than the dataset. After the warmup, all the input file is cached inside the page cache, except the beginning part. As the application runs again, it still accesses data from the very beginning. The beginning part misses in the page cache and evicts other pages, which leads to more and more cache misses. Finally, the page cache fails to buffer any data in fact.

One might note that *other* time of λ-IO Device is relatively large, as illustrated in Figure 4. This is because the host issues requests with 8 threads, while the device only has 4 threads to process. So a request has to pend in the queue and wait for the completion of previous requests. But request pending and waiting actually exist in all modes. For example in Buffer IO, in the progress of `pread`/`pwrite`, IO requests also pend and wait in queues throughout the IO stack. The overhead is included in the IO time.

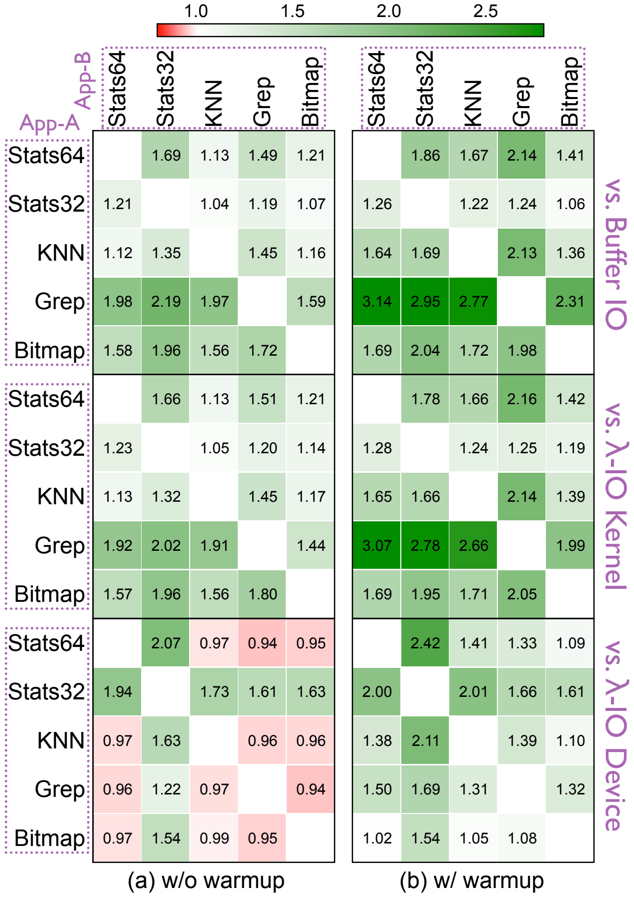
λ-IO Device does not always outperform vanilla IO, e.g. on Stats32. λ-IO Device consumes 6.65× computation time against λ-IO Kernel on Stats32. We examine the generated native machine code of Stats32 and find that eBPF does not support 32-bit integers well. Specifically, eBPF programs use 64-bit registers even for 32-bit integers in Stats32. It introduces a pair of left and right shift instructions to clear the upper 32 bits of registers. This mechanism induces significant overhead in the device processors. We leave optimization of supporting 32-bit integers for future work.

2) λ-IO Kernel vs. vanilla IO. λ-IO Kernel executes the computational logic in the host kernel and exhibits similar performance against Buffer IO. We look deeper into the time breakdown of λ-IO Kernel. It takes slightly more time to compute, since sBPF incurs overhead compared to the native ISA. Oppositely, λ-IO Kernel reduces data copy from the kernel page cache to the userspace buffer. These two aspects together result in similar performance to Buffer IO.

We further examine three vanilla IO modes. As shown in Figure 4, Buffer IO performs the best. Direct IO is slower, as it bypasses the page cache and therefore misses the opportunity of readahead. We note that computation takes up almost all the execution time in Mmap on read applications. It is because the computational logic accepts the mmaped virtual memory address and loads data from the file system via user-agnostic page faults. Page fault handling in the kernel induces high overhead [59], so leads Mmap to the slowest mode. Different from read applications, Bitmap still spends time on IO, because it writes the mapped data via `msync`.

3) λ-IO modes. The rightmost bars in Figure 4 present the performance of λ-IO modes. Without warmup, the execution time of λ-IO is almost the minimum value of λ-IO Kernel and λ-IO Device on all applications, although the dispatcher introduces less than 4.98% overhead.

With warmup, λ-IO improves more significantly. Stats64, KNN, and Grep are 2.19×, 2.71×, 5.12× faster than Buffer IO. λ-IO is faster than both λ-IO Kernel and λ-IO Device. When the application accesses the beginning part of the file, the λ-IO request dispatcher finds that the device processes faster. It thus submits these requests to the device and keeps the page cache untouched. As the beginning part passes, the rest file data is buffered in the page cache. Then the dispatcher detects the kernel is better and submits requests. In this way, λ-IO takes full advantage of both the kernel and the device, achieving the best performance. The write application, Bitmap, does not gain extra benefits from warmup, compared to the performance without warmup. As we need to synchronize the data to the device after writing, it always finishes faster in the device side.



vs.	App-A	(a) Stats64	(a) Stats32	(a) KNN	(a) Grep	(a) Bitmap	(b) Stats64	(b) Stats32	(b) KNN	(b) Grep	(b) Bitmap
Buffer IO	Stats64		1.69	1.13	1.49	1.21		1.86	1.67	2.14	1.41
	Stats32	1.21		1.04	1.19	1.07	1.26		1.22	1.24	1.06
	KNN	1.12	1.35		1.45	1.16	1.64	1.69		2.13	1.36
	Grep	1.98	2.19	1.97		1.59	3.14	2.95	2.77		2.31
	Bitmap	1.58	1.96	1.56	1.72		1.69	2.04	1.72	1.98	
λ-IO Kernel	Stats64		1.66	1.13	1.51	1.21		1.78	1.66	2.16	1.42
	Stats32	1.23		1.05	1.20	1.14	1.28		1.24	1.25	1.19
	KNN	1.13	1.32		1.45	1.17	1.65	1.66		2.14	1.39
	Grep	1.92	2.02	1.91		1.44	3.07	2.78	2.66		1.99
	Bitmap	1.57	1.96	1.56	1.80		1.69	1.95	1.71	2.05	
λ-IO Device	Stats64		2.07	0.97	0.94	0.95		2.42	1.41	1.33	1.09
	Stats32	1.94		1.73	1.61	1.63	2.00		2.01	1.66	1.61
	KNN	0.97	1.63		0.96	0.96	1.38	2.11		1.39	1.10
	Grep	0.96	1.22	0.97		0.94	1.50	1.69	1.31		1.32
	Bitmap	0.97	1.54	0.99	0.95		1.02	1.54	1.05	1.08	

(a) w/o warmup (b) w/ warmup

Figure 5: Speedup of Running Applications Concurrently in λ-IO Mode. *App-A and App-B are two collocated applications. Digits denote the speedup of App-A running in λ-IO against Buffer IO, λ-IO Kernel, and λ-IO Device.*

The results demonstrate that λ-IO dispatches requests effectively and efficiently. Moreover, λ-IO yields better performance than kernel-only and device-only ISC approaches for read applications running with warmup.

5.3 Collocated Applications

We evaluate executing collocated applications concurrently, to present how λ-IO works in this scenario.

With five applications, we have ten combinations of two different applications. For a combination, we run two applications concurrently, each with 4 threads and its own 16GB dataset. We evaluate the combination in Buffer IO, λ-IO Kernel, λ-IO Device, and λ-IO modes. Then we measure execution time of each application and calculate the speedup of each application in the λ-IO mode against other three modes. We draw results of all combinations in Figure 5, where digits mean the speedup of App-A.

For better understanding the figure, we take two digits 1.21 and 1.69 in the top left corner of subfigure (a) as an example. We run Stats32 and Stats64 concurrently in λ-IO mode. Compared to run them concurrently in Buffer IO mode, Stats32 and Stats64 complete 1.21× and 1.69× faster respectively.

As we evaluate in §5.2, Stats64, KNN, Grep, and Bitmap run faster in the device. We classify results of Figure 5(a) into two categories according to whether chosen applications both run faster in the device. 1) When Stats32 is chosen, λ-IO speeds up both applications by up to 2.19× compared to other three modes. This is because λ-IO dispatches Stats32 to the host and the other (Stats64, KNN, Grep, or Bitmap) to the device, so as to exploit both sides. 2) Otherwise when Stats32 is not chosen, λ-IO outperforms Buffer IO and λ-IO Kernel by up to 1.98×, because λ-IO executes them in the faster device side. Compared to λ-IO Device, λ-IO introduces less than 5.64% dispatching overhead. As shown in Figure 5.3(b), the speedup with warmup is higher than that without warmup, because the dispatcher is aware of the page cache.

Since collocated applications run different computational functions (λ) on different dataset files (`file_Path`), λ-IO estimates the execution time of their requests and dispatches separately (§3.4). Based on the design, λ-IO can dispatch collocated applications effectively and efficiently, as demonstrated in this experiment.

5.4 Sensitivity Analysis

5.4.1 Dataset Size

We evaluate the impacts of the dataset size on the performance with warmup. We take Stats64 as an example due to space limitation. Figure 6 reports the results in six modes with varying dataset sizes. We categorize them into three types, less than (8GB), approximately equal to (16GB), and greater than (24GB, 32GB, and 40GB) the page cache capacity.

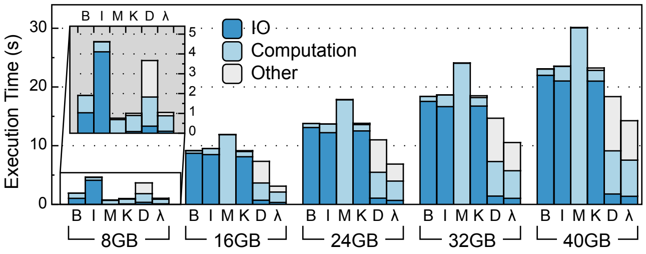


Figure 6: Stats64 Performance with Varying Dataset Sizes.

Dataset size < page cache capacity. Recall that the host memory is 16GB in our testbed, so the 8GB dataset fully resides in the page cache after the warmup run. The host does not have to read the data from the device, getting rid of the peripheral IO bottleneck. Direct IO still bypasses the page cache and loads data from the device, so it is significantly slower. The device has wimpier processors than the host, so λ-IO Device exhibits lower performance. λ-IO performance is close to λ-IO Kernel, which again proves the effectiveness of dynamic dispatching.

Dataset size ≈ page cache capacity. The page cache capacity is just narrower than the host memory 16GB size, as running

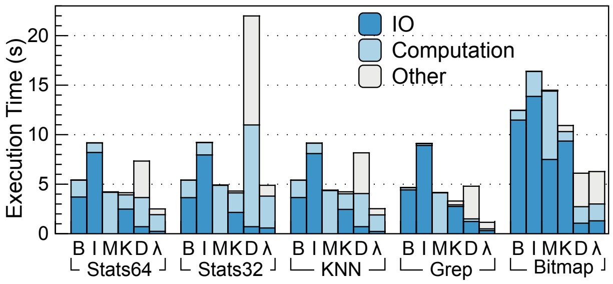


Figure 7: Performance with Random Warmup.

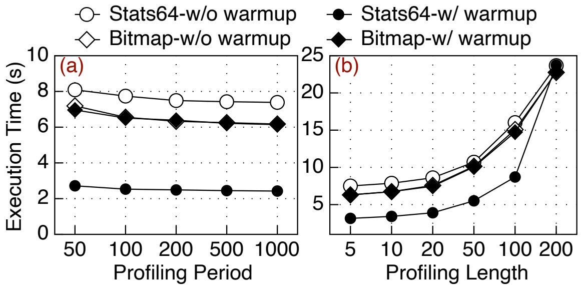


Figure 8: Performance with Varying Profiling Periods and Profiling Lengths.

programs occupy a fraction of space. This setting is the same as §5.2. The host-side page cache takes little effect and λ-IO exploits both the host kernel and the device simultaneously.

Dataset size > page cache capacity. When the dataset size exceeds page cache capacity, a fixed size of data is cached after the warmup. λ-IO performs better than the other five modes by 1.28× to 1.60× because it dispatches requests to both sides efficiently.

5.4.2 Warmup

We evaluate the impacts of the warmup approach on performance. As we state in §5.1, we warm up the page cache by sequentially reading the input/output file via Buffer IO. In this experiment, we change sequential read to random read and show the results in Figure 7.

Compared to results in Figure 4, Buffer IO performs better. After warmup by random read, random parts of the dataset file are in the page cache. During the execution, the application accesses the dataset file sequentially, so it can access partial data quickly when that part of the data is in the page cache. Nevertheless, λ-IO can still outperform Buffer IO by 4.05×. Dynamic request dispatching of λ-IO works effectively and efficiently, no matter how the warmup is taken.

5.4.3 Profiling Period and Profiling Length

Figure 8 depicts the performance with varying profiling periods and lengths. We choose Stats64 and Bitmap as the representative of read and write applications. When the profiling period becomes larger, the execution time decreases as λ-IO profiles fewer requests. The execution time remains stable when the profiling period is over 200, as the profiling and dispatching overhead is small enough. So we choose 200 as the default value of the profiling period because a small period leads to a quick response to the status.

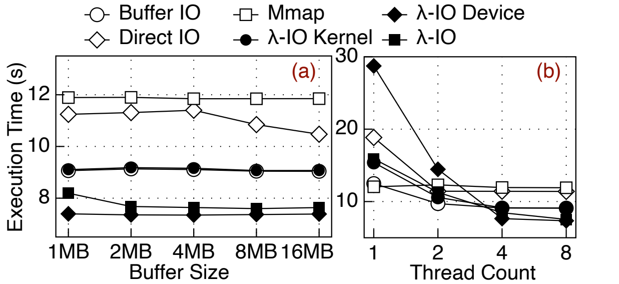


Figure 9: Stats64 Performance with Varying (a) Buffer Sizes and (b) Thread Counts.

When the profiling length goes up, the execution time increases significantly. With a profiling length of 200, λ-IO profiles all the requests and submits them to both the kernel and the device. We choose 5 as the default value of the profiling length. We calculate the average by removing the maximum and minimum values, to avoid accidental measurement errors.

5.4.4 Buffer Size

Figure 9(a) illustrates the performance of the Stats64 application with different data buffer sizes in each iteration. Most IO modes stay stable when the buffer size varies. Direct IO runs faster as the buffer size surpasses 4MB. Buffer IO and λ-IO Kernel perform almost the same, as in previous experiments.

5.4.5 Thread Count

Figure 9(b) plots the execution time of the Stats64 application with different number of host threads. Most IO modes execute faster when the number of threads grows up. Mmap rarely improves as using 1 thread reaches the top performance. The performance of λ-IO Device mode scales linearly from 1 thread to 4 threads. For all IO modes, 4 threads are enough to exploit the performance potential although the host CPU has 8 hyperthreads.

5.5 Overhead of sBPF

In this experiment, we evaluate the overhead of sBPF against eBPF. We choose two representative applications Stats64 and Stats32, because they run faster in the host and the device separately. We compare computation and total execution time in 8 settings, as shown in Table 4.

Settings. Eight settings are divided into two groups, kernel and device. We compile the computational logic to eBPF/sBPF programs. Two eBPF settings in the table mean running the bytecode by the eBPF verifier and JITer. We disable loop and pointer verifications to test the bare performance. DL checks dynamic-length loops on the basis of eBPF. DL+HF checks pointer access by helper functions on top of DL. We add two bpf helper functions for pointer read and write to the input and output buffers. All the input and output buffer

Setting		Stats64		Stats32	
		Compute (s)	Total (s)	Compute (s)	Total (s)
Kernel	eBPF	0.84	9.06	1.38	9.08
	DL	0.86	9.06	1.41	9.16
	DL+HF	3.99	9.47	9.19	14.15
	sBPF	0.99	9.06	1.60	9.12
Device	eBPF	2.39	6.38	8.64	18.75
	DL	2.64	6.84	9.04	19.54
	DL+HF	11.82	25.11	31.99	65.45
	sBPF	2.94	7.34	10.27	22.02

Table 4: Performance on eBPF and sBPF. *DL: eBPF with checking Dynamic-length Loops. DL+HF: eBPF with checking Dynamic-length Loop and checking memory access by Helper Functions.*

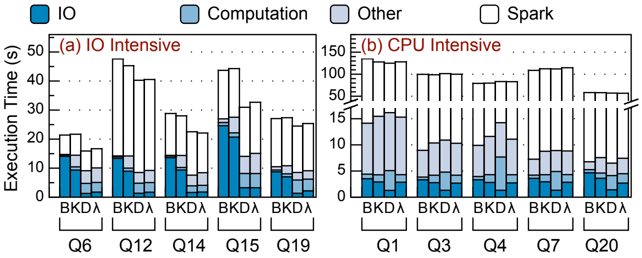


Figure 10: TPC-H Performance w/o Warmup. ***B**: Buffer IO, **K**: λ-IO Kernel, **D**: λ-IO Device, **λ**: λ-IO. **Q**`i`: query `i`.*

pointer accesses in the computational logic are modified to use helper functions. sBPF is the default setting used in all the other experiments. It combines dynamic loop and pointer access as we describe in §3.3.

Results. Table 4 depicts the results in 8 settings. Comparing the results of eBPF and sBPF settings, we find that the loop check induces at most 2.44% and 10.09% overhead in the host kernel and the device. Together with the pointer check, sBPF adds no more than 16.96% and 22.68% computation time in two sides respectively. The total execution time is almost unchanged in the host, and increases by 15.14% - 17.44% in the device. The results of UL+HF settings are notable. It expands computation time by up to 6.67× and 4.93× in two sides and slows down the total execution time seriously. The results demonstrate that loop and pointer checks of sBPF come at an acceptable cost.

5.6 Case Study: Spark SQL

In this experiment, we port a real application, Spark SQL [10], to λ-IO and evaluate its end-to-end performance with TPC-H [11]. Spark SQL is a prevalent SQL application for relational processing on structured data [60], as a module of Apache Spark. Spark SQL follows the schema of reading and processing data, so that it offers the opportunity of offloading computational logic to read via λ-IO. For an SQL query, Spark SQL first parses the query, constructs JAVA source code for processing data, and generates JAVA bytecode for later execution. Afterward, Spark SQL executes in two steps, 1) reads data from files, and 2) executes the JAVA bytecode to process input data and returns results. Thus, we can extract part of processing logic in the JAVA source code and integrate it into the first-step read via λ-IO.

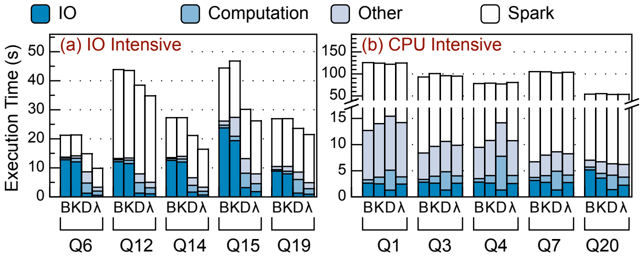


Figure 11: TPC-H Performance w/ Warmup.

Workloads. TPC-H is a widely-used OLAP benchmark [11] that defines a dataset of 8 tables and 22 SQL queries. We generate a 41.6GB dataset with a scale factor of 40. Three largest tables, `LINEITEM`, `ORDERS`, and `PARTSUPP` are 28.61GB (68.77%), 6.58GB (15.84%), and 4.53 GB (10.88%) in size respectively, together taking up 95.49% space of the whole dataset. We equip the host with 32GB of physical memory, which is still smaller than the dataset size. As for queries, we classify them into two categories, IO intensive and CPU intensive. We show 5 queries of each category due to the paper space limitation. They cover various SQL operations of projection, selection, aggregation, join, subquery, etc.

We focus on the largest table `LINEITEM` in this experiment, as all the 10 queries retrieve data from it. We add a preprocessing module between Spark SQL and λ-IO. For a SQL query, we extract filter logic (projection and selection) of `LINEITEM` from generated JAVA source code, convert it to C code, and integrate it to read in the preprocessing module. In this way, the execution progress finishes in two steps. 1) The preprocessing module reads file data and filters. 2) Spark SQL retrieves filtered data from the preprocessing module, further processes (e.g. aggregation, join), and returns results in the Spark environment. The filter logic of every query has less than 50 LoC and thus the porting overhead is low.

Comparing targets. We implement and run the preprocessing module in four modes, Buffer IO (B), λ-IO Kernel (K), λ-IO Device (D), and λ-IO (λ). In the Buffer IO mode, the preprocessing module reads file data via `pread` and runs the filter logic of projection and selection totally in the userspace. In the other three modes, the processing module integrates filter logic into read via `pread_λ` of λ-IO. We do not present the performance of the unmodified Spark SQL in the paper, as it is always slower than our modified Spark SQL with the preprocessing module, and instead, we use the Buffer IO mode as the baseline for fairness.

Results. Figure 10 and Figure 11 report experimental results without and with warmup. We warm up by reading the dataset sequentially before each execution, as we state in §5.1. As we mention in the settings above, the modified Spark SQL executes a SQL query in two steps, first in the preprocessing module and then in the Spark environment. We divide time in the preprocessing module into three parts of IO, computation, and other, as in Figure 4. Plus time in Spark, we break down end-to-end execution time into four parts in the figures.

Figure 10(a) shows the performance of IO intensive queries without warmup. In these queries, IO occupies 27.02% – 60.41% of end-to-end execution time in the Buffer IO mode. The execution time of λ-IO Kernel is similar to Buffer IO, like in previous experiments. λ-IO dispatches requests to the device efficiently and reduces preprocessing time (IO + computation + other) by up to 81.85% against Buffer IO. According to our statistics, the preprocessing module reduces the input data size to only 0.59% – 6.75%. Taking Spark time and scheduling overhead into account, λ-IO outperforms Buffer IO by 8.56% – 31.58%. λ-IO accelerates the end-to-end execution time of Spark SQL on IO intensive queries.

Figure 11(a) reports the performance with warmup. The dataset is larger than the page cache. We focus on the execution time of λ-IO. It performs faster than Buffer IO, λ-IO Kernel, and λ-IO Device by up to 2.15×, 2.16×, and 1.51× respectively. We also evaluate on a 20.8GB dataset (Scale Factor=20). With all data cached in the host, λ-IO Kernel is faster than λ-IO Device by 9.16% – 35.56%. λ-IO dispatches requests to the kernel with less than 2.98% overhead. These demonstrate the effectiveness of λ-IO dispatching.

Figure 10(b) and Figure 11(b) show performance of CPU intensive queries. λ-IO Kernel performs similarly to Buffer IO. Preprocessing time of Q20 in λ-IO Device is 16.28% less than λ-IO Kernel, where the row selectivity 15.1% is as low as IO-intensive queries. But for Q1, Q3, Q4 and Q7, preprocessing in the λ-IO Kernel is faster than λ-IO Device by up to 18.45%. This is because they select 30.3% – 98.5% rows, much higher than IO intensive queries. The device copies and returns more data and becomes less effecient than the host. Even though preprocessing is faster either in the host or the device, λ-IO chooses the faster side for all the 5 queries. The key difference from IO intensive queries is that Spark occupies dominant part, more than 87.48% of execution time. Therefore, the end-to-end execution time of all modes does not differ significantly.

In summary, taking Spark SQL as an example, real applications can benefit from λ-IO.

6 Related Work

In-storage computing (ISC) in the storage device originates in the disk era [4,12] and revives with the advent of SSDs [5,13]. We classify recent works into two categories, case study and general framework. Case study works accelerate a specific application or system by ISC, such as SQL [15,61,62], big data [14], graph [16,17,49], file system [22,29,63], and data training [18,19,64]. General ISC frameworks target offloading user-defined computational logic [23–27,55] and are closer to ours. As we state before, existing ISC frameworks mostly focus on providing manipulation interfaces in the userspace and accelerating computation in the device, but λ-IO redesigns the IO stack to support offloading computation. We discuss two works in more detail. Summarizer [24] proposes an automatic dispatching approach to saturate the device first, but does not consider many factors affecting the execution time in both sides. λ-IO takes many factors into consideration, and proposes profiling-based dynamic dispatching to designate requests. MetalFS [26] integrates into Linux as a file system driver, it only offloads computation to the FPGA device, without utilizing the host computation resources. λ-IO employs dynamic dispatching to exploit both sides.

As the network system continues delving into eBPF [35, 37], it also gains increasing attention in the storage system [29], especially ISC researches [31–34]. ExtFuse [29] accelerates file systems by embedding specialized request handlers into the kernel. Kourtis et al. [31] propose pushing computation to the disaggregated storage device, in order to avoid multiple network roundtrips. As the first one targets a specific acceleration, other ISC works make preliminary exploration and envision of bringing eBPF to in-storage computing. Zhong et al. [30,65] focus on pointer-chasing storage functions, e.g. a chain of IO requests, and aim to resubmit them in a lower layer of the host kernel to alleviate software stack overhead. But they do not pay enough attention to data computation inside one IO or offloading computation to computational storage devices. λ-IO analyzes and identifies two critical limitations that render eBPF inapplicable to general ISC. Responding to this issue, λ-IO proposes sBPF to break the limitations. Moreover, we build λ-IO on the full-stack software and hardware environment, not only the host side.

7 Conclusion

In this paper, we present λ-IO, which extends Linux IO to enable offloading computation to both the host kernel and the device. It carries and executes a user-defined computational logic during data transfer. We implement λ-IO in the full-stack and real software and hardware environment, and evaluate it with synthetic and real applications against vanilla Linux IO, showing significant performance improvement.

Acknowledgments

We sincerely thank our shepherd Alex Conway 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. 61832011, 62022051, & 62202255), and Huawei.

References

[1] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. Bluedbm: An appliance for big data analytics. *SIGARCH Comput. Archit. News*, 43(3S):1–13, jun 2015.

[2] SmartSSD - Samsung Semiconductor. https://samsungsemiconductor-us.com/smartssd/.

[3] Computaional Storage | SNIA. https://www.snia.org/computational.

[4] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91, October 1998.

[5] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 119–132, 2013.

[6] pread(2) — Linux manual page. https://man7.org/linux/man-pages/man2/pread.2.html.

[7] readv(2) — Linux manual page. https://man7.org/linux/man-pages/man2/readv.2.html.

[8] A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[9] eBPF - Introdcution, Tutorials & Community Resources. https://ebpf.io/.

[10] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[11] TPC-H Homepage. https://www.tpc.org/tpch/.

[12] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. Citeseer, 1998.

[13] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 91–102, 2013.

[14] Yangwook Kang, Yang-suk Kee, Ethan L Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. In *2013 IEEE 29th symposium on mass storage systems and technologies (MSST)*, pages 1–12. IEEE, 2013.

[15] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[16] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H Peter Hofstee, Gi-Joon Nam, Mark R Nutter, and Damir Jamsek. Extrav: boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment*, 10(12):1706–1717, 2017.

[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] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, pages 395–410, 2019.

[19] 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*, ASPLOS 2021, page 717–729, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Xiaohao Wang, Yifan Yuan, You Zhou, Chance C. Coats, and Jian Huang. Project almanac: A time-traveling solid-state drive. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. Rm-ssd: In-storage computing for large-scale recommendation inference. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1056–1070, 2022.

[22] Zhe Yang, Youyou Lu, Erci Xu, and Jiwu Shu. Coinpurse: A device-assisted file system with dual interfaces. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[23] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 67–80, USA, 2014. USENIX Association.

[24] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 219–231, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, pages 379–394, 2019.

[26] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with fpgas. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Antonio Barbalace, Martin Decky, Javier Picorel, and Pramod Bhatotia. Blockndp: Block-storage near data processing. In *Proceedings of the 21st International Middleware Conference Industrial Track*, Middleware '20, page 8–15, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[29] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, pages 121–134, 2019.

[30] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. Bpf for storage: an exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 128–135, 2021.

[31] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. Safe and efficient remote application code execution on disaggregated nvm storage with ebpf. *arXiv preprint arXiv:2002.11528*, 2020.

[32] Wenjun Huang and Marcus Paradies. An evaluation of webassembly and ebpf as offloading mechanisms in the context of computational storage. *CoRR*, abs/2111.01947, 2021.

[33] Giulia Frascaria, Animesh Trivedi, and Lin Wang. A case for a programmable edge storage middleware. *CoRR*, abs/2111.14720, 2021.

[34] Corne Lukken, Giulia Frascaria, and Animesh Trivedi. ZCSD: a computational storage device over zoned namespaces (ZNS) ssds. *CoRR*, abs/2112.00142, 2021.

[35] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on FPGA nics. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990, 2020.

[36] eBPF Introduction - Netronome. `https://www.netronome.com/technology/ebpf/`.

[37] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[38] Bounded loops in BPF for the 5.3 kernel. `https://lwn.net/Articles/794934/`.

[39] bpf-helpers Linux manual page. `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html`.

[40] filefrag(8) - linux manual page. `https://man7.org/linux/man-pages/man8/filefrag.8.html`.

[41] Fiemap ioctl. `https://www.kernel.org/doc/Documentation/filesystems/fiemap.txt`.

[42] fallocate(2) - linux manual page. `https://man7.org/linux/man-pages/man2/fallocate.2.html`.

[43] Ian F. Adams, John Keys, and Michael P. Mesnier. Respecting the block interface - computational storage using virtual objects. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'19, page 10, USA, 2019. USENIX Association.

[44] proc(5) - Linux manual page. https://man7.org/linux/man-pages/man5/proc.5.html.

[45] Daisy OpenSSD Platform. https://www.crz-tech.com/crz/article/daisy/.

[46] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ openssd: Rapid prototype for flash storage systems. *ACM Trans. Storage*, 16(3), jul 2020.

[47] CRZ-Technology OpenSSD. https://github.com/CRZ-Technology/OpenSSD-OpenChannelSSD.

[48] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 147–162, 2021.

[49] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. Graphssd: graph semantics aware ssd. In *Proceedings of the 46th international symposium on computer architecture*, pages 116–128, 2019.

[50] Myoungsoo Jung. Openexpress: Fully hardware automated open research framework for future fast nvme devices. In *2020 USENIX Annual Technical Conference (USENIXATC 20)*, pages 649–656, 2020.

[51] OpenExpresss Download Page. https://openexpress.camelab.org.

[52] NVM Express Base Specification 2.0. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2_0-2021.06.02-Ratified-5.pdf.

[53] NVM Express NVM Command Set Specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-2021.06.02-Ratified-1.pdf.

[54] Intel Optane SSD 9 Series. https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html.

[55] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 153–165, 2016.

[56] zainryan/INSIDER-System: An FPGA-based full-stack in-storage computing system. https://github.com/zainryan/INSIDER-System.

[57] HLS Pragmas. https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html.

[58] Vivado 2021.2 - High-Level Synthesis (C based). https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html.

[59] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped i/o for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX-ATC 20)*, pages 813–827, 2020.

[60] Spark SQL and DataFrames | Apache Spark. https://spark.apache.org/sql/.

[61] 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*, pages 1221–1230, 2013.

[62] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment*, 7(11):963–974, 2014.

[63] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, page 257–270, USA, 2013. USENIX Association.

[64] Shine Kim, Yunho Jin, Gina Sohn, Jonghyun Bae, Tae Jun Ham, and Jae W Lee. Behemoth: A flash-centric training accelerator for extreme-scale dnns. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 371–385, 2021.

[65] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

Revitalizing the Forgotten On-Chip DMA to Expedite Data Movement in NVM-based Storage Systems

Jingbo Su[†] Jiahao Li[†] Luofan Chen[†] Cheng Li[†ζ] Kai Zhang[§] Liang Yang[§]
Sam H. Noh[‡¶*] Yinlong Xu[†ζ]

[†]*University of Science and Technology of China* [‡]*UNIST* [¶] *Virginia Tech*
[§]*SmartX*[†] [ζ]*Anhui Province Key Laboratory of High Performance Computing*

Abstract

Data-intensive applications executing on NVM-based storage systems experience serious bottlenecks when moving data between DRAM and NVM. We advocate for the use of the long-existing but recently neglected on-chip DMA to expedite data movement with three contributions. First, we explore new latency-oriented optimization directions, driven by a comprehensive DMA study, to design a high-performance DMA module, which significantly lowers the I/O size threshold to observe benefits. Second, we propose a new data movement engine, `Fastmove`, that coordinates the use of the DMA along with the CPU with judicious scheduling and load splitting such that the DMA's limitations are compensated, and the overall gains are maximized. Finally, with a general kernel-based design, simple APIs, and DAX file system integration, `Fastmove` allows applications to transparently exploit the DMA and its new features without code change. We run three data-intensive applications MySQL, GraphWalker, and Filebench atop `NOVA`, `ext4-DAX`, and `XFS-DAX`, with standard benchmarks like TPC-C, and popular graph algorithms like PageRank. Across single- and multi-socket settings, compared to the conventional CPU-only NVM accesses, `Fastmove` introduces to TPC-C with MySQL 1.13-2.16× speedups of peak throughput, reduces the average latency by 17.7-60.8%, and saves 37.1-68.9% CPU usage spent in data movement. It also shortens the execution time of graph algorithms with GraphWalker by 39.7-53.4%, and introduces 1.12-1.27× throughput speedups for Filebench.

1 Introduction

Emerging non-volatile memory (NVM) technologies such as STT-MRAM [45], PCM [40], ReRAM [6], and 3D-XPoint [15] offer byte-addressibility and comparable latency as DRAM but with substantially larger capacity. In addition, it provides data durability with orders of magnitude higher performance than prior durable devices like SSDs [55]. Recently, numerous studies have been proposed to combine faster, volatile DRAM, for caching, with slightly slower, denser NVM, for persisting data, in storage systems to revolutionize I/O performance of data-intensive applications with persistence demands [12].

In NVM-based storage systems, data are often moved between the two types of memories, due to DRAM cache fill-up, logging, or flushing. However, recent studies [28, 55] highlight that the DRAM-NVM data movement is not efficient, mainly because of their performance gaps in latency and bandwidth [13]. Additionally, we further notice that such data movement leads to heavy CPU consumption since NVM chips are attached to the memory bus, and their accesses must make use of the `load` and `store` instructions. Such negative performance effects worsen with multiple sockets, which modern high-end servers often provide, because of the negative NUMA impact [28]. This data movement bottleneck severely impairs the overall performance of I/O intensive applications and consequently, undermines the benefits brought by incorporating NVM.

To address this bottleneck, the slowness of NVM motivates us to re-think the usage of the on-chip DMAs that still come with the CPU but have deteriorated in use with the advent of fast storage devices. In this paper, we seek to transparently expedite data movement in NVM-based storage systems by (partially) offloading data movement to DMA to improve overall performance. However, while exploiting the on-chip DMA is a natural optimization, there are a few obstacles to incorporating it into NVM-based storage systems.

First, we need to handle more complex I/O patterns and have significantly different optimization goals than existing work [41, 54], which have already applied DMA as a minor technique to free CPU cycles of page migration in tiered DRAM-NVM systems. They handle I/Os that are always large, i.e., 2MB, and run in the background. However, NVM-based storage systems face I/Os with much smaller and variable sizes that are often on the critical path of the foreground user requests. Thus, our primary optimization goal is to shorten the execution time of DMA requests. Second, latency-critical optimization requires an in-depth understand-

[*] This work was done at UNIST.
[†] "SmartX" is also known as Beijing Zhiling Haina Technology Co., LTd.

ing of the strengths and limits of DMA, in conjunction with NVM and storage-facing I/Os, which is largely beyond existing studies [21].

To address the above challenges, first, we conduct a comprehensive study to understand the latency behaviors of using DMA for DRAM-NVM data movement on the Intel I/OAT and Optane PM combination, the only such pair in existence. This study suggests that the potential of DMA is heavily constrained by various factors, e.g., uneven advantages between reads and writes over the CPU, the non-negligible costs that grow with I/O size, bandwidth and concurrency limits, etc.

Second, we derive principles from the study to design `Fastmove`, a general data movement system that sits at the lower level of the software hierarchy. At the core, it includes a high-performance DMA module, which encapsulates the upper-level I/O requests into low-level hardware commands that comply with the workflows of data movement in NVM-based storage systems. We also maximize the benefits of DMA by introducing various optimizations such as batching the page pinning and descriptor submission activities for grouped DMA tasks and balancing and coordinating concurrent accesses to DMA channels. Furthermore, to compensate for the limitations of the stand-alone DMA solution such as the extra overhead and the concurrency and bandwidth constraints, we devise a lightweight `Scheduler` to prioritize bulk I/Os to go through DMA, while smaller I/Os are routed to the original CPU path. `Scheduler` additionally splits bulk read I/Os and balance loads between the DMA and CPU paths, adapting to real-time changes in DMA resource availability.

Finally, we incorporate `Fastmove` into the Linux kernel as an OS library with a limited number of simple APIs, which can be used to easily replace system functions that trigger data movements. To demonstrate its practicality, we adapt three NVM-based storage systems, `NOVA`, `ext4-DAX`, and `XFS-DAX`, to make use of `Fastmove` with minimal (2 to 4 lines of code) change. Consequently, applications running atop these systems can transparently enjoy the data movement acceleration brought by `Fastmove`. Additionally, we enable such acceleration for the cross-socket setting by deploying file systems atop the Linux device mapper with 2 lines of code change. This design enables the POSIX `read()` and `write()` APIs to freely employ `Fastmove`. To prove this, we successfully run three I/O-intensive applications, one industry-adopted database, MySQL [3], one graph engine, GraphWalker [49], and one file system and storage benchmark, Filebench [1] atop the modified file systems without any modifications to the applications.

We conduct extensive evaluations with three standard benchmarks FIO [8], fileserver [1], and TPC-C [5], and three popular random walk algorithms GraphLet, PageRank and SimRank. The results highlight that, for workloads containing substantial I/Os with moderately large sizes and beyond, considerable performance improvements are attained, regardless of local or remote NVM access. For TPC-C in MySQL, `Fastmove` increases its peak throughput by 13-116% compared to the original ones that use only the CPU, reduces the average latency by 17.7-60.8%, and saves CPU cycles used for data movement by 37.1-68.9%. Also, `Fastmove` brings 1.65-2.14× speedups of execution time for the GraphWalker algorithms, and 1.12-1.27× speedups of throughput for the Filebench fileserver workload.

In summary, `Fastmove` makes the following contributions:

- We present a comprehensive and general study to understand the characteristics of on-chip DMA in conjunction with NVM far beyond earlier studies [21], which showed DMA use just as a minor optimization in limited experimental settings [7, 25, 41].
- We propose and implement a fast memory copy engine `Fastmove` that accelerates DRAM-NVM data movement in NVM-based storage systems. Driven by the study findings, it incorporates new latency-oriented optimizations to reduce associated DMA costs and coordinates the CPU-only and DMA paths to maximize overall performance. `Fastmove`'s design principles significantly differ from earlier studies that concentrated on movement of data in a tiered memory setting [41, 54], where optimizations are simple due to the large size of memory copy requests.
- We present transparent in-kernel system support with integration of `Fastmove` into three NVM-aware DAX file systems, while extending the device mapper to enable cross-socket NVM access. This allows unmodified applications to run atop `Fastmove`.

2 Background and Motivation

2.1 NVM-based Storage Systems

NVM chips sit close to the CPU either by being placed on the memory bus and connected to CPU sockets via the processors' integrated memory controller (iMC) or by being exposed via cache coherence interconnects like Compute Express Link (CXL) [11, 18, 39]. In 2019, Intel released Optane PM, the first commercial NVM chip based on the 3D XPoint technology [15]. Beyond Optane PM, multiple companies are developing new products based on technologies other than 3D XPoint [15] such as STT-MRAM [45], FRAM [23], Nano-RAM [42], and ReRAM [6].

Despite the different implementations, they are expected to offer memory interfaces with byte-addressability, data persistence, and large capacity. Therefore, there have been extensive research focusing on incorporating NVM to build scalable storage systems [9, 22, 24, 27, 34, 53] that accelerate the data access of latency-critical, data-intensive applications. These applications persist all their data on NVM, while caching the working set and metadata like indexes in DRAM. When accessing non-cached data, applications need to load them from storage, while upon modification, the dirty pages and log entries need to be flushed back to storage for data durability.

Typically, they make use of NVM-aware DAX file systems such as `NOVA` that retain the standard file system interfaces

Table 1: I/O size (KB) distribution of various workloads

	size	TPC-C	fileserver	Graphlet/PPR/SR
read	[0,16)	-	80.2%	-
	[16,32)	100%	11.5%	
	[32,∞)	-	8.3%	100%
write	[0,16)	6.5%	82.2%	-
	[16,32)	82.9%	10.2%	-
	[32,∞)	10.6%	7.6%	-

and provide strong consistency guarantees along with various NVM-oriented performance optimizations [7, 20, 53]. Therefore, the aforementioned data copies often involve memory allocated in user space, while requiring kernel memory copy module support.

2.2 The Data Movement Bottleneck

DRAM-NVM data movement can be a critical bottleneck in terms of performance in data-intensive applications. To understand this, we perform a study on the I/O size distributions of various applications, from domains ranging from traditional SQL databases to graph analytic frameworks, and their impact on performance and resource usage.

As shown in Table 1, driven by the standard database TPC-C workloads with 5000 warehouses and 16KB innodb page size in MySQL, more than 93% of write I/Os in MySQL are beyond 16KB, where a significant number of these bulk writes are sitting on the critical path of writing logs for foreground update transactions. In the fileserver workload of Filebench, 8.3% and 7.6% of the reads and writes are beyond 32KB, respectively. Though the number of bulk I/Os is relatively small in fileserver, they already account for 44.1% of the overall data movement volume. Finally, GraphWalker, a single-threaded graph processing system, periodically reads from NVM into DRAM, all in 128KB chunks, which it later consumes with its in-memory processing [49].

To assess the negative impacts of data movement, we run the `msppr` workload [49] in GraphWalker atop `NOVA`, an NVM-based file system, with Optane PM. Note that `NOVA` uses Linux `memcpy` to access data on Optane and does not make use of SIMD as SIMD cannot be used within the kernel [43]. We find that over 92% of the execution time is spent on reading data from NVM under a single socket setting, while, when cross-socket data movement is involved, this number increases to over 97%. While these numbers will vary depending on the application, our observation is that for many applications, the time consumed for data movement is a clear bottleneck.

The inefficiencies of CPU-directed data movement are mainly caused by the performance gap between DRAM and NVM. In particular, with 6 interleaved Optane DIMMs within a single socket, reading a 4K page from Optane takes 952ns, 2.9× longer than that of DRAM. Similar to latency, PM shows 74.4%/35.3% lower read/write throughput than DRAM. Even worse, it takes 18 CPU cores for Optane to reach its peak load throughput while it only takes 5 for DRAM to reach a similar load throughput [55]. Finally, when accessing remote memory across sockets, both DRAM and NVM suffer negative NUMA effects due to the extra writes introduced by the default directory-based cache coherence protocol [28]. However, the performance loss of remote NVM accesses is larger because of its lower write bandwidth. Our findings are consistent with recent studies [14, 28, 55].

2.3 On-Chip DMA and its Challenges

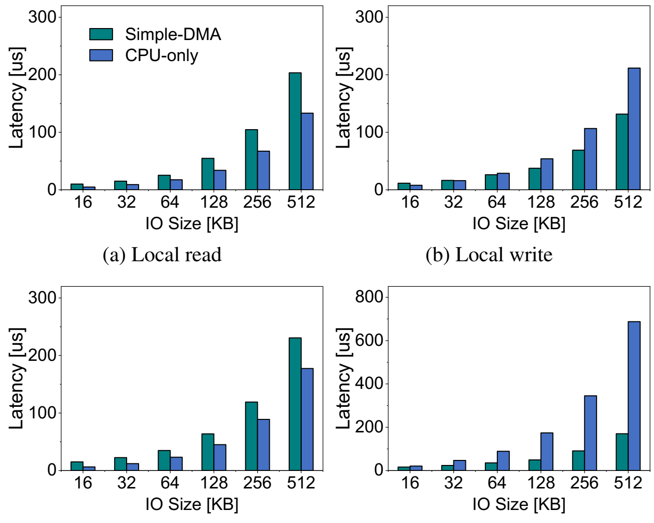
Modern processors have included on-chip DMA engines since as far as one can remember. For instance, Intel's I/O Acceleration Technology (I/OAT) DMA engine [16] lies in the integrated I/O module of the CPU, which also connects to cores and iMCs through a mesh interconnect. Similarly, AMD's second-generation EPYC processors are also equipped with on-chip DMA engines [37]. With the advent of high performance storage devices, however, they have deteriorated to a mostly unused component. The observations behind the data movement overhead problem motivate us to re-think the role of the on-chip DMA in NVM-based storage systems. We advocate that it will be beneficial to use on-chip DMAs to offload data copy jobs in NVM-based storage systems, thereby improving the copy performance itself as well as saving CPU cycles that could be used for other useful work.

To explore the latency improvement potential of DMA, we evaluate the speed of moving data between DRAM and NVM achieved by Intel I/OAT, in comparison with the CPU-only counterparts. Here, we refer to the I/OAT setting as Simple-DMA as we use it as-is without optimizations, which are explored later.

We use the FIO benchmark [8] to generate single-threaded read and write requests with I/O sizes ranging from 16KB to 512KB, where the former load data from NVM to DRAM while the latter store data in the opposite direction. These requests trigger kernel memory copy functions through `NOVA` to operate the underlying NVM—Optane PM [55], and we measure the time consumed for those functions.

Figure 1a and Figure 1c show that Simple-DMA performs consistently worse than CPU-only, and delivers 29.9-134.4% higher read latency, regardless of local and remote accesses. Contrary to reads, for local writes as shown in Figure 1b, Simple-DMA delivers comparable latency as CPU-only at 64KB, with meaningful differences expanding with I/O sizes from 128KB and beyond. For instance, when writing 256KB, the latency of Simple-DMA is only 64.7% of the CPU-only latency. Compared to the single-socket results, in Figure 1d, when considering two sockets, we observe that the performance of remote writes achieved by CPU-only and Simple-DMA both worsen. However, the request size threshold where Simple-DMA catches up with CPU-only becomes smaller at 16KB, which is only 25% of that observed for local writes.

The above latency comparison suggests that there is hardly any opportunity to allow reads within NVM-based storage systems to benefit from Simple-DMA; while for large writes, opportunities seem to exist. However, whether such large



(a) Local read (b) Local write

(c) Remote read (d) Remote write

Figure 1: Simple-DMA versus CPU-only read/write latency as request size is varied with FIO workloads.

writes (not smaller than 128KB for local writes) are amply available in typical applications is questionable. For example, as shown in Table 1 in our evaluation, around 80% of the bulk writes for MySQL-TPC-C concentrate on the range of [16KB, 32KB), which is certainly below the benefit threshold of Simple-DMA. Our conclusion is that we need to explore whether there are optimization opportunities.

Moreover, we have witnessed initial adoptions [7,21,41,54] of on-chip DMA to accelerate DRAM-NVM data movement. However, these early attempts mostly focus on tiered memory systems, and cannot be directly applied to NVM-based storage systems, which is our focus, due to the following reasons.

First, our optimization goal differs from using DMA in tiered memory systems, where data movements triggered by page migration run in the background, not on the critical path of user requests. Related works primarily focus their optimization goal on deriving advanced migration policies, and use DMA as a minor optimization to free CPU cycles [41]. In contrast, for NVM-based storage systems, data copy jobs such as user reads and log flushing are part of an end-to-end execution of foreground requests, which directly affect user experience. Thus, the key performance measure is latency.

Second, the I/O patterns and workflows differ significantly between NVM-based storage systems and tiered memory systems. The page migration workloads in tiered memory systems are quite simple and always happen at 2MB huge page granularity [41, 54]. In contrast, the sizes of bulk I/Os in NVM-based storage systems are much smaller and vary considerably. It is equally important that the workflow of handling memory copies via DMA in NVM-storage systems contains considerably more steps than that of tiered memory. These differences imply that the associated overhead of DMA is not negligible in NVM-based storage systems.

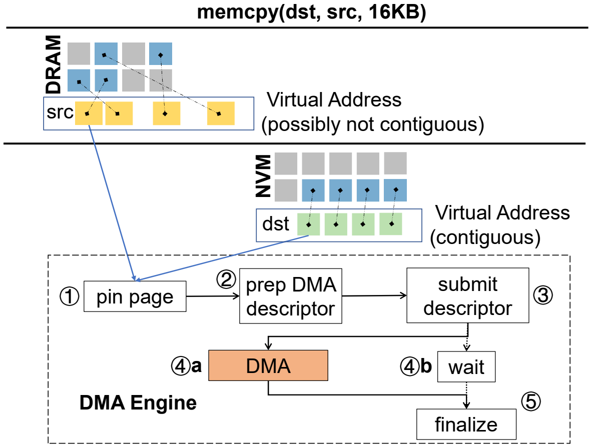


Figure 2: Workflow of memory copy using Simple-DMA.

In summary, the Simple-DMA performance, the demand for reducing latency and the storage-specific I/O patterns present us with unique challenges in making use of the DMA in NVM-based storage systems. In this paper, through an in-depth study of the behavior of on-chip DMA, we explore avenues of optimization opportunities. In addition, through `Fastmove`, we develop the necessary abstractions and transparent latency-sensitive optimizations so that applications may reap the benefits of the DMA without any code change.

3 DMA Optimization Opportunities

Here we provide a comprehensive study on DMA in conjunction with NVM to derive the optimization directions for lowering the latency of DMA-enabled memory copies and for unleashing its potentials to (partially) alleviate the above DRAM-NVM data stall problem.

3.1 DMA-enabled Data Moving Workflow

To begin our study, we first illustrate in Figure 2 the workflow of handling memory copy requests issued by applications via DMA, which implements exactly the same logic as the Linux `memcpy`. Take a 16KB I/O as an example. The virtual addresses of data residing in DRAM for NVM-based storage systems are possibly not contiguous, which leads to this single memory copy operation at the application side being divided up into four DMA subtasks. Each subtask corresponds to a 4KB page and will go through the following steps. ① pins the target DRAM pages as we need to prevent those pages from being swapped out or modified during DMA execution. An alternative way to do so is to allocate a DMA buffer, but at the cost of imposing extra memory copies or giving up transparent support to applications. ② prepares the DMA descriptor, the required metadata for I/OAT, which is then submitted to the hardware at step ③. Meanwhile, the submitter waits (④b) until the completion of ④a and reaches the final step ⑤ to finalize the corresponding DMA subtask execution, e.g., unpinning the page and notifying the application. Note that all steps except ④a are managed by a CPU thread, often the I/O thread of the application.

Table 2: Breakdown time costs of local read and write requests that use Simple-DMA

	size	cost (%)				#subtasks
	(KB)	pin	submit	I/OAT	other	
read	16	4.7	12.7	80.4	2.2	8
	32	4.9	14.1	78.7	2.3	16
	64	5.1	14.8	77.9	2.3	32
write	16	6.2	15.7	75.3	2.9	8
	32	7.1	19.8	69.8	3.3	16
	64	7.9	23.3	65.3	3.5	32

3.2 I/OAT and Optane PM Demonstration

To make the study concrete, in this section, we focus on the combination of Optane PM and Intel's I/OAT DMA.

3.2.1 Associated Time Costs

First, we investigate the latency breakdown results of Simple-DMA, which are summarized in Table 2, with the same setup as Figure 1a and Figure 1b. "pin", "submit", and "I/OAT" correspond to steps ①, ②-③, and ④a of Figure 2, respectively, while "others" denotes the remaining overhead.

The execution on the I/OAT hardware is the longest step of DMA-enabled memory copy requests across reads and writes. However, its ratio decreases from 80.4% to 77.9%, and 75.3% to 65.3% for reads and writes, respectively, when I/Os expand from 16KB to 64KB. In contrast, the associated overhead, excluding I/OAT, is also non-negligible and grows proportionally with request size, reaching to 34.7% for local 64KB writes. This is mainly because bulk I/Os within NVM-based storage systems trigger a series of I/OAT subtasks at 4KB granularity, as introduced in Section 3.1.

This growing overhead can be further doubled when the source and destination addresses of the corresponding I/O request are not aligned. Figure 3 illustrates such an example. The *src* of page#1 is not aligned with *dst* of page#a. As the DMA does not support cross-page copy when it cannot tell if the physical address is contiguous between pages, we have to split page#1 into two separate portions, namely ① and ②, where the former fits in the empty space of page#a, while the latter will have to fit on the lower part of page#b. Each of these portions will trigger a separate I/OAT subtask. Moreover, the remaining two pages #2 and #3 will go through the same effort. As the FIO workloads exhibit unaligned memory addresses, as shown in Table 2, bulk I/Os consist of 8-32 DMA subtasks and pay the associated time cost one more time. As this shows, in the case of transferring unaligned memory addresses, the overhead involved can turn out to be even more significant.

Trimming down the associated costs seems promising for improving the latency of writes. For instance, one can imagine that reducing them by 30.7% for 16KB writes will allow the DMA latency turning point to be reduced from 64KB to 16KB, enabling more applications, like MySQL, to gain performance benefits. However, this is not so with reads, since even completely eliminating these overheads still results in the DMA performing 11.1%-39.3% slower than CPU-only. Thus, other means to overcome this challenge must be conceived.

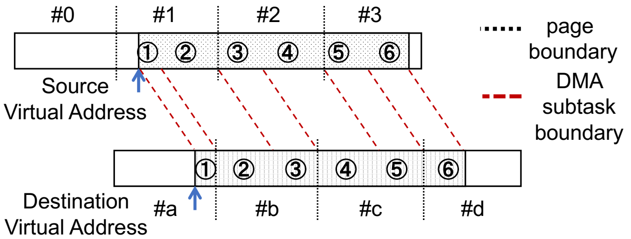


Figure 3: Composition of DMA subtasks for a single application data movement job with unaligned source and destination addresses. Three source pages (#1-#3) are involved but six subtasks are generated (①-⑥).

In addition, using Transparent Huge Page (THP) in the kernel makes the addresses, with high probability, to be contiguous. For contiguous copies, the cost of I/OAT still dominates, but with the submission and unalignment cost significantly diminished, compared to the above non-contiguous ones. This is because under such setting, memory copy requests will no longer be divided into multiple DMA subtasks.

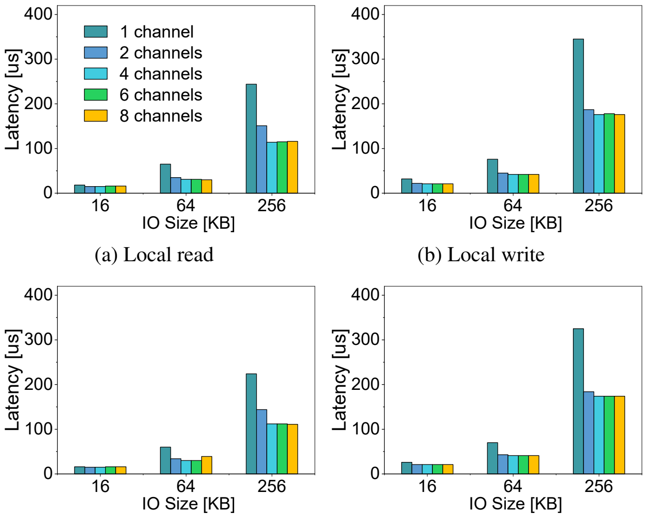
3.2.2 Intra-Request Parallel Copy

Each DMA device consists of M multiple channels that can process DMA subtasks in parallel. Therefore, we explore parallelizing hardware copy of a single request, where we split the request into N chunks ($N \leq M$) and thus, N DMA subtasks, each chunk making use of one channel. Here, we derive two different parallel execution modes, namely, para-A and para-B, where para-A uses a single submitter for channel submission, while para-B spawns N submitters, each of which manages its own channel independently.

For 64KB reads and writes, compared to Simple-DMA, para-A indeed reduces the I/OAT copy time, but the reduction is not proportional to the number of parallel chunks. In addition, we observe a significant increase in the submission overhead, which eventually offsets the benefits of intra-request, multi-channel parallel copy. In the end, para-A does not improve much on the end-to-end latency of Simple-DMA for reads, while even leading to performance loss for writes.

Para-B fares worse than para-A, worsening latency for both reads and writes. Our analysis shows that para-B sharply increases the hardware copy time by up to 68.7%. This is because of the heavy contention on DMA bandwidth driven by the parallel subtasks. This case differs from para-A, as the single submitter setting in para-A enables pipeline parallelism, which does not heavily stress the DMA. In addition, para-B introduces heavy CPU usage due to the multiple submitters.

Finally, as we cannot parallelize intra-request copies within DMA, we also explore the possibilities of balancing these copy subtasks between the CPU and DMA. Unfortunately, this is not applicable for writes, as using the DMA can easily saturate NVM's bandwidth. We find that the bandwidth competition can lead to amplified interference between the two tasks, resulting in 14.6% higher latency compared to the sole execution of using the DMA. In contrast, we find this



(a) Local read

(b) Local write

(c) Remote read

(d) Remote write

Figure 4: I/OAT latency when 4 FIO threads doing read/write workloads as number of channels and as request size is varied.

solution works well for reads as the DMA cannot consume all of the NVM bandwidth, and thus, the joint use of the CPU and DMA leads to better bandwidth consumption. We take this last approach as part of our optimization.

3.2.3 Impacts of Inter-Request Parallelism

Finally, we evaluate the impact of inter-request parallelism as in reality, application threads may concurrently execute data movement requests and make use of the DMA. First, we investigate whether using more DMA channels influences the performance of DMA operators. To this end, we use four concurrent threads to submit DMA requests to its multiple DMA channels on our two-socket NUMA machine. Here, we exercise up to 8 channels per DMA device/NUMA node. Figure 4 shows the latencies of DMA operators with varied I/O sizes. With the increasing number of channels, irrespective of local/remote reads/writes, the DMA operators become faster. For instance, compared to the 1 channel setting, adding one more channel leads to 38.1%-53.3% latency reduction for the 256KB memory copy operators. Trends are similar with more concurrent threads and cross-socket NVM accesses.

Second, we explore the changes in read/write effective bandwidth with the increase in the number of concurrent threads submitting DMA requests with bulk I/Os. We find that Simple-DMA observes an increase in read/write effective bandwidth for up to four threads, but beyond this, it starts to decline sharply. (Results not shown due to space limit.) The key limiting factor here is not drive scalability but, instead, the I/OAT DMA bandwidth. This suggests that a limit on concurrent DMA access should be set to prevent the DMA resource from being over-used.

3.3 Study Generalization

While the performance study above takes into consideration the performance characteristics of the underlying hardware, it also lays out the general study flow and key factors to be considered independent of particular NVM and DMA devices. With the advent of new hardware, the general study always needs to answer the following two questions:

First, *how can the DMA be best configured so that using it can be faster than CPU-only even for small I/O requests?* This part requires understanding the DMA subtask associated cost, the DMA parallel execution, and the effects of concurrency that drive the latency-oriented optimizations. Furthermore, it also requires exploring the effects of balancing loads among DMA channels and even between DMA and CPU.

Second, *how do we choose among the different copy paths?* We decide the best-effort path with the minimal time cost among three choices, namely, CPU, DMA, and DMA-CPU cooperation. Furthermore, we have to check if there are available DMA resources, i.e., the current DMA bandwidth usage, monitored during DMA execution, is below the profiled maximum bandwidths of DMA and NVM, respectively.

In summary, our general study framework will offer useful guidelines for accelerating data movement in storage systems that combine future DMA implementations and near-DRAM storage devices such as the upcoming CXL devices.

4 Overview of `Fastmove`

Driven by the study in Section 3, we aim to let data-intensive applications transparently make the best use of DMA to alleviate the NVM data stall problem presented in Section 2.2. Done properly, this should lead to better performance and alleviate CPU involvement required for memory copies between DRAM and NVM. First, we need to improve the latency of DMA-enabled data movement by taking into consideration the access constraints of DMA such as extra overhead, resource allocation, and interference within DMA or with CPU. Second, to complement DMA's limitations, we need to judiciously determine when and how much to resort to the normal CPU data path. Finally, while DMA is supported by Linux kernel functions, applications should not be burdened by high development and optimization overhead to exploit the DMA. Thus, a clean abstraction that requires minimal changes to applications is imperative.

4.1 `Fastmove`'s Architecture

Figure 5 shows the overall design of `Fastmove`, our efficient data movement engine. It sits below DAX file systems such as `NOVA`, `ext4-DAX`, and `XFS-DAX`, which are compatible with POSIX APIs and designed to use recent PM, as well as the Linux device mapper module, which allows file systems to use PMs across sockets. With this design, applications that run atop a POSIX file system should seamlessly be able to use our engine. `Fastmove` consists of three major system components, namely, `Scheduler`, DMA module, and CPU module. We retain the original design of the CPU module, where we let the corresponding I/O request execute the `load` and `store`

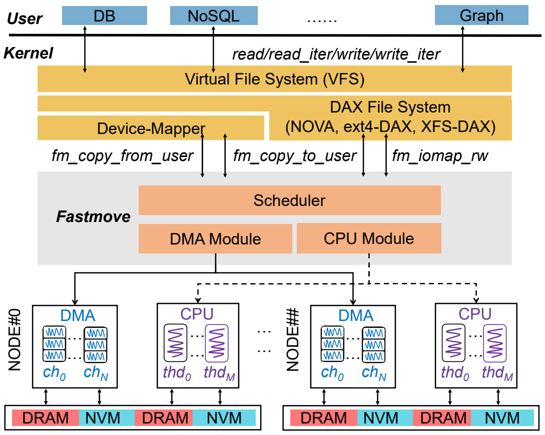


Figure 5: The overall architecture of Fastmove, which manages both DMA and CPU resources. Each NUMA node has a DMA device (dashed line box), which has multiple channels.

instructions as usual. However, we introduce a new DMA module that manages DMA resource allocation and memory copy offloading, with various optimizations to alleviate DMA costs and improve DMA resource usage. (Details will be discussed in Section 5.1.)

As the core logic, Scheduler is responsible for making decisions on selecting either DMA or CPU to execute requests. Its detailed design will be discussed in Section 5.3. This decision-making procedure should be fast so as not to incur overhead on the end-to-end request latency. It should also be smart so as to prioritize the use of DMA to fully make use of its strengths, while resorting to the CPU-only path as needed to compensate for the limitations of DMA for overall enhanced performance (Section 5.2).

4.2 API Abstraction

To exploit DMA transparently at the application level, we introduce three APIs that are simple extensions to existing APIs used by DAX file systems. The key observation here is that DAX file systems universally make use of a limited number of APIs for data movement, namely, copy_from_user, copy_to_user, and dax_iomap_rw. The first two are called by the read and write file system functions, while the last API is used by the read_iter and write_iter file system functions to perform memory copies in batches. These APIs are replaced by the APIs that we describe below.

As shown in Table 3, the three APIs that we introduce are fm_copy_from_user, fm_copy_to_user, and fm_iomap_rw. The first two new APIs have four arguments, dst, src, len, and bdev. dst and src specifies the destination and source of the copy (from PM to DRAM or vice versa), while len refers to the number of bytes to copy. The last argument bdev is the PM block device descriptor that includes rich information of the PM device such as the NUMA node id of the target PM. The last API fm_iomap_rw has three parameters, where iocb specifies the operational semantics such as read or write, iov_iter encodes parameters such as source and destination address vectors, and iomap_ops that is passed by file systems for I/O address mapping.

Table 3: Fastmove APIs

fm_copy_from_user(dst, src, len, bdev);
fm_copy_to_user(dst, src, len, bdev);
fm_iomap_rw(iocb, iov_iter, iomap_ops);

Finally, we only need to replace the old APIs with the new ones at the file system level. Thus, upper layer applications can take advantage of Fastmove without any code change. (Details are discussed in Section 5.4.)

5 Design and Implementation

5.1 High-Performance DMA Module

Under Fastmove, we offer a dedicated wrapper module to easily use the low-level primitives that DMA offers. This wrapper executes the I/O requests admitted by Scheduler. Here, we encapsulate the DMA requests by inheriting the values of parameters from the Fastmove APIs and the DMA channel assignment from Scheduler. Then, the wrapper executes DMA requests by going through all the steps in Figure 2 with the following major techniques and optimizations.

Batched DRAM page pinning. Memory addresses passed from user space are all virtual and need to be translated into physical ones that the DMA can consume. Furthermore, to satisfy DMA requirements, the virtual-to-physical address mapping must remain valid and unchanged during the execution of the corresponding DMA copy. This can be done by calling the pin_user_page and the dma_map_page kernel functions. However, pinning user pages one by one incurs high overhead for bulk I/Os, which span across multiple pages. To lighten this overhead, we leverage the pin_user_pages function available in the recent Linux kernel (version 5.9) that pins all the pages belonging to a single I/O. Similarly, we apply the same optimization for unpin_user_page via the new unpin_user_pages function.

src/dst page pairing. A bulk I/O will be mapped to a list of DMA subtasks at 4KB page granularity, each of which requires to pair the addresses of the source and destination pages for preparing the DMA descriptor. If the two addresses are not aligned, to ensure the correctness of DMA execution, which assumes that copies take place within page boundary, we have to carefully match the capacity of *dst* pages and the content size of *src* pages so that cross-page copies can be avoided. However, this leads to doubling the number of DMA subtasks, as described in Section 3.2.1.

Here, we make a key observation that NVM is managed contiguously in the kernel, and thus the cross-page copies can be tolerated. We exploit this finding as when preparing the DMA descriptor, we specify the length of the corresponding subtask in a page aligned manner on the DRAM side. For instance, take the situation in Figure 6 assuming that the source and destination are DRAM and NVM, respectively. We take the first portion of the source (① of page#1), which

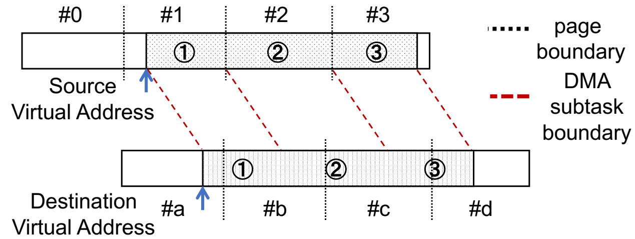


Figure 6: Composition of DMA subtasks with halved numbers for a data movement job, enabled by the contigious NVM address management, in comparison to Figure 3.

will always be smaller or equal to a page but aligned on the right end, as the size of the first DMA subtask. Thereafter, the size of the subsequent DMA subtasks will always be a page and aligned (② of page#2) except possibly for the last portion (③ of page#3), which will be page aligned on the left end. This enables us to reduce the number of DMA subtasks by half, in comparison to Figure 3.

Metadata buffer pre-allocation. DRAM space must be allocated with varying sizes to store the DMA request metadata, i.e., descriptors. The `scatterlist` structure is used to store the list of descriptors of DMA subtasks belonging to a single bulk I/O, where each item is typically 32 bytes. To accelerate memory allocation, we pre-allocate a fixed-size buffer to store this information prior to the execution of DMA copy. We set the buffer size to 4KB, which can accommodate DMA request metadata for 128 user pages (in total 512KB) at once.

Batch submission. Finally, to amortize the DMA subtask submission, considering that leveraging multiple channels performs no better than using a single channel (Section 3.2.2), we submit `scatterlist` in a batch to a single DMA channel assigned by `Scheduler`. This batched submission reduces the locking overhead for coordinating the concurrent accesses of the task queue associated with the DMA channel [17].

5.2 DMA-CPU Cooperated Bulk Reads

With Simple-DMA, the application thread (CPU) submits requests to the DMA, which solely moves the data (see top part of Figure 7). However, as shown in Section 3.2.3, bulk reads could be made faster through DMA and CPU cooperation. Motivated by this, we design an optimized bulk read within `Fastmove` that is depicted by the lower part in Figure 7. Here, the application thread first splits the bulk read into two chunks, and then submits one chunk (#1) via the normal DMA path with optimizations mentioned in Section 5.1, followed by the other chunk (#2) being copied by the CPU. Upon completion of chunk#2, the corresponding CPU thread polls the status of the DMA. Finally, the execution of the target read completes when both the DMA and CPU finish their assigned chunks. This design not only improves the NVM read bandwidth but also hides the copy latency due to the CPU.

While the optimized bulk read is a natural sharing of load, the challenge we face here is how to decide the loads that will go through the CPU and DMA. Chosen inappropriately, the gap between the execution time of CPU and DMA could lead to either waste of CPU cycles for polling the DMA status or lower DMA utilization. To balance their execution time, we set the chunk #1 and #2 size ratio to the ratio of the average single-threaded bandwidth on the CPU and DMA paths. which are monitored by our `Scheduler`.

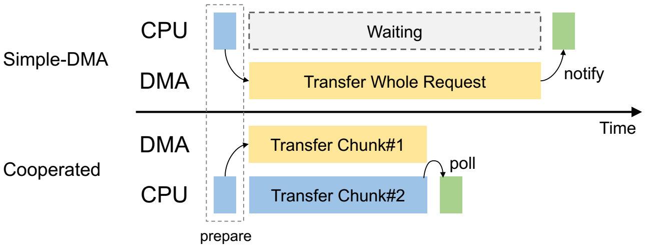


Figure 7: The workflow of DMA-CPU cooperated reads.

5.3 Controlling and Scheduling

We design a light-weight `Scheduler` that outputs the proper memory copy path assignment plan for each I/O request going through the above `Fastmove`'s APIs, distributes loads of bulk reads between CPU and DMA, and properly allocates DMA resources for offloaded tasks.

Initial configuration. Decision-making by `Scheduler` is driven by the four pre-chosen I/O size thresholds for local/remote NVM reads/writes, beyond which DMA path should be involved for better performance, and the concurrency sweet spot M per DMA device, which corresponds to the maximal number of concurrent threads leading DMA to reach the peak bandwidth. In addition, `Scheduler` also monitors the following four variables: (1) C_i, which is used to keep track of the number of on-the-fly requests submitted to device i and that works as an indicator of the workload intensity level of that device; (2) S_i, which points to the next available DMA channel on the DMA device i; and (3) B_C and B_D, that record the bandwidth dynamically consumed by the CPU and DMA, respectively.

Scheduling. `Scheduler` first inspects every I/O request to figure out the following parameters: the NUMA node id of the target NVM (N_P), the request type RW, the NUMA information LR, and I/O length L. RW and LR are both boolean values indicating read/write and local/remote, respectively. Then, the path scheduling logic is straightforward as follows. `Scheduler` compares the request length L to the DMA threshold, pointed by the pair of RW and LR, to identify bulk I/Os. For bulk I/Os, `Scheduler` chooses the DMA as long as the DMA device on node N_P is under its concurrent limit, i.e., $C_{N_P} < M$. If so, `Scheduler` chooses the next DMA channel associated with N_P's DMA in a round-robin fashion (based on S_{N_P}) and updates the required resource variables, i.e., $C_{N_P} = C_{N_P} + 1$ and $S_{N_P} = (S_{N_P} + 1)\ mod\ G$. Otherwise, we fall back to the CPU-only data path. Additionally, we use B_C and B_D to derive the split ratio of bulk reads between the CPU and DMA by following the logic presented in Section 5.2.

Performance consideration. To minimize the overhead that may incur due to request processing, we make the following two design choices. First, instead of implementing the Fastmove logic as a centralized component for coordination, we provide the logic as a function, which runs at the memcpy caller side. This precludes inter-thread communication between I/O threads and Scheduler helping enhance performance. Second, coordination of concurrent access to globally shared variables like S_N and C_N adopt lightweight mechanisms such as atomic counters to further reduce overhead.

5.4 Implementation Details

We implement Fastmove[1] under the DMA framework [32] in Linux kernel 5.9 with 2417 lines of C code for its core logic.
Integration with NVM-based storage systems. We integrate Fastmove into three widely-adopted, DAX file systems, namely, NOVA [53], ext4-DAX [30], and XFS-DAX [33], where NOVA is tailored for hybrid DRAM-NVM settings, while the other two systems are more general and compatible with NVM. Fastmove's transparent design leads to minimal changes to the above systems. Specifically, we introduce only 2 lines of code changes to both ext4-DAX and XFS-DAX, which simply replace the memory copy functions in read_iter() and write_iter() system calls with the APIs in Table 3. NOVA requires 2 additional changes to its read() and write() functions.

Though Fastmove enables NUMA NVM access by design, DAX file systems cannot naturally use NVM devices sitting across NUMA sockets. We address this problem by leveraging the Linux native device mapper [31], as shown in Figure 5. For the device mapper, similarly, only 2 lines in dm_copy_from_iter and dm_copy_to_iter functions need to be replaced. Note, however, that the current version of NOVA does not support the use of the device mapper. Therefore, we extend NOVA to work with the device mapper and its new code base can be found in Fastmove [1]. With these minimal changes, Fastmove is able to transparently benefit many applications that run atop these three file systems.
Correctness guarantee. The use of DMA in Fastmove will not introduce any data inconsistencies compared to CPU-only data accesses. First, while not mentioned in any public documentation from the hardware vendor, Kalia et al. [21] experimentally show that I/OAT preserves ordering during execution. Second, Fastmove always monitors the execution status of parallel DMA subtasks and knows which set of pages failed to be copied even though these pages may not be consecutive. This slightly relaxed memcpy semantic is enough since (1) most applications including filesystems and databases have their own well-designed fault handling mechanism, which can leverage Fastmove 's fault reports to recover state correctly, and (2) in kernel, there are many strict checks to avoid copy failures, such as permission validation prior to copy execution. Thus, failures will be rare.

6 Evaluation

6.1 Experimental Setup

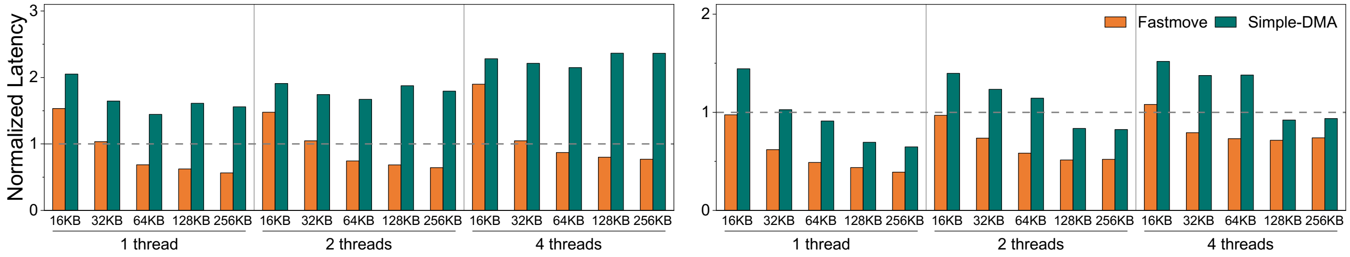
We deploy our experiments on a physical server with two 20-core Xeon Gold 6248 processors and 192GB DRAM. This machine has two NUMA nodes, each connected with six Intel Optane PM chips (128GB each and 1.5TB in total). We evaluate Fastmove with both the Optane PM device and emulated NVM to demonstrate the generality of Fastmove. With Optane PM, we configure it to be interleaved within each NUMA node and under the App Direct mode, and use the Linux device mapper under its striped mode to enable cross-socket NVM accesses. For the NVM emulated experiments, we use 64GB DRAM to emulate an advanced NVM device with DRAM-like latency and bandwidth, which is significantly better than Optane PM, using a Linux built-in emulator [35]. Note that our evaluation primarily focuses on Optane PM, while the emulated NVM performance results are only presented in Section 6.4.1.
Baseline and configurations. We exercise NOVA, ext4-DAX, and XFS-DAX enhanced by Fastmove. Our natural baselines are these file systems with their memory copy operations going through the conventional CPU path, denoted by "CPU-only". We use default configurations for both baselines.
Case study applications and workloads. We take three data-intensive applications MySQL, GraphWalker and Filebench, with no code changes, to transparently use Fastmove by simply running them atop the three slightly modified DAX file systems. To evaluate Fastmove's benefits, we run experiments with the FIO microbenchmark [8] and a synthetic workload generated based on FIO, application workloads like the widely-adopted standard database workload TPC-C [5] and the file access workload fileserver [1], and three popular graph processing tasks, namely, Graphlet Concentration, Personalized PageRank and SimRank. The detailed configurations are presented in Section 6.3.

6.2 Microbenchmark Results

6.2.1 Latency Threshold Choices

To help figure out the read/write thresholds with different concurrency levels required to drive the memory copy path selection in Fastmove, we run the FIO workloads to evaluate both the original and modified NOVA file systems. Here, we generate read and write workloads with different I/O sizes ranging from 16KB to 64KB, which are issued by 1 to 4 concurrent threads. We test both local and remote (cross-socket) NVM accesses. In Figure 8, we show the normalized average latency of Fastmove against the CPU-only baseline. (We omit the results for remote NVM access as they show similar trends with the local accesses.) In addition, we also include the results of "Simple-DMA", the baseline with DMA enabled but not highly optimized, to demonstrate the validity and effectiveness of Fastmove's optimizations and our Fastmove. Note that, these results look exactly the same across three

[1] Publicly available at https://github.com/fastmove-open/fastmove



(a) Local read

(b) Local write

Figure 8: The latency comparison between Fastmove and Simple-DMA, with 1,2,4-threaded FIO workloads, normalized to the latencies of CPU-only memory copying. (Remote read/write results omitted due to similar trends and space limit.)

file systems, thus we omit the evaluation of ext4-DAX and XFS-DAX for this part.

As shown in Figures 8a, across all exercised I/O sizes, CPU-only delivers constantly lower local read latency than Simple-DMA. In contrast, Fastmove visibly improves the performance of Simple-DMA and introduce 1.20-3.07× speedups for various I/O sizes, leading read requests with relatively small sizes to benefit from DMA. Compared to CPU-only, the turning points of Fastmove are 32KB across the 1, 2 and 4 threaded workloads, respectively. Including and beyond these turning points, Fastmove starts to observe a visible reduction in average request latency. For instance, Fastmove reduces the local read latency of CPU-only accesses by 13.0-25.6% for 64KB. While not shown, for remote read latency requests with I/O size starting with 32KB can benefit from Fastmove while 64KB for Simple-DMA.

For writes, we observe larger improvements than reads. Figure 8b shows that for local writes, Simple-DMA runs faster than CPU-only at 64KB, 128KB, and 128KB for the three concurrency settings, respectively. Fastmove dramatically improves Simple-DMA's latency, and drops the turning points to 16KB, 16KB, and 32KB. With 2 threads, Fastmove achieves 36.9%-49.0% and 26.3%-48.6% reduction on average latency for I/Os at 32KB and beyond, compared to Simple-DMA and CPU-only, respectively. The benefits of the two DMA variants further expand for remote writes (again, not shown). First, they perform better than CPU-only for even 16KB. Second, the latency gap between the DMA usage and CPU-only becomes visibly larger, e.g., for 256KB cross-socket I/O requests, Simple-DMA and Fastmove reduce latency by 75.3% and 86.1%, respectively, compared to CPU-only. Third, Fastmove significantly outperforms Simple-DMA by 40.7-48.1%, 65.9-73.7%, and 86.7-96.2% for 16KB, 32KB, and 64KB, respectively.

Finally, Table 4 illustrates the impact of the batched submission optimization on tail latency. We find that batching within Fastmove does not prolong, but rather, improves tail latency. For instance, with the same setting of 2-thread experiments in Figure 8, the P99 latency numbers in Table 4, indicating a 8.0-38.5% reduction, compared to the non-batching baseline. This is because Fastmove is not batching DMA subtasks across I/O requests from upper applications but is batching submissions of DMA subtasks that belong to a single request.

Table 4: P99 latency (us) comparison of local read/write with batching enabled or disabled in Fastmove, corresponding to the same setting of 2-thread experiments in Figure 8.

size	read		write	
(KB)	batching	non-batching	batching	non-batching
16	8	13	10	11
32	9	11	14	19
64	16	21	23	30
128	27	37	46	55
256	49	72	80	87

6.2.2 Breakdown Analysis

We use two synthetic FIO workloads to investigate the performance improvements introduced by each individual optimization within Fastmove. The bulk dominating workload contains I/Os with an average size of 256KB, while the mixed one has a mixture of bulk and small I/Os, ranging between 8KB and 256KB. For the two workloads, we use 6 concurrent threads to issue local read or write requests to the underlying NOVA file system.

Figure 9 reports the normalized throughput numbers, which indicate that different workloads see different optimization sweet points. The direct usage of DMA with loads evenly distributed among channels leads to a 46.6% and 25.7% throughput drop for the bulk and mixed workloads, respectively, compared to CPU-only. This is because small I/Os do not benefit, yet still go through the DMA, and the associated DMA overheads have not yet been ameliorated. As we start to avoid overloading the DMA resources by adding the concurrent limit optimization (here, set to 4), Fastmove's performance improves by 23.0% and 16.5% for the two workloads. The batching optimization makes Fastmove begin to outperform CPU-only, with a throughput increase of 55.8% and 17.9%. The latency threshold filtering further improves Fastmove's performance by 0.3% and 9.5%, where the mixed workload observes larger improvements as this optimization avoids its small I/Os from paying the latency penalty of going through the DMA. Finally, the bulk read split design choice brings another 12.3% and 3.1% improvement. In the end, adding all these optimizations together brings a 1.15× improvement in throughput for the two workloads, compared to CPU-only.

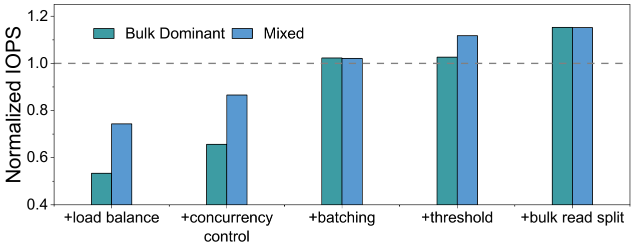


Figure 9: Breakdown analysis of `Fastmove` with synthetic FIO workloads when gradually enabling optimizations. Throughput is normalized to the CPU-only baseline.

6.3 Overall Performance

Next, we evaluate the positive impact of using `Fastmove` on the performance of real-world applications that introduce more complex characteristics than microbenchmarks such as non-uniformed I/O size distribution, computation-related cost, foreground and background processing division, etc.

6.3.1 Application Configurations

MySQL. We install MySQL version 5.7.33 with the default 16KB `innodb_page_size`. `innodb_buffer_pool_size` is set to half of the DRAM space, the recommended setting. We run the TPC-C workload with a read and write ratio of 1.78:1. For each run, we populate a 466GB database with 5000 Warehouses during the initialization phase and use 14 connections during the evaluation phase.

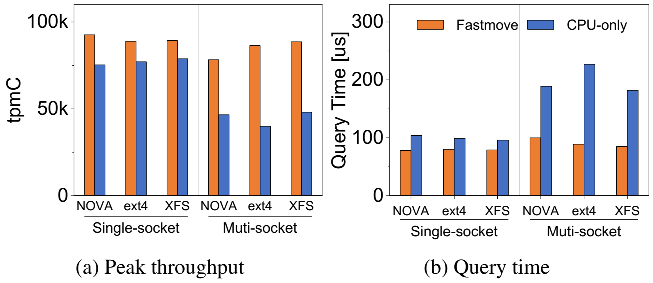
GraphWalker. GraphWalker [49] supports fast random walks on large graphs with a single machine. We exercise three common random walk algorithms, namely, Graphlet Concentration (Graphlet), Personalized PageRank (PPR) and SimRank (SR). We also follow GraphWalker to generate a Kron30 dataset using the Graph500 Kronecker [2], which consists of 1 billion vertices and 32 billion edges that take 638GB and 136GB of persistent media space to store its original text data and the compressed CSR data, respectively. We use the GraphWalker default configurations.

Fileserver. We exercise the predefined workload, fileserver, within the Filebench framework [1]. It uses 8 concurrent threads to issue I/Os with variable sizes presented in Table 1.

Enabling/disabling THP. We test MySQL and Fileserver without using transparent huge pages (THP), resulting in non-contiguous memory copies. This is recommended by the MySQL official site as THP introduces negative performance impacts on random memory accesses with small I/O sizes. Contrary, we enable THP for GraphWalker with contiguous copies, as its workloads are read-dominating and bulk-sized.

6.3.2 MySQL Enhancement

Single-socket results. First, we consider the performance within a single socket, where application threads and PM are located under socket 0. Figure 10a shows the throughput comparison (officially measured as tpmC by TPC-C) between CPU-only and `Fastmove` execution of MySQL. Across all settings, `Fastmove` consistently delivers better performance than CPU-only, and the improvements associated with different underlying file systems look similar. For instance, `Fastmove` introduces 1.23×, 1.15×, and 1.13× speedups of peak throughput over the CPU-only baseline across `NOVA`, `ext4-DAX`, and `XFS-DAX`, respectively. Figure 10b reports the corresponding average query time results. Consistent with the throughput results, `Fastmove` reduces the average latency of CPU-only by 17.7-25.0%.



(a) Peak throughput

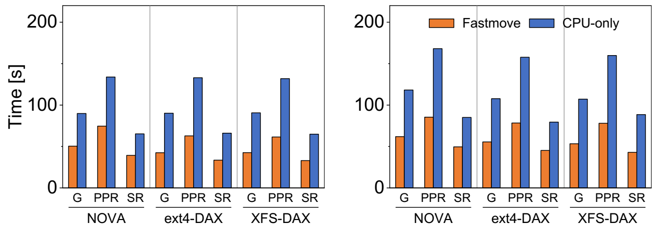
(b) Query time

Figure 10: Throughput (measured as tpmC) and query time achieved by running TPC-C against MySQL.

To understand the source of improvements, we profile the I/O distribution of the TPC-C workload. As shown in Table 1, almost all of its read requests are smaller than 32KB. As this is below the 32KB threshold, the vast majority of read requests go through the ordinary CPU-only path in `Fastmove`. As a consequence, the performance improvements here are driven by the 90.9% of bulk writes beyond 16KB, which correspond to the logging activities handled by the 4 background flush threads. To conclude, `Fastmove` indeed choose proper memory copy paths for I/O with varied sizes, and I/OAT DMA does alleviate the NVM accessing data stalls.

Multi-socket results. Next, we explore the performance implications under two sockets, where we replicate the above experiments by evenly distributing application threads to two CPUs and spreading the data on all 12 PMs via Linux device mapper under its stripped mode.

Figure 10a shows the absolute throughput numbers achieved by CPU-only with two sockets decrease by 38.1-48.1%, compared to the single-socket counterparts. This is because performance degrades for cross-socket memory copy operations as depicted in Figure 8. In contrast, `Fastmove` observes lighter negative impact of cross-socket NVM access with only 0.8-15.5% drop in peak throughput. `Fastmove` significantly outperforms the CPU-only baseline, introducing 1.68-2.16× tmpC improvements. Additionally, in Figure 10b, `Fastmove` brings a significant latency reduction of 47.1-60.8%. Contrary to the single-socket results, we see that `Fastmove`'s improvements over CPU-only expand. This is because the threshold for remote reads drops to 16KB, which allows for cross-socket NVM reads to take advantage of the DMA if DMA usage is not full, and also the DMA benefits for remote reads and writes are larger than those for local ones.



(a) Single-socket execution time (b) Multi-socket execution time

Figure 11: Execution times running Graphlet (G), PPR and SR over GraphWalker with `NOVA`/`ext4-DAX`/`XFS-DAX`.

6.3.3 GraphWalker Enhancement

Single-socket results. Figure 11a shows the execution times of three graph analytic tasks over GraphWalker. The performance of the CPU-only baseline looks similar across different file systems, and so does our `Fastmove`. However, we observe that `Fastmove` significantly reduces the execution time over CPU-only, despite the fact that the graph analytic jobs are read-only workloads towards the underlying data systems. More specifically, `Fastmove` introduces 1.78-2.13×, 1.79-2.14×, and 1.65-1.97× speedups for Graphlet, PPR, and SR, respectively. The significant improvements come from the dominating bulk read I/Os as shown in Table 1.

Multi-socket results. Consistent with the above TPC-C results, the improvements of `Fastmove` for the graph analytic workloads become larger compared to the single-socket counterparts. Figure 11b depicts that `Fastmove` brings 1.91-2.01×, 1.97-2.05×, and 1.71-2.06× execution time speedups for Graphlet, PPR and SR running in GraphWalker, respectively, across three different NVM-based file systems.

6.3.4 Fileserver Enhancement

The performance trends of the fileserver workload within Filebench atop `NOVA` look similar to those of TPC-C and graph algorithms above. Due to the space limit, we omit the figures. To summarize, `Fastmove` introduces 1.12× and 1.27× speedups in peak throughput, measured by IOPS, for the single-socket and multi-socket settings, respectively.

6.3.5 CPU Consumption Improvement

Finally, we explore another possible benefit of using `Fastmove`, which is the CPU consumption improvement. Here, we measure the CPU cycles spent in moving data between DRAM-NVM and processing the application logic. For MySQL TPC-C workload, `Fastmove` reduces its data movement CPU usage from 62% to 39% and from 90% to 28% for single-socket and multi-socket settings, respectively. We also observe a significant increase in its `utime`. This is because the saved CPU cycles from data movement are used to perform useful work, leading to improved throughput numbers (presented in Section 6.3.2). Unlike this, for GraphWalker, `Fastmove`'s CPU usage improvement seems little. For instance, `Fastmove` reduces its CPU usage for data movement by up to 5%. This is because workloads with GraphWalker benefits largely by the DMA-CPU cooperated bulk read optimization, which requires CPU involvement.

6.4 Other Factors

6.4.1 Emulated NVM Performance

We deploy `NOVA` on emulated NVM, replicate the experiments for Figure 8, and report the latency comparison results between CPU-only, Simple-DMA, and `Fastmove` in Figure 12. `Fastmove` outperforms CPU-only for local reads and writes with I/O sizes of 16KB and beyond. The benefits observed are larger than those corresponding to experiments with Optane PM (Figure 8). This is because emulated NVM is of DRAM-like read and write performance. Considering the association cost in Section 3.2.1, the dominating DMA copy execution step becomes faster, leading to visible end-to-end read/write latency improvements. Also, this implies that the time cost of NVM device access plays a key role in assigning DMA resources, i.e., the performance turning point based on I/O size decreases when NVM device performance improves, and vice versa. Furthermore, we find that the DMA bandwidth within `Fastmove` saturates when concurrency reaches 4 threads, exactly the same as the Optane PM experiments. This is because both the emulated NVM and Optane based experiments make use of I/OAT DMA, and under both cases, DMA bandwidth capacity is lower than Optane PM and emulated NVM.

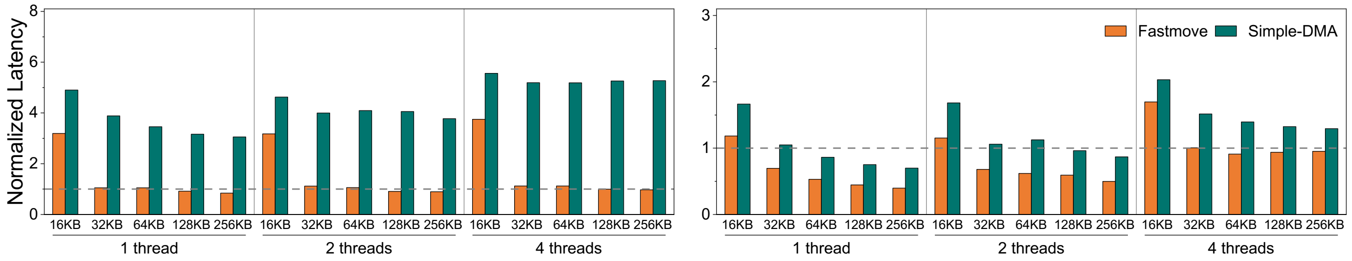
6.4.2 DDIO Impacts

Enabling DDIO introduces no impact on the CPU-only baseline. However, DDIO affects DMA reads and writes differently. To unify our settings, we chose to turn DDIO off for our evaluation. With DDIO enabled, MySQL-TPCC-Fastmove outperforms the CPU-only baseline by 5.4%, but performs 15% worse than the DDIO-disabled counterpart. Unlike this, we observe no differences for GraphWalker's three algorithms, when switching on/off DDIO. This is because DDIO makes DMA writes slower and thus, does not affect GraphWalker whose workload is read-only.

7 Related Work

I/OAT usage. Previous studies have used I/OAT to offload `memcpy` operations that move data from DRAM to DRAM [46,47] as well as to improve network bandwidth with lower CPU utilization in data center environments [25, 48]. Unlike these, our study stands to speed up data movement between DRAM and NVM, where the interaction between I/OAT and hybrid memory architectures is more complex and its acceleration demands careful system design. Most recent work have included I/OAT as a minor optimization for data movement in NVM-based systems with special purposes ranging from log replication [7] to memory migration [41].

Unlike them, `Fastmove` is a general system to make use of on-chip DMA to address the inefficiencies (e.g., lower bandwidth or extra CPU overhead) introduced by CPU-only accesses to NVM for bulk, storage-facing I/Os, which has



(a) Local read

(b) Local write

Figure 12: The latency comparison between Fastmove and Simple-DMA, with 1,2,4-threaded FIO workloads , normalized to the latencies of CPU-only memory copying when deploying NOVA on in-kernel NVM-emulator.

been observed to be a critical performance limitation in combined use of Optane PM and Intel processors. These systems can take advantage of Fastmove with little effort. Kalia et al. [21] present a number of optimizations for efficient remote NVM accesses via network, which includes an initial attempt to use I/OAT to improve single-core RPC performance for bulk remote NVM writes. This work is orthogonal to ours.

NVM-related studies and systems. There is a large body of work focusing on the analysis of the basic performance characteristics of using NVM [12, 14, 51, 55, 56]. The rich findings from these studies have spawned numerous studies for re-designing scalable and high performance data structures [24, 28, 50], file systems [19, 29, 53, 57] and key-value stores [9, 27]. Our work extends the existing study by incorporating the interaction between NVM and DMA, and complements the prior NVM-based systems as they can benefit from either the general design or the real implementation of Fastmove to alleviate data stalls. OdinFS [57] decouples application threads from the background NVM access threads and additionally parallelizes NVM accesses across sockets. Its NVM threads can benefit from Fastmove and its integration will be explored in the future.

Tiered memory systems. Fastmove handles more complex I/O patterns than those in tiered memory. In addition, Fastmove is implemented in the kernel with simple APIs. Therefore, Fastmove could be directly used in tiered memory systems. In fact, we have successfully adapted Nimble [54] to transparently use Fastmove through simple API replacement. However, through preliminary evaluations, we find that the DMA, in particular I/OAT, may not be a good option for improving page migration in tiered memory. This is because the DMA bandwidth is easily overwhelmed by the workload. Therefore, Fastmove does not deliver any significant improvement over Nimble-DMA [54], a Linux patch that adapts Nimble to use I/OAT.

Zero-copy technologies. Another line of work on PM attempts to move data management from kernel space to user space to eliminate data copies along the I/O path. For instance, the memory mapped file I/O (e.g., the mmap system call) is enabled such that users may access files in the same way as memory data [52]. However, mmap-based solutions may incur high overhead due to page faults [10, 26] and may have to have applications handle data persistence and reliability on their own [36, 38]. Yet another line of work leverages kernel by-pass I/O interfaces such as SPDK and PMDK [4] to avoid the use of the complicated OS I/O stack [44]. However, the performance gains come at the price of substantial effort for re-writing the I/O handling part of the target applications.

In contrast, our work demonstrates better applicability since there is no code change required to run existing applications atop Fastmove, as long as they use kernel file systems. Moreover, it is possible to extend our design to handle memory copy operations in user space, where these operations may have an even bigger impact on the overall performance compared to their counterparts in kernel space. This is because by bypassing the kernel, memory copying will contribute to a larger portion of the end-to-end access performance.

8 Conclusion

In this paper, we first study the DRAM-NVM data movement problem and then propose and implement Fastmove, a general engine that exploits the on-chip DMA technology. With a clean abstraction and transparent design, applications can use Fastmove via slightly-modified file systems with no further changes. Experimental results with industry-standard workloads on MySQL and popular random walk algorithms on GraphWalker highlight that Fastmove brings significant benefits such as peak throughput increase, execution time reduction, and CPU consumption savings.

9 Acknowledgment

We sincerely thank all anonymous reviewers for their insightful feedback and especially thank our shepherd Sanidhya Kashyap for his guidance in our camera-ready preparation. This work is supported in part by the National Natural Science Foundation of China under Grant No.: 62141216, 62172382 and 61832011, the USTC Research Funds of the Double First-Class Initiative under Grant No.: YD2150002006, and the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00503, Researches on next generation memory-centric computing system architecture). Cheng Li is the corresponding author.

References

[1] Filebench. https://github.com/filebench/filebench. [Online; accessed Jan-2023].

[2] Graph500. https://graph500.org/. [Online; accessed Jan-2023].

[3] MySQL. https://github.com/mysql. [Online; accessed Jan-2023].

[4] PMDK. https://github.com/pmem/pmdk. [Online; accessed Jan-2023].

[5] TPC Benchamrk C. http://tpc.org/tpcc/. [Online; accessed Jan-2023].

[6] Hiroyuki Akinaga and Hisashi Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010.

[7] 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.

[8] Jens Axboe. FIO. https://github.com/axboe/fio. [Online; accessed Jan-2023].

[9] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An efficient hybrid pmem-dram key-value store. *Proc. VLDB Endow.*, 14(9):1544–1556, may 2021.

[10] 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)*, Santa Clara, CA, July 2017.

[11] CXL Consortium. Compute Express Link: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/, 2022. [Online; accessed Jan-2023].

[12] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. Maximizing persistent memory bandwidth utilization for olap workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 339–351, New York, NY, USA, 2021.

[13] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the idiosyncrasies of real persistent memory. *Proc. VLDB Endow.*, 14(4):626–639, December 2020.

[15] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[16] Intel. Intel I/O Acceleration Technology. https://www.intel.com/content/www/us/en/wireless-network/accel-technology.html. [Online; accessed Jan-2023].

[17] Dave Jiang. libnvdimm: add DMA supported blk-mq pmem driver. https://lore.kernel.org/linux-nvdimm/150412628764.69288.12074115435918322858.stgit@djiang5-desk3.ch.intel.com/#r. [Online; accessed Jan-2023].

[18] 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.

[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 SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 804–818, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019.

[21] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 105–119, New York, NY, USA, 2020.

[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, 2018.

[23] Yoshihisa Kato, Yukihiro Kaneko, Hiroyuki Tanaka, Kazuhiro Kaibara, Shinzo Koyama, Kazunori Isogai, Takayoshi Yamada, and Yasuhiro Shimada. Overview and future challenge of ferroelectric random access memory technologies. *Japanese Journal of Applied Physics*, 46(4S):2157, 2007.

[24] Ana Khorguani, Thomas Ropars, and Noel De Palma. Respct: Fast checkpointing in non-volatile memory for multi-threaded applications. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 525–540, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 756–771, New York, NY, USA, 2021.

[26] 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 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, page 1–8, New York, NY, USA, 2020.

[27] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, Carlsbad, CA, July 2022. USENIX Association.

[28] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. Pactree: A high performance persistent range index using pac guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 424–439, New York, NY, USA, 2021.

[29] 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)*, pages 35–50, Santa Clara, CA, February 2022. USENIX Association.

[30] Linux. Add support for NV-DIMMs to ext4. `https://lwn.net/Articles/613384/`. [Online; accessed Jan-2023].

[31] Linux. Device Mapper. `https://www.kernel.org/doc/Documentation/device-mapper/`. [Online; accessed Jan-2023].

[32] Linux. DMAEngine framework. `https://www.kernel.org/doc/Documentation/driver-api/dmaengine/`. [Online; accessed Jan-2023].

[33] Linux. xfs: DAX support. `https://lwn.net/Articles/635514/`. [Online; accessed Jan-2023].

[34] 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.

[35] Maciej Maciejewski. How to emulate Persistent Memory. `https://pmem.io/blog/2016/02/how-to-emulate-persistent-memory/`, 2016. [Online; accessed Jan-2023].

[36] 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)*, pages 97–111, February 2021.

[37] Philip Ng. Accelerating intra-host pvrdma storage traffic in a future dell amd server. talk at vmworld 2019, 2019. [Online; accessed Jan-2023].

[38] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped i/o on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, page 277–293, New York, NY, USA, 2021.

[39] Jonathan Prout. Expanding Beyond Limits With CXL-based Memory, 2022. [Online; accessed Jan-2023].

[40] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[41] 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 SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021.

[42] Thomas Rueckes. High density, high reliability carbon nanotube nram. In *Flash Memory Summit*, 2011.

[43] Stackoverflow. Why are SIMD instructions not used in kernel? https://stackoverflow.com/questions/46677676/why-are-simd-instructions-not-used-in-kernel, 2022. [Online; accessed Jan-2023].

[44] 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)*, pages 431–445, Carlsbad, CA, July 2022. USENIX Association.

[45] AA Tulapurkar, Y Suzuki, A Fukushima, H Kubota, H Maehara, K Tsunekawa, DD Djayaprawira, N Watanabe, and S Yuasa. Spin-torque diode effect in magnetic tunnel junctions. *Nature*, 438(7066):339–342, 2005.

[46] K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda. Efficient asynchronous memory copy operations on multi-core systems and i/oat. In *2007 IEEE International Conference on Cluster Computing*, pages 159–168, 2007.

[47] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing efficient asynchronous memory operations using hardware copy engine: A case study with i/oat. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[48] Karthikeyan Vaidyanathan and Dhabaleswar K Panda. Benefits of i/o acceleration technology (i/oat) in clusters. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 220–229. IEEE, 2007.

[49] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. Graphwalker: An i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 559–571, 2020.

[50] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. NyxCache: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 1–16, Santa Clara, CA, February 2022. USENIX Association.

[51] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys ’22, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery.

[52] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’19, page 427–439, New York, NY, USA, 2019.

[53] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016.

[54] Zi Yan. Accelerate page migration and use memcg for PMEM management. https://lwn.net/Articles/784925/. [Online; accessed Jan-2023].

[55] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020.

[56] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. MT^2: Memory bandwidth regulation on hybrid NVM/DRAM platforms. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 199–216, Santa Clara, CA, February 2022. USENIX Association.

[57] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM performance with opportunistic delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, Carlsbad, CA, July 2022. USENIX Association.

NVMeVirt: A Versatile Software-defined Virtual NVMe Device

Sang-Hoon Kim
Ajou University

Jaehoon Shim
Seoul National University

Euidong Lee
Seoul National University

Seongyeop Jeong
Seoul National University

Ilkueon Kang
Seoul National University

Jin-Soo Kim
Seoul National University

Abstract

There have been drastic changes in the storage device landscape recently. At the center of the diverse storage landscape lies the NVMe interface, which allows high-performance and flexible communication models required by these next-generation device types. However, its hardware-oriented definition and specification are bottlenecking the development and evaluation cycle for new revolutionary storage devices.

In this paper, we present NVMeVirt, a novel approach to facilitate software-defined NVMe devices. A user can define any NVMe device type with custom features, and NVMeVirt allows it to bridge the gap between the host I/O stack and the virtual NVMe device in software. We demonstrate the advantages and features of NVMeVirt by realizing various storage types and configurations, such as conventional SSDs, low-latency high-bandwidth NVM SSDs, zoned namespace SSDs, and key-value SSDs with the support of PCI peer-to-peer DMA and NVMe-oF target offloading. We also make cases for storage research with NVMeVirt, such as studying the performance characteristics of database engines and extending the NVMe specification for the improved key-value SSD performance.

1 Introduction

NAND flash memory gains significant popularity for consumer devices and enterprise servers, and the fast advancement of semiconductor technologies fosters the non-volatile memory (NVM) to build storage devices, enlightening high-density low-latency storage devices. Nowadays, we can purchase off-the-shelf storage devices, which feature tens of microsecond latency and several GiB/s of bandwidth [16, 47].

Along with the performance and data density improvement, there has been an active trend toward making storage devices smarter and more capable. For efficient and effective data processing and management, many innovative device concepts have been proposed, including but not limited to Open-Channel SSD (OCSSD) [5, 34, 41], zoned namespace SSD (ZNS SSD) [4, 12], key-value SSD (KVSSD) [14, 19, 23, 45], and computational storage [8, 11, 20, 29, 31, 33, 52]. These new types of devices are significantly diversifying the storage device landscape. In this trend, software-based storage emulators are becoming more important than ever. For instance, when academia and/or industry propose an innovative storage device concept, fully developing an actual product from the conceptual idea takes a while. Meanwhile, we can implement a new concept in an emulator and see its benefits and pitfalls while running real workloads. This can provide us invaluable insights, facilitating rapid design space exploration. Moreover, by collecting various performance metrics from the emulator, we can understand the I/O characteristics of operating systems and the applications. This information can be used to optimize both the software and hardware of the target system. Finally, each emulator has a sophisticated performance model along with many knobs that can control a certain performance characteristic of the emulated device. This can help us predict the application performance on future storage devices that exhibit different performance characteristics.

However, to the authors' best knowledge, none of the previously proposed emulators fully satisfy the requirements to be used in the modern storage environment. Many emerging device types are often optimized to their primary targeting workloads and require a customized communication model between the host and device. This requirement makes the *NVMe interface* the most preferred interface for the emerging device types due to its flexibility and extendibility. This implies that a proper storage emulator should provide a comprehensive method to customize at the NVMe interface level. However, emulating the full NVMe interface in software is challenging as the NVMe interface inherently involves the protocol defined at the hardware level. Previous work proposes to circumvent the difficulty of emulating the NVMe interface by interposing hooks in the host NVMe device driver or leveraging virtualization technologies [12, 32, 35, 55]. However, these approaches fail to present a suitable NVMe device instance that is fully functional in the diverse modern storage environments such as when the kernel is being bypassed [24, 54] or when a

	Simulators		Emulators				
	Trace-driven [30,36]	**Full-system [10,22,49]**	**VM-based [12,32,55]**	**Block-driver level [56]**	**NVMe-driver level [35]**	**HW platforms [21,28]**	**NVMeVirt**
Deployable in real environments	No	Yes	Yes	Yes	Yes	Yes	**Yes**
Execution speed	Fast	Very slow	Slow	Fast	Fast	Real-time	**Real-time**
NVMe Multi-queue support	No	Yes	Yes	No	Yes	Yes	**Yes**
NVMe interface modification	Impossible	Easy	Easy	Impossible	Easy	Difficult	**Easy**
Low-latency device support	Possible	Possible	Difficult	Possible	Possible	Difficult	**Possible**
Kernel bypassing with SPDK	No	No	No	No	No	Yes	**Yes**
PCI peer-to-peer DMA support	No	No	No	No	No	Yes	**Yes**
NVMe-oF target offloading	No	No	No	No	No	Yes	**Yes**

Table 1: Comparisons of various virtual storage device approaches.

device directly accesses storage devices through NVMe-over-fabrics or PCI peer-to-peer communication [3, 9, 42]. Table 1 compares the previous approaches and their limitations.

This paper presents NVMeVirt, a storage emulator facilitating software-defined NVMe devices. NVMeVirt is a Linux kernel module providing the system with a virtual NVMe device of various kinds. Currently, NVMeVirt supports conventional SSDs, NVM SSDs, ZNS SSDs, and key-value SSDs. The device is emulated at the PCI layer, presenting *a native NVMe device to the entire system*. Thus, NVMeVirt has the capability not only to function as a standard storage device, but also to be utilized in advanced storage configurations, such as the NVMe-oF target offloading, kernel bypassing, and PCI peer-to-peer communication. In addition, this level of emulation allows developers to modify the NVMe interface layer easily, making it possible to explore various design spaces over NVMe and to support new device types. Unlike other emulators with similar goals, NVMeVirt does not rely on the virtualization technology, allowing comprehensive communication models at a consistently low overhead. The performance of these devices can be controlled with several performance knobs, making the virtual device perform close to real devices. Hence, NVMeVirt opens up a new opportunity for co-designing highly intelligent storage devices over the NVMe interface and stimulates the invention of a novel storage device architecture.

In the evaluation, we demonstrate the supported features of various device types with a working prototype. We explain two case studies to demonstrate that NVMeVirt can be helpful for storage domain research and engineering. The source code of NVMeVirt is publicly available at https://github.com/snu-csl/nvmevirt. The followings are the contributions of this paper:

- Provide a software framework to facilitate NVMe device research with various and stable I/O characteristics.
- Envision the fast prototyping and development of NVMe devices and interface through a software-defined NVMe device.
- Analyze the correlation and impact between the application and storage performance using representative database benchmarks.
- Make a case for extending the NVMe interface to improve key-value SSDs.

The rest of the paper is organized as follows. Section 2 explains the background and related work. Section 3 discusses the motivation of our work and explains the internal structure of NVMeVirt. Section 4 shows the evaluation result of NVMeVirt by representing its flexibility and feasibility. Finally, Section 5 concludes the paper.

2 Background and Related Work

2.1 NVM Express Standard

In modern computer architectures, peripheral devices are often connected to the processor through Peripheral Component Interconnect Express (PCIe) links [18, 42]. PCIe defines the entire communication stack, from the layout of connector pins to the message protocol between the host and devices. NVM Express, or NVMe, was first aimed to extend the PCIe communication protocol for emerging non-volatile memory devices, such as solid-state drives (SSDs). As designed from the ground up for modern storage devices, NVMe provides a more efficient low-latency interface than legacy interfaces, such as Small Computer System Interface (SCSI) and Serial ATA (SATA). Later, the NVMe specifications are extended further to support various storage device types, such as zoned namespace (ZNS) SSDs [4, 12] and key-value SSDs [14, 19, 23, 45].

The latest NVMe 2.0 specifications were announced in June, 2021. They comprise multiple documents: NVMe Base Specification, Command Set Specifications (NVM Command Set specification, ZNS Command Set specification, KV Command Set specification), Transport Specifications (PCIe Transport specification, Fibre Channel Transport Specification, RDMA Transport Specification, and TCP Transport Specification) and the NVMe Management Interface Specification. The Base Specification defines the host control interface. The Command Set Specifications contain the host-to-device protocol for SSD commands used by operating systems for read/write/flush/trim operations, firmware management, error handling, etc. As observed from the Transport Specifications, NVMe operations can be performed over various transport layers, such as PCIe, TCP/IP, and remote DMA (RDMA). Specifically, combined with RDMA-capable transport, NVMe-oF allows NVMe commands to be delivered to remote nodes and directly routed to target devices [9]. If the network adapter supports the target offloading feature, the NVMe commands can be processed completely at the hardware level without any involvement of software layers on the remote node. Thus, NVMe-oF can minimize the latency for disaggregated storage and is considered a key technology for high-performance scalable storage systems in future data centers.

2.2 NVMe Operation

The NVMe specifications standardize two communication interfaces for NVMe devices: *NVMe control block* and *NVMe message queues*. The NVMe control block is the primary path for setting up the NVMe device. It contains several configuration fields with which the host device driver sets up the device. Specifically, the host can specify the location of the administration queue pair, set and clear the interrupt mask, point to the address of the controller memory buffer (CMB), and shutdown and restart the device.

Meanwhile, the NVMe message queue is the interface primarily for scalable I/O. The NVMe architecture supports up to 65,535 I/O queues each with 65,535 commands (called queue depth). The queue can be created, modified, or destroyed by submitting requests to the special queue called *administration queue*. To perform I/O, the host driver builds an NVMe command according to the specification and submits it to the *submission queue* of the NVMe device. Each queue has the associated *doorbell*, which indicates the index of the latest request in the queue. When the host driver alters the doorbell, the NVMe device senses the change and starts processing the enqueued requests. The completion of the request processing is handled in a similar manner. Each submission queue has a paired *completion queue*, whereas the submission queue and the completion queues are collectively called a *queue pair*. When the I/O request processing is completed, the device places an NVMe completion message in the paired completion queue of the submitted request. The device driver on the host can sense the moment of completion by either polling the completion queue or waiting for an interrupt from the device. After processing the completion message, the device driver notifies the device of the completion by setting the doorbell of the completion queue. Accordingly, the device releases the resources associated with completed requests.

The administration queue pair is initialized during device initialization by specifying a physical address in the NVMe control block. The host can ask the device to create regular queue pairs by posting queue creation messages into the administration queue. The host can also make device management requests, such as identifying the device ID, querying supported features, and setting up an interrupt for completion notification, through the administration queue pair. The administration requests are processed in the same manner as regular I/O requests.

2.3 Related Work

A myriad of studies has attempted to imitate real storage devices in software [10, 12, 21, 22, 28, 30, 32, 35, 36, 49, 55, 56]. As summarized in Table 1, we can classify these works into two categories: simulators and emulators. Simulators imitate the internal operations of real devices with a data processing model [10, 22, 30, 36, 49]. They often build the model for a target device, parameterize the performance of internal operations, and calculate desired performance metrics from the model. They enable a detailed analysis with sophisticated models. However, they are often limited as they rely on a trace collected from real systems or are extremely slow when the full system is simulated to run the real workload.

Emulators provide *device instances* to the host; hence, they can be used like a real device [12, 21, 28, 32, 35, 55, 56]. FlexDrive [35] proposes a software-defined NVMe device, similar to our work. By modifying the NVMe device driver, it controls the flow rate of I/O requests in the host I/O stack, allowing the exploration of the space of various device performances. Combined with a RAM disk, FlexDrive can be used for projecting the performance of future devices. However, it can only emulate the conventional SSD type and not the emerging devices, such as KVSSDs and ZNS SSDs. Also, because of its implementation as a modified device driver, it can only handle the simplest data communication where requests are coming down through the kernel I/O stack, thereby unable to support complicated I/O models, such as the NVMe-oF offloading, kernel bypassing, P2P device communication, etc. Finally, the NVMe driver is on the critical path of the host I/O subsystem, so it might be too intrusive to be applied to a working system.

FEMU [32] proposes an accurate and scalable virtual NVMe device using host virtualization technology. Specifically, FEMU provides guest operating systems with a virtual NVMe device by leveraging the device virtualization feature of the QEMU [44]. According to the split driver model, the frontend in the VM receives the NVMe commands, and forwards them to the FEMU backend running in the host operating system. Due to this organization, it requires to keep switching between the host and guest operating systems, incurring non-negligible and highly variable latency (see Section 4.2). In addition, the virtualized environment inherently limits the control of the virtualized device implementation. For example, to perform DMA (and RDMA), the PCI device should be able to access the memory in the DMA/physical address space of the host. This becomes complicated in the virtual machine environment where the guest physical memory is scattered in the host physical memory through the virtual memory schemes on the host. This prohibits the study and exploration of device-oriented approaches in particular, such as NVMe-oF-based technologies and P2P device communication.

3 NVMeVirt Internals

3.1 Motivation

The increasing demand for high-performance and efficient storage devices has been pushing the academia and industry to develop various new storage device types, such as NVM SSD, KVSSD, ZNS SSD, and computational storage. They usually require a custom host-device interface tailored to their primary target workloads to make them work most effectively. For example, KVSSDs are for handling a huge number of small key-value pairs. They are most effective only when the host-device communication layer can handle small key-value payloads efficiently. Computational storage devices require a mechanism to deliver the code to run on the device. In this sense, we can claim that the most innovative storage research can be fostered by making it easy to modify or extend the host-device interface.

Currently, the NVMe interface is the most preferred host-device interface due to its flexibility and extendibility. The NVMe protocol itself is flexible and easy to extend; however, applying any changes to an actual system is an entirely different matter. Specifically, the NVMe interface inherently involves a protocol defined at the hardware level. To extend the interface for a new device feature, the developer should incorporate the changes not only to the device driver on the host but also to the firmware or controller logic on the real devices. This level of work usually demands a huge amount of engineering efforts and research resources, restricting the research for novel storage devices.

This motivated us to build a storage emulator that provides a comprehensive way to customize the NVMe interface and support various storage device types on top. To this end, we attempt to virtualize devices from the PCI level so that they can behave like real physical devices from the entire host's point of view. We argue this is crucial for a storage emulator that should support various device types and advanced storage configurations, such as KVSSDs with custom operations, NVM-based ultra-low-latency SSDs, the target for NVMe-oF offloading, direct access from user-space bypassing the kernel, peer-to-peer data transfer between PCIe devices, and so on. We emphasize that this is the point of difference between NVMeVirt and previous work.

3.2 Virtualizing a PCIe/NVMe Device

To help understand the challenges in virtualizing NVMe devices, we first detail how PCIe/NVMe devices interact with the host [42]. As shown in Figure 1, the CPU and memory subsystem are connected to peripheral devices through a hardware component called *PCIe root complex*. The root complex generates PCI transactions to the devices on behalf of the CPU when the CPU accesses the device memory-mapped regions. The root complex has multiple PCIe ports, each of which can be connected to a PCIe device (i.e., PCIe endpoint) or a PCIe switch. The PCIe switch allows the hierarchical organization of PCIe devices by implementing a *PCIe bus*, through which multiple devices can be multiplexed.

PCIe devices, including NVMe devices, essentially communicate with the host operating system (and the host firmware) through a memory-mapped region for their initialization. A PCIe device is supposed to present its PCI *configuration header* in the PCI configuration address space. The configuration header contains essential information to initialize the device. This information includes the device ID, vendor ID, type code of the device, status of the device, and list of resources that the device provides. The host, specifically the root complex device driver, scans the PCI configuration address space to find the configuration headers presented by installed devices. For each detected configuration header, its corresponding device driver is invoked according to its device type and IDs. This process is called a PCI bus scan. To facilitate device-specific requirements during the PCI scan, the PCI subsystem in the Linux kernel allows customizing the operations for accessing the configuration header.

With this PCI initialization protocol, the most obvious way of creating a virtual PCIe device might be injecting a forged PCI configuration header into the root complex driver. However, in this case, when the root complex recognizes the PCIe device, it will attempt to directly communicate with the device at the hardware level. This inevitably leads to a system design that requires a hardware modification, which is impractical and even too intrusive for commodity servers.

To circumvent this pitfall, we virtualize PCIe devices *indirectly* through the PCI bus. First, NVMeVirt allocates a part of the reserved memory region for the PCI configuration header of the virtual device. The configuration header is set to indicate an NVMe device with required PCI capabilities (the "PCIe Device Emulator" part in Figure 1). With the configuration header, NVMeVirt creates a virtual PCIe bus with a non-existing PCI bus ID of the system (the ID is provided as a configuration parameter) and asks the PCI subsystem to scan the bus with custom configuration header operations. When the PCI subsystem performs the PCI bus scan, it effectively detects an NVMe-type device. When the subsystem attempts to initialize it by accessing the configuration header, NVMeVirt hooks in through the custom configuration header operations and emulates requested operations. This effectively presents an NVMe-type PCIe device to the PCI subsystem, making it ask the NVMe layer to initialize the device.

The NVMe device emulation is implemented on top of the PCIe device emulation. According to the NVMe specifications, NVMe devices should present their NVMe control

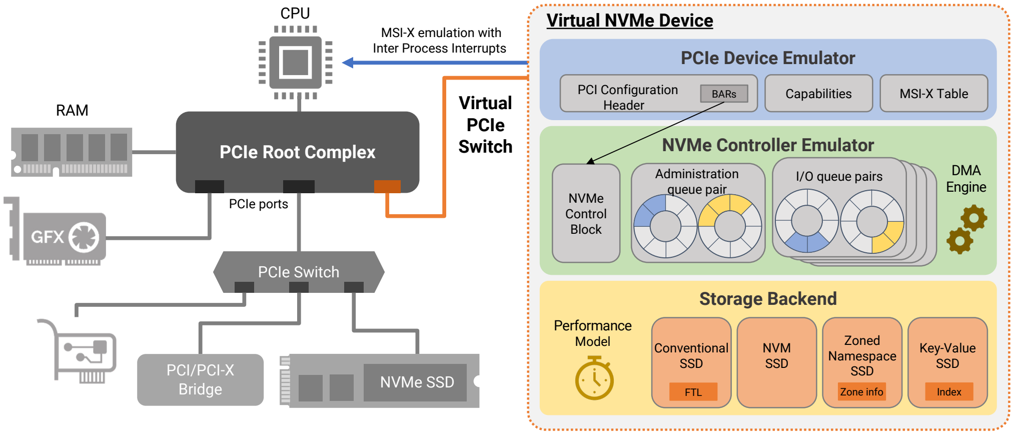


Figure 1: The overall architecture of NVMeVirt. NVMeVirt virtualizes a virtual NVMe device through the PCI bus and switch, so the device is seen as a native PCIe device from the host.

block through the base address register (BAR) fields in the PCI configuration header. Accordingly, NVMeVirt sets up the PCI configuration header so that the BARs point to a reserved memory region used for the control block. The NVMe layer on the host identifies the control block and operates on it according to NVMe standards.

In real devices, accesses to the NVMe control block are delivered to the device in the form of PCI transactions, initiating an action from the device. However, as the control block of NVMeVirt devices is only a memory region, accesses to it are processed silently without causing any event. To respond to the updates of the control block, we used the similar idea of vIOMMU [1]. Specifically, NVMeVirt runs a kernel thread called *dispatcher*. The dispatcher keeps checking the values of the control block to determine whether any changes have occurred. When the current value of the control block is changed since the last check, it implies that the host made some requests to the NVMe device. The dispatcher identifies the intention from the update and thus initiates the processing of the request.

We opt for the busy-waiting approach (i.e., keep scanning targets) over an event-driven approach (i.e., signal the dispatcher in response to incoming requests) to provide the low latency of NVM-based storage devices. The event-driven approach might save CPU cycles much; however, waking up the sleeping thread incurs non-negligible time overhead, making it unable to meet the demand for high I/O processing performance of modern storage devices.

Due to the emulation from the PCI layer level, NVMeVirt provides unique capabilities and opportunities that other emulators cannot provide. First, the emulated device operates like a real device from the perspective of the rest of the OS and even other devices. Any entity can instruct the NVMeVirt instance to perform any NVMe operations provided that it can set the control block and/or place operations in the NVMe queues under PCI and NVMe specifications. This makes it possible for a user-level application to directly access the device bypassing the kernel with SPDK [54]. In addition, even other PCI devices can place NVMe messages to the NVMeVirt instance according to the PCI peer-to-peer DMA protocol. This permits NVMeVirt to foster the studies using the NVMe-oF target offloading [9] and the direct communication between GPU and storage for AI applications [3]. Note that none of the previously presented simulators and emulators relying on device drivers or virtual machines can support these advanced storage use cases.

Another unique capability of NVMeVirt is that it allows the inspection of the NVMe message queues in detail. With real devices, it is infeasible to track the exact number of I/O requests queued in submission and completion queues from the host side since the device does not expose the processing progress to the host (i.e., the device is not supposed to report individual processing progress but only notify completions in bulk). As the dispatcher directly accesses the NVMe queue pairs and doorbells, we can track the exact state (i.e., queue depth of queue pairs, queuing delay, etc) of the device in software, allowing an in-depth understanding of the communication characteristics and behaviors. In addition, we can easily configure the maximum number of queue pairs by changing the tunable parameter, which enables the opportunity to study the implication of multi-queues on various configurations.

3.3 Supporting Various NVMe Device Types

While the dispatcher focuses on processing *the control requests* to the device, time-consuming *I/O requests* are handled by a set of kernel threads called *I/O workers*. As illustrated in Figure 2, each I/O worker maintains an *I/O process queue*, which lists the pending NVMe requests. When the dispatcher detects a doorbell ringing, it fetches the I/O requests from the corresponding submission queue. The dispatcher estimates

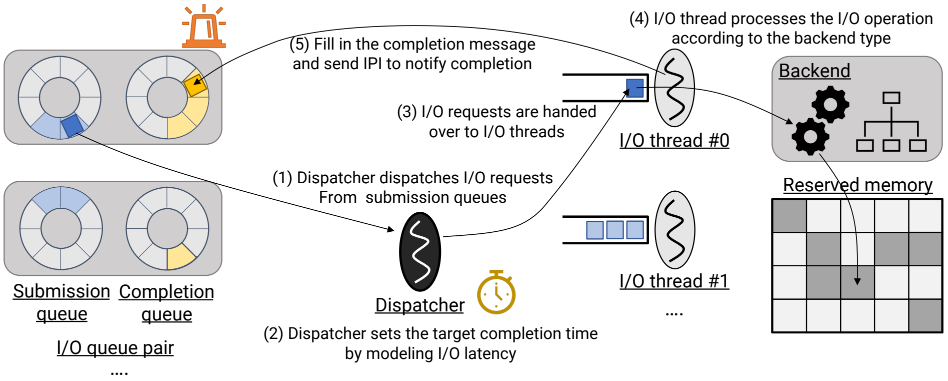

(4) I/O thread processes the I/O operation
according to the backend type
(5) Fill in the completion message
and send IPI to notify completion
Backend
(3) I/O requests are handed
over to I/O threads
I/O thread #0
Reserved memory
(1) Dispatcher dispatches I/O requests
From submission queues
I/O thread #1
....
Submission queue
Completion queue
Dispatcher
(2) Dispatcher sets the target completion time
by modeling I/O latency
I/O queue pair
....

Figure 2: Processing I/O requests in NVMeVirt. The host inserts an I/O request into the NVMe submission queue and rings the associated doorbell. The dispatcher in NVMeVirt dispatches the I/O request (1), and computes the target completion time according to the latency model of the deployed backend (2). Then the request is handed over to an I/O worker, which processes the operation of the configured backend type (3)(4). When the desired target completion time is passed, NVMeVirt inserts a completion message into the NVMe completion queue and signals an Inter-Processor Interrupt (IPI) to the core that made the I/O request (5). Finally, the host I/O stack eventually wakes up the context that waits for the completion of the I/O request.

the target completion time of each I/O request (Section 3.4 discusses the timing model) and hands over the request to an I/O worker by placing it in the corresponding I/O process queue. The target I/O worker can be selected in a round-robin manner or as desired, and requests are placed in the queue sorted by their completion time.

The I/O worker processes the requests depending on the device type that it emulates. Currently, NVMeVirt supports the conventional SSD, NVM SSD, ZNS SSD, and KVSSD, and the device type can be specified at compile time. NVMeVirt initializes the NVMe control block to advertise itself as the selected device type, making the corresponding host device driver interact with the NVMeVirt instance. To process an I/O operation for a device type, the I/O worker invokes the I/O processing routine of the corresponding *backend* of the device type. For example, the conventional SSD backend copies the data payload to the backend memory for write requests or copies data from the memory to a specified I/O buffer. The ZNS backend checks whether the request is valid according to the ZNS specification and processes it similarly to the SSD backend if the request is valid. The KVSSD backend looks up the requested key from the index and stores or retrieves the value of the requested key. The details of the data handling in the backend memory will be discussed in Section 3.5.

The data is copied from/to the backend memory using the Intel I/OAT DMA engine [15] instead of the traditional `memcpy` for improving the I/O processing performance and reducing the CPU overhead. When an application initiates an I/O request, the requested data is on some pages in the host (i.e., in the I/O buffer or in the backend memory for write and read requests, respectively). Thus, the CPU can copy data within the memory of the host at a low overhead. However, the overhead might be non-negligible when data is on the device memory for P2P I/O requests. To support the inter-device communication, the PCI root complex and MMU collaborate to present an illusion of device memory in the physical address space of the host. When a PCI device moves the data in the device memory through DMA, the PCI root complex routes the accesses to the target device at the PCI level, providing low latency for data moving. However, when the CPU accesses the device memory-mapped address range for `memcpy`, the accesses are translated into PCI transactions and processed by the PCI device at a small I/O unit. This inevitably and significantly impairs the data copy performance, making NVMeVirt unable to guarantee the high performance of real devices. As the DMA engine allows low-overhead data transfer from/to device memory-mapped memory regions, NVMeVirt can achieve compelling I/O processing rates regardless of the I/O configurations.

After performing the actual I/O operation, the I/O worker compares the current and target completion times of the request. If the current time passes the target completion time, the I/O worker places a completion message into the corresponding completion queue. To notify the host OS of the processing completion, NVMeVirt sends an Inter-Process Interrupt (IPI) to the waiting processor bound to the queue pair. NVMeVirt supports Message Signaled Interface-X (MSI-X) for a scalable high-performance completion path. Each queue pair has its own dedicated IRQ vector; hence, NVMeVirt can efficiently signal to the target core with the specified IRQ vector. The I/O stack on the signaled core will eventually wake up the user context that initiated the request.

3.4 Performance Model

For a device emulator, the capability to imitate the performance of real devices is important. To that end, there has been no shortage of studies attempting to replicate the latency and bandwidth characteristics of real storage devices with emulators and simulators [10,30,32,35,36,49,55]. We employed a similar approach as implemented in those works. Basically, an I/O request is divided into smaller chunks, which are independently processed in parallel by multiple data processing units. The data processing unit drives underlying storage media to read from or write to it. The chunk and data processing operation are, for example, considered the flash page and its read operation and programming in SSDs, respectively. The I/O completion time for an I/O request is determined by the completion time of the last operation for the request. The time for processing each chunk can be controlled with tunable parameters, which can be set based on the I/O bandwidth and latency of the target device. Further, the size of the chunk and the number of data processing units are configurable. With a set of parameter values we can expect a latency for small requests and the maximum bandwidth with large requests for the device. The latency is mainly determined by the operation time of the data processing operations, whereas the maximum bandwidth is bounded by the aggregated performance of the data processing units. For the remainder of the paper, we refer to these as the *target latency* and *target bandwidth*. For example, the OptaneDC SSD could be successfully modeled as the SSD that has a target latency of 12 us and a target bandwidth of 2,400 MiB for read requests. We refer to this model as the *simple model*.

We can produce the performance characteristics of real NVM SSD and KVSSD using the simple model as presented in Section 4.3. In general, the most complicated performance characteristics of flash-based storage devices are originated from garbage collection. However, as many studies have analyzed earlier [17,51,53], NVM SSDs are expected to allow in-place updates, enabling them to operate without garbage collection. For KVSSD, the size of key-value pairs is small; hence its performance tends to be bound by the host-device interfacing performance and the key indexing time, rather than the performance of the storage media. Thus, the simple model was sufficient for those device types.

Meanwhile, the model for conventional SSD is much more complex. FTL (Flash Translation Layer) controls data placement and performs garbage collection if necessary. In addition, modern SSDs aggressively use parallel processing techniques to maximize their performance. Accordingly, we had to redesign the performance model from the ground up to mimic the performance characteristics of real SSDs.

First, we implemented a page-mapped FTL based on that of FEMU [32]. The FTL determines data placement on the fly, and performs garbage collection when the number of free blocks gets below a threshold. Currently, the FTL selects victim blocks according to the greedy policy. To consider the on-device buffer, it is complemented with a small amount of memory as the write buffer. A write request is signaled to be completed as soon as the time to copy the payload into the write buffer is modeled. The buffer is processed in the background later with flash page writes.

In addition to the FTL, we revamped the data processing model to incorporate the parallel architectures widely used in modern SSDs. The storage space of modern SSDs is often divided into several partitions, and each partition is handled by one FTL instance [25]. Therefore, in the advanced model called the *parallel model*, multiple FTL instances exist in one SSD, and the instances share one PCIe link connected to the host. Data transfer from/to the host are serialized through the PCIe link. However, the rest of the FTL operations can be processed in parallel. Each partition comprises multiple NAND channels, which are connected to multiple dies in turn. Again, data transfer through the NAND channel are serialized, but the dies can operate independently.

In this architecture, FTL orchestrates the dies and channels to process I/O requests. For example, FTL instructs multiple dies to perform a read operation. When the dies are ready to transfer data, FTL schedules data transfer from the dies to the PCIe link through the shared NAND channels. The performance of the device is calculated based on the parameters of the FTL and model. We can speculate the parameters from the device specification and/or by observing the performance behavior of the device, similar to those approaches proposed in the literature [25,45,53]. With the parallel model, we can tune the parameters to provide a certain target latency and bandwidth independently similar to the simple model.

The ZNS SSD backend uses the same data processing model as the conventional SSD. However, it does not need the FTL since the host explicitly controls data placement according to the ZNS SSD specifications.

Note that NVMeVirt actually does not limit the performance model. Indeed, any performance models can be implemented and integrated into NVMeVirt as required.

3.5 Data Storage and Handling

NVMeVirt should store requested data somewhere in the system, and retrieve them later for read requests. Similar to many other storage simulators and emulators, NVMeVirt stores the data in the main memory. As running as a kernel module, NVMeVirt cannot luxuriate in comfortable functions from user space, such as virtual memory. However, the memory management overhead should be low and consistent to emulate future-generation devices such as PRAM- and MRAM-based SSDs. Different device types require different memory management policies and mechanisms. We explain the approaches for each device type that NVMeVirt currently supports.

Common. Regardless of the device type, NVMeVirt requires an extensive amount of memory for data storage. NVMeVirt obtains the required memory by reserving a part of the physical address space with the booting parameter during the system initialization. We configured the NUMA setting not to interleave memory, and the memory is reserved from a dedicated NUMA node where the NVMeVirt threads are pinned down. Therefore, the reserved memory is physically contiguous. At the beginning of the reserved memory area are the NVMe control block and PCI resources, such as the MSI-X table and PCI capabilities, followed by the bulk memory region used for storing data.

SSD and ZNS SSD. Basically the backends for SSD and ZNS SSD use a simple linear mapping for data placement. For a physical block/page number in the flash address space, its in-memory location is calculated by adding the starting address of the reserved memory region. The FTL for conventional SSDs maintains the logical-to-physical flash address space mapping on top of the linear mapping as in FEMU [32]. Once the target address is calculated from the block number, NVMeVirt moves data from/to the in-memory location. One might suggest reusing the RAM disk facility or using `alloc_pages()` or `vmalloc()` for allocating the data storing pages. In this case, however, we cannot control the location of pages from NUMA domains. Even if we can use `alloc_pages_node()` to specify the NUMA domain to allocate pages from, the time to process page allocation may vary significantly according to the status of the memory subsystem. When the system has free pages for a core in its per-process free page list, the page allocation can occur fast. However, if the list is empty, the page should be allocated by dividing a large memory chunk through the buddy system allocator, which is time-consuming. For these reasons, we opt to use the linear mapping scheme, rather than the RAM disk or the dynamic mapping scheme. Further, the ZNS SSD backend maintains the metadata to track the status of zones (i.e., open zone list and the write pointers within the open zones).

KVSSD. The page-level address mapping is sufficient for SSDs since the allocation unit is a fixed size, larger than (or equal to) the page size. However, generally most the keys and values in KVSSDs are much smaller than a single page (often tens to hundreds of bytes long), and their sizes are highly variable. This necessitates a proper memory management scheme to prevent/control the external fragmentation of the address space yet to provide a stable performance. Accordingly, we divide the first half of the reserved memory into 1 KiB chunks and the second half to 4 KiB ones. NVMeVirt maintains a bitmap to track the availability of each chunk.

KVSSD also requires an index for key-value pairs. For simplicity, we implemented it with a hash table. During the device initialization, we allocate a slice of memory and initialize it as a table. Each entry in the table contains the actual key and location of data as the chunk index. To process a key-value operation, the key is hashed with the Fowler-Noll-Vo hash function [38] to produce an integer index. This integer index is used as the index of the table, and the hash collision is handled with the linear probing scheme. Currently, the maximum size is set to 16-byte and 4 KiB for the key and value, respectively.

4 Evaluation

In this section, we provide the evaluation results to demonstrate the features and versatility of NVMeVirt. We aimed to discover the following with the evaluations;

1. What device types and their features does NVMeVirt provide?
2. How precisely can NVMeVirt emulate the performance of various storage devices, including off-the-shelf SSDs, NVM SSDs, ZNS SSDs, and key-value SSDs?
3. What type of advanced storage configurations can NVMeVirt support?
4. How can NVMeVirt contribute to storage-domain research?

4.1 Evaluation Setup

Evaluation Environment. To evaluate NVMeVirt, we used two identical servers with the same hardware and software configurations. Each server is equipped with two Intel Xeon Gold 6240 processors running at 2.60 GHz in a NUMA configuration. Each processor has 36 cores and 196 GiB of memory, which provides 72 cores and a total of 392 GiB of memory. The system is also equipped with commercial storage devices for performance comparison and analysis: Samsung 970 Pro SSD and Intel P4800X SSD based on the OptaneDC persistent memory technology. The devices are 512 GB and 350 GB in size, and represent an off-the-shelf high-end SSD and NVM SSD, respectively. We refer to them as “SSD” and “Optane” in the rest of the paper. To evaluate KVSSD, we used the Samsung KVSSD [23]. We also used a ZNS SSD provided by a company as an evaluation prototype. This ZNS SSD comprises 96 MiB zones that can be written only at 192 KiB units. The servers are also equipped with one Mellanox ConnectX-5 VPI HCA and connected through Mellanox SX6012 switch supporting 56 Gbps FDR bandwidth. We implemented NVMeVirt based on the Linux kernel 5.10.37, and the implementation takes approximately 9,000 lines of the kernel module code.

Configuration. To minimize the cross-interference between the applications and operations of NVMeVirt, we set them to run on different processors. Specifically, one processor (processor 1) is completely dedicated to NVMeVirt, whereas applications and benchmarks run on the other processor (processor 0). During the system initialization, the entire memory for the processor 1 is reserved with kernel booting parameters and used for storing data. The dispatcher and I/O workers are pinned down to different cores on the dedicated processor. The virtual PCIe bus is registered to the system as if it is attached to the processor 1. The memory for the NUMA

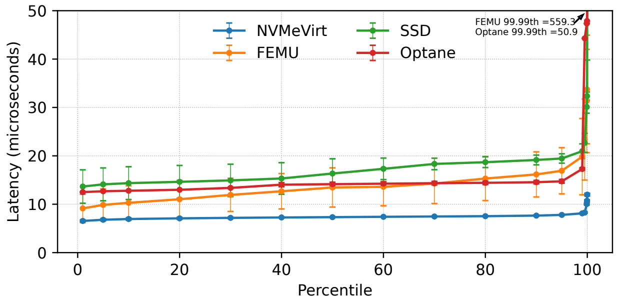


Figure 3: Performance variance of real and emulated devices for 4 KB random write operations

node 0, which is dedicated to applications, is configured to 32 GiB considering the size ratio between the memory and storage devices. In the evaluation, NVMeVirt is configured to use one I/O worker and a maximum of 72 queue pairs so that each core has its own queue pair. Note that we can easily change the maximum number of queue pairs of the virtual device.

4.2 Emulation Quality

NVMeVirt is primarily focusing on facilitating various NVMe device types, and FEMU [32] is the most relevant work sharing a similar goal. We compare the emulation quality by measuring the latency distribution of random write operations by repeating the same test 10 times. In each test, FIO writes 128 GiB of data with 4 KiB requests at random locations. We set FEMU and NVMeVirt to operate at the maximum speed without adding any latency to the incoming request. Figure 3 summarizes the percentile distributions of the 10 runs. The error bars indicate the standard deviation of the runs; thus, the longer the bars are, the more performance fluctuation the system exhibits.

Compared to the performance of real devices, NVMeVirt exhibits much lower latency with a very stable performance over the entire percentile range. These performance characteristics are very promising for emulating future storage devices. The FEMU-emulated device, however, barely meets the required performance to emulate modern storage devices. The maximum performance of FEMU is slightly faster than Optane. Hence, we are unable to utilize FEMU to project the performance implication of future low-latency storage devices. In addition, we observe high run-to-run performance variance from FEMU. Its standard deviations range from 28.7% to 39.7% of its average performance. The 99.99th percentile of FEMU goes off-the-chart, showing an average of 559.3 us and a standard deviation of 462 us. Considering the influences of the performance variance on the applications' tail latencies, FEMU will operate with a very high non-realistic tail latency.

Table 2: Presumed model parameters to emulate a real device performance.

Simple model		
	Optane	KVSSD
Page size	4 KiB	4 KiB
# of I/O units	1	10
Read latency	12 us	154 us
Read bandwidth	2.4 GiB	2.6 GiB
Write latency	14 us	56 us
Write bandwidth	2.0 GiB	1.3 GiB

Parallel model		
	SSD	ZNS
PCIe link bandwidth	3.6 GiB	3.3 GiB
# of NAND channels	8	8
NAND channel bandwidth	800 MiB	800 MiB
Dies per channel	2	16
Read unit size	32 KiB	64 KiB
NAND read time	36 us	40 us
Write unit size	32 KiB	192 KiB
NAND write time	185 us	1,913 us

4.3 Emulating a Real Device Performance

One of the primary goals of NVMeVirt is to emulate the performance of real devices. To verify this, we measured various performance metrics from real devices and configured NVMeVirt devices, as we discussed in 3.4. Table 2 summarizes the key parameters that we used for emulating the target devices. The values are obtained empirically from in-house microbenchmarks and device specification documents [16, 47, 50]. We used various benchmark tools and configurations for evaluating various device types. For Optane and the SSD, we used FIO [2] to measure the read and write latency while varying the request size from 4 KiB to 256 KiB. We also used FIO for measuring the performance of the ZNS SSD. We evaluated the read performance in the same manner. However, the ZNS SSD only allows 192 KiB writes to opened zones. Thus, we evaluated write performance with various numbers of threads that generate 192 KiB requests in accordance with the ZNS zone restriction. To evaluate KVSSD, we used OpenMPDK KVBench [46] which is an open-source benchmark based on the ForestDB benchmark suite [6]. We report the aggregated bandwidth from various payload sizes. In addition, we report the aggregated bandwidth measured with KVCeph [27] that generates realistic key-value operation workloads.

Figure 4 compares the performance of virtual devices to the real devices on various configurations. In each category, the values are normalized to the left-most entry value of the category (i.e., 4 KiB performance of the real device) Throughout the evaluation, we can confirm that the virtual devices provided by NVMeVirt faithfully reflect the configured target performance. We observe that the performance difference between the real and the virtualized devices is small. The

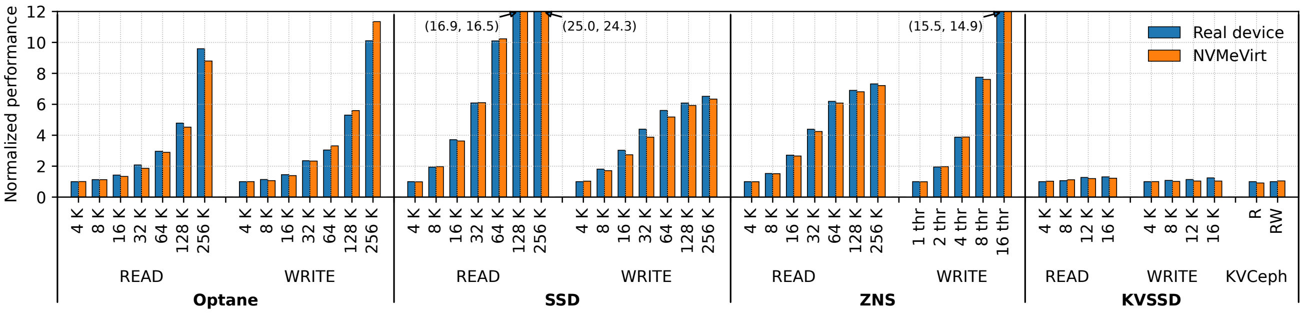
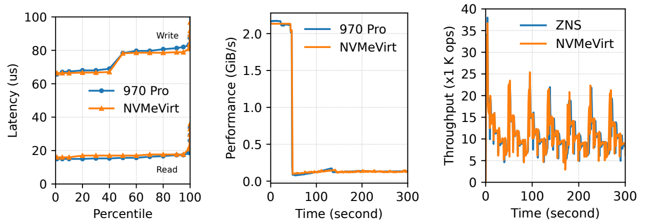
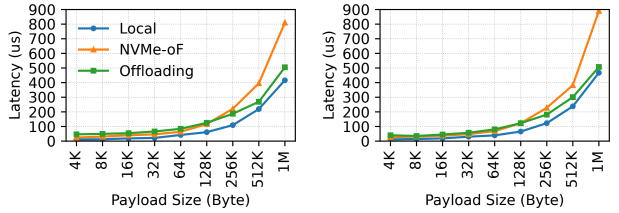


Figure 4: Performance comparison of the real devices and virtual devices. The performance is normalized to the left-most entry value of the category.



(a) Latencies (b) The effect of GC (c) RocksDB YCSB-A

Figure 5: The comparison of various performance characteristics



(a) Optane (b) NVMeVirt

Figure 6: The write performance as the NVMe-oF target

performance difference is by up to 12.2%, 11.8%, 3.3%, and 3.8% for Optane, SSD, ZNS SSD, and KVSSD, respectively.

Next, we analyzed various I/O characteristics. Firstly, Figure 5a summarizes the latency distribution for processing 16 KiB requests in Samsung 970 Pro and its NVMeVirt counterpart. We can see the similar latency distributions between the setups. Secondly, we evaluated the performance change when the garbage collection is performing. We built a microbenchmark that fills in the storage space with sequential write and keeps performing random writes. These writes will eventually trigger GC, which will cause performance drops. Figure 5b shows the performance change, and we can see that NVMeVirt exhibits the realistic performance change when GC is involved. Lastly, Figure 5c shows the performance of RocksDB running on ZNS SSD. We measured the throughput over a period of time while running the YCSB-A benchmark. We can observe repeated performance changes from ZNS SSD, and NVMeVirt can model the performance changes very closely. From these evaluations, we conclude that NVMeVirt can model various performance aspects of the target devices.

4.4 Supporting Various Storage Environments

In this evaluation, we elaborate on the versatility of NVMeVirt in various storage configurations. First, we demonstrate the feasibility of the NVMe-oF target offloading. We configured an NVMeVirt instance as the NVMe-oF target and measured their performance with the FIO benchmark. The NVMeVirt instance is configured to emulate Optane with the target performance listed in Table 2. Figure 6 compares their performance under various payload sizes and different NVMe-oF configurations. Note that 'NVMe-oF' indicates the performance of the default NVMe-oF configuration without the target offloading, and 'Offloading' is with the target offloading enabled. We present the results from write operations only since read operations showed the same trend.

We can observe that NVMeVirt emulates the performance of Optane over NVMe-oF closely. Specifically, the baseline configuration without the target offloading outperforms that with the target offloading when the payload is smaller than 128 KiB. We attribute the performance trend to the NVMe command processing overhead in the adapter that outweighs the performance gain from the optimized data path. However, when the payload is larger than 128 KiB, the performance gain outweighs the overhead, making the target offloading-enabled configuration show much lower latency.

We also demonstrate the PCI peer-to-peer DMA support for AI workloads. Specifically, GPUDirect Storage (GDS) is one of the promising techniques to accelerate GPU-intensive workloads [39, 40]. GPU directly accesses NVMe devices through PCI peer-to-peer DMA protocol, enabling decreased latency and CPU utilization on the host. We analyze the im-

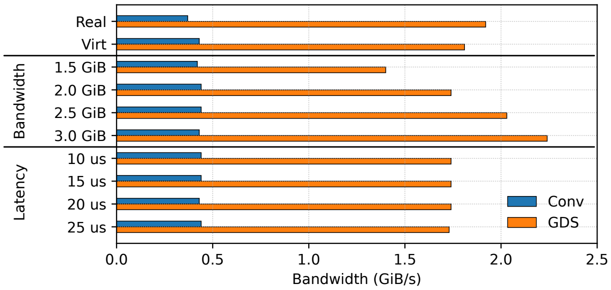


Figure 7: Checkpointing performance of Megatron DeepSpeed.

plication of storage performance on the GDS environment with NVMeVirt.

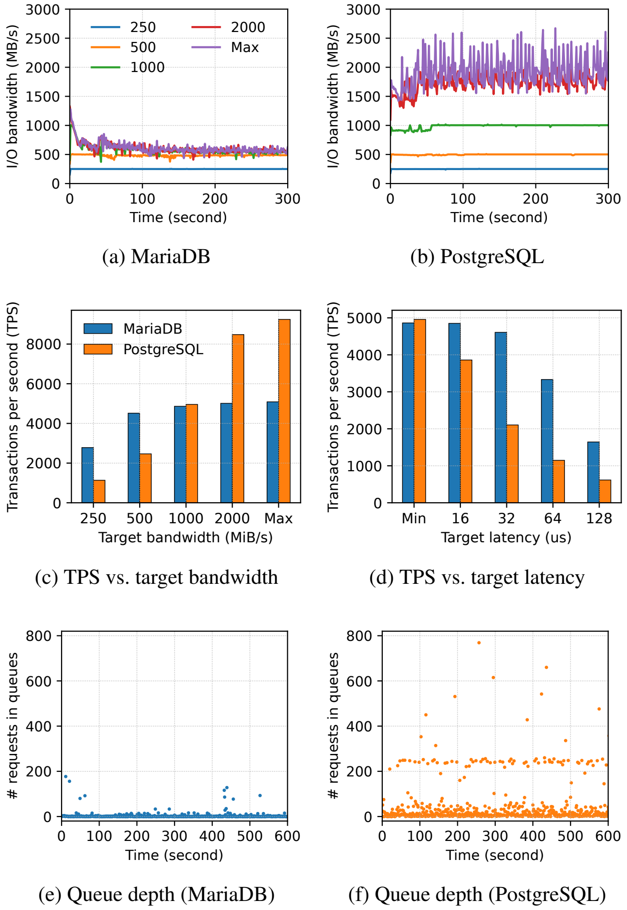
We measure the performance of checkpointing while running Megatron DeepSpeed [37] with NVIDIA A100 GPU. As shown in Figure 7, with Samsung 970 Pro SSD, the checkpointing occurs at the rate of 0.37 GiB/s with the conventional storage configuration, where checkpointing data is stored through the host's filesystem (the data is labeled with 'Real'). With GDS enabled, the checkpointing data goes to the NVMe device directly, showing a substantial performance gain of 5.2x. The NVMeVirt instance configured for the SSD exhibits the same performance trend, as labeled with 'Virt'.

We set up an NVMeVirt SSD instance in the same environment and evaluate the implication of storage device performance. We measured the checkpointing performance on various target bandwidths, but with a fixed latency of 10 us (grouped in 'Bandwidth'). We also evaluated the performance from various latencies while the target bandwidth is fixed to 2.0 GiB (grouped in 'Latency').

As shown in Figure 7, the storage performance does not much influence the AI application when the checkpointing occurs through the conventional storage configuration. The checkpointing performance remains consistent at a rate of 0.43 GiB/s regardless of changes in bandwidth and latency. However, the performance is affected by bandwidth in the GDS configuration. This confirms that the direct storage access is promising in that it can circumvent the inherent performance bottleneck in the conventional I/O path. Also, it implies that to fully exploit the reduced overhead through GDS, the storage performance should be improved further. From the consistent performance over a wide range of latencies, we can infer that the workload is likely to process I/O with a high queue depth.

4.5 Case for the Database Engine Analysis

To promote the tunable performance of NVMeVirt, we analyze the implication of storage performance on database engine performance. We selected MariaDB and PostgreSQL, two database engines that are very popular in the industry [7].



(a) MariaDB

(b) PostgreSQL

(c) TPS vs. target bandwidth

(d) TPS vs. target latency

(e) Queue depth (MariaDB)

(f) Queue depth (PostgreSQL)

Figure 8: Performance characteristics of the MariaDB and PostgreSQL database engines on various storage configurations

We created an NVMeVirt instance configured as an NVM SSD, and configured the database instance on it with recommended configurations from optimization tools [13, 43]. The database instance is then populated with sysbench [48] to have 10 tables of 50,000,000 bytes in size, taking approximately 120 GiB of space in total. Then we run the OLTP workload with sysbench with 72 threads for 60 minutes. We measured various performance metrics while running the benchmark, and Figure 8 summarizes the results. We only report the trends of the first 5 minutes since the performance became stable afterwards.

Overall, we verified that MariaDB and PostgreSQL react very differently to the storage performance. Figure 8a and 8b compare the I/O bandwidth utilization of MariaDB and PostgreSQL over time when the target bandwidth is set to the given value (i.e., 250 implies the storage bandwidth is limited to 250 MiB/s). In this evaluation, the I/O latency was set to a minimum (i.e., does not impose any delay while processing I/O operations). For MariaDB, it fully utilizes the I/O bandwidth up to 500 MiB/s, but does not utilize it further. Even though the storage device provides a higher bandwidth, the I/O bandwidth utilization remains low, approximately at 600 MiB/s. On the other hand, PostgreSQL fully utilizes the I/O bandwidth up to 1,000 MiB/s and exhibits a saturated performance at around 1,800 MiB/s. This hints that PostgreSQL is designed to utilize the storage device more eagerly than MariaDB. However, this does not necessarily mean that PostgreSQL outperforms MariaDB. Figure 8c compares the processing performance measured in transactions per second (TPS) on various bandwidth limits. The I/O bandwidth limit influences both database engines, but PostgreSQL is much more sensitive than MariaDB. Specifically, MariaDB exhibits a higher TPS than PostgreSQL when the bandwidth is low, but the TPS is not improved much when the device has more bandwidth. Meanwhile, PostgreSQL shows a lower performance when the bandwidth is low. However, the performance improves as the device supports more bandwidth. When the bandwidth limit is low, MariaDB outperforms PostgreSQL by 2.45x at 250 MiB/s bandwidth limit. However, PostgreSQL outperforms MariaDB by 1.82x at the unlimited bandwidth.

The engines exhibit a similar trend with respect to the latency. Figure 8d compares the performance when the bandwidth is fixed to 1,000 MiB/s, and both read and write latencies are set to the values on the y-axis. When the device exhibits a high latency, MariaDB outperforms PostgreSQL by up to 2.67x when the latency is 128 us. However, as the latency decreases, the TPS of PostgreSQL keeps increasing, becoming comparable to that of MariaDB when the latency is minimum. Figure 8e and 8f shows the number of queued requests in the submission queues of the device over time. The device is configured to have the minimum latency and a target bandwidth of 2.0 GiB. MariaDB operates with a low queue depth, whereas PostgreSQL utilizes a higher queue depth with a noticeable unique pattern near qd=200.

From the evaluation, we can conclude that PostgreSQL is more promising on modern storage devices, whereas MariaDB is more efficient when the storage is slow. We can verify that NVMeVirt allows us to estimate the performance of applications on future storage devices.

4.6 Case for NVMe Interface Study

As NVMeVirt handles inbound NVMe operations in software, it opens up the opportunity to extend the host-device interface easily. To demonstrate this, we made a case with one of the recent studies whose evaluation is limited by the host-device interface modification. Specifically, Kim et al. [26] proposed to extend the NVMe command set so that one NVMe command can batch multiple key-value operations, thereby amortizing the interface overhead for small key-value operations. To realize this so-called 'compound command' concept, the KVSSD firmware should be modified to understand and process the extended NVMe command. However, the authors were unable to modify the firmware and ended up estimating the performance gain from the single operation performance.

We attempted to verify the benefit of the compound command by using the KVSSD instance with NVMeVirt. Specifically, we modified the KVSSD backend to understand the compound command and process packed operations in a batch. Each I/O operation in a compound command is processed as an individual key-value operation in the backend. This modification took less than a week for one of the authors, and we argue that this manifests the advantage of the software-level NVMe abstraction that NVMeVirt provides. To evaluate the performance, we built a user-level microbenchmark tool that builds the compound command with multiple requests, and submits the command to the device through the NVMe pass-through interface. Note that the extended KVSSD backend uses the same performance model and configuration as explained in Section 4.3.

From the evaluation, we can verify the significant performance improvement with the compound command. Without the compound command, processing eight 4 KiB key-value put operations takes approximately 469 us, which is reduced to 86 us with the compound command, giving a 5.4x performance gain. This improvement is higher than the value reported in the original work (92.0 us to 41.5 us), and we attribute the extra improvement to the conservative estimation in the original work.

5 Conclusion

We presented NVMeVirt, a virtual, software-only NVMe device. It opens a new opportunity for developers to co-design highly intelligent storage devices over the NVMe interface. With NVMeVirt, we demonstrated the usefulness of NVMeVirt for storage research.

Currently, NVMeVirt only supports a single instance of one device type. We are planning to support multiple NVMe device instances with various storage device types. Furthermore, we are working on a separate study to present the in-depth methodology and analysis to model the performance characteristics of various NVMe-based SSDs using NVMeVirt.

Acknowledgments

We would like to thank our shepherd, Robert Ross, and anonymous reviewers for their invaluable feedback. We also thank Hyeong-Jun Kim who developed the initial prototype of NVMeVirt. This work was supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (23ZS1310), the National Research Foundation of Korea (NRF) grant (No. 2019R1A2C2089773), and Institute of Information & communications Technology Planning & Evaluation (IITP) grant (No. IITP-2021-0-01363) funded by the Korea government (MSIT).

References

[1] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: Efficient IOMMU emulation. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, Portland, OR, June 2011.

[2] Jens Axboe. fio: flexible I/O tester. https://github.com/axboe/fio.

[3] Jonghyun Bae, Jongsung Lee, Yunho Jin, Sam Son, Shine Kim, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. FlashNeuron: SSD-enabled large-batch training of very deep neural networks. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, February 2021.

[4] 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 2021 USENIX Annual Technical Conference (ATC)*, Virtual, July 2021.

[5] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 359–375, Santa Clara, California, USA, February–March 2017.

[6] Couchbase Labs. ForestDB benchmark. https://github.com/couchbaselabs/ForestDB-Benchmark.

[7] DB-Engines ranking. https://db-engines.com/en/ranking.

[8] Jaeyoung Do, Victor C. Ferreira, Hossein Bobarshad, Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Diego Souza, Brunno F. Goldstien, Leandro Santiago, Min Soo Kim, Priscila M. V. Lima, M. G. França, and Vladimir Alves. Cost-effective, energy-efficient, and scalable storage computing for large-scale AI applications. *ACM Transactions on Storage*, 16(4):1–37, 2020.

[9] NVM Express. NVMe-oF specification. https://nvmexpress.org/developers/nvme-of-specification/.

[10] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber: Enabling precise full-system simulation with detailed modeling of all SSD resources. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–481, Fukuoka, Japan, October 2018.

[11] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 153–165, Seoul, South Korea, June 2016.

[12] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: advanced zoned namespace interface for supporting in-storage zone compaction. In *Proceedings of the 15nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual, July 2021.

[13] Major Hayden. MySQLTuner. https://github.com/major/MySQLTuner-perl.

[14] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. PinK: High-speed in-storage key-value store with bounded tails. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 173–187, Virtual, July 2020.

[15] Intel. Intel I/O acceleration technology. https://www.intel.co.kr/content/www/kr/ko/wireless-network/accel-technology.html.

[16] Intel. Intel Optane SSD 9 series. https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html.

[17] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module.

[18] Mike Jackson and Ravi Budruk. *PCI Express Technology*. MindShare Technology, 2012.

[19] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A flexible, high-performance key-value SSD. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384, Austin, TX, USA, February 2017. IEEE.

[20] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. GraFBoost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45rd ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Los Angeles, CA, June 2018.

[21] Myoungsoo Jung. OpenExpress: Fully hardware automated open research framework for future fast nvme devices. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, Virtual, July 2020.

[22] Myoungsoo Jung, Jie Zhang, Ahmed Abulila, Miryeong Kwon, Narges Shahidi, John Shalf, Nam Sung Kim, and Mahmut Kandemir. SimpleSSD: Modeling solid state drives for holistic system simulation. *IEEE Computer Architecture Letters*, 17:37–41, September 2017.

[23] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, page 144–154, Haifa, Israel, 2019.

[24] 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 *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Denver, CO, June 2016.

[25] Joonsung Kim, Kanghyun Choi, Wonsik Lee, and Jangwoo Kim. Performance modeling and practical use cases for black-box SSDs. *ACM Transactions on Storage*, 17(2), jun 2021.

[26] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. Transaction support using compound commands in key-value SSDs. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Renton, WA, July 2019.

[27] Open memory platform development kit. http://github.com/OpenMPDK.

[28] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ OpenSSD: Rapid prototype for flash storage systems. *ACM Transactions on Storage*, 16(3), aug 2020.

[29] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. Hardware/software co-programmable framework for computational SSDs to accelerate deep learning service on large-scale graphs. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2022.

[30] Parallel Data Lab. The DiskSim simulation environment v4.0. https://www.pdl.cmu.edu/DiskSim/index.shtml.

[31] Young-Sik Lee, Luis Cavazos Quero, Sang-Hoon Kim, Jin-Soo Kim, and Seungryoul Maeng. ActiveSort: Efficient external sorting using active SSDs in the MapReduce framework. *Future Generation Computer Systems*, 65:76–89, December 2016.

[32] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The case of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018.

[33] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[34] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 257–270, Santa Clara, California, USA, February 2013.

[35] Krishna T. Malladi, Manu Awasthi, and Hongzhong Zheng. FlexDrive: A framework to explore NVMe storage solutions. In *Proceedings of the 2016 IEEE 18th International Conference on High Performance Computing and Communications*, page 1115–1122, Dec 2016.

[36] Krishna T. Malladi, Mu-Tien Chang, Dimin Niu, and Hongzhong Zheng. FlashStorageSim: Performance modeling for SSD architectures. In *Proceedings of the 2017 International Conference on Networking, Architecture, and Storage (NAS)*, page 1–2, August 2017.

[37] Microsoft. Megatron-DeepSpeed. https://github.com/microsoft/Megatron-DeepSpeed.

[38] Landon Curt Noll. FNV hash. http://isthe.com/chongo/tech/comp/fnv/.

[39] NVIDIA. GPUDirect Storage: A direct path between storage and GPU memory. https://developer.nvidia.com/blog/gpudirect-storage/.

[40] NVIDIA. The NVIDIA Magnum IO GPUDIrect Storage overview guide. https://docs.nvidia.com/gpudirect-storage/overview-guide/index.html.

[41] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th ACM International Conference*

on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Salt Lake City, Utah, USA, March 2014.

[42] PCI SIG. PCI SIG. https://pcisig.com.

[43] PGTune: configuration for PostgreSQL based on the maximum performance for a given hardware configuration. https://pgtune.leopard.in.ua/.

[44] QMEU. QEMU: A generic and open source machine emulator and virtualizer. https://qemu.org.

[45] Manoj P. Saha, Adnan Maruf, Bryan S. Kim, and Janki Bhimani. KV-SSD: What is it good for? In *Proceedings of the 58th ACM/IEEE Annual Design Automation Conference (DAC)*, pages 1105–1110, 2021.

[46] Samsung Electronics. OpenMPDK KVSSD. https://github.com/OpenMPDK/KVSSD/.

[47] Samsung Electronics. Samsung Enterprise SSD. https://www.samsung.com/semiconductor/ssd/enterprise-ssd/.

[48] Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.

[49] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: a framework for enabling realistic studies of modern multi-queue SSD devices. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018.

[50] Tech Power Up. Samsung 970 Pro 512 GB. https://www.techpowerup.com/ssd-specs/samsung-970-pro-512-gb.d54.

[51] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An early evaluation of intel's optane DC persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, SC '19, November 2019.

[52] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carolejean Wu, David Michael Brooks, and Guyeon 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 (ASPLOS)*, Virtual, April 2021.

[53] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of intel optane SSD. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, June 2019.

[54] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. SPDK: a development kit to build high performance storage applications. In *Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, 2017.

[55] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, and Jaehyuk Cha. VSSIM: Virtual machine based SSD simulator. In *Proceedings of the 29th IEEE Conference on Massive Data Storage (MSST)*, May 2013.

[56] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, California, USA, February 2012.

SMRSTORE: A Storage Engine for Cloud Object Storage on HM-SMR Drives

Su Zhou, Erci Xu*, Hao Wu, Yu Du, Jiacheng Cui, Wanyu Fu, Chang Liu, Yingni Wang, Wenbo Wang, Shouqu Sun, Xianfei Wang, Bo Feng, Biyun Zhu, Xin Tong, Weikang Kong, Linyan Liu, Zhongjie Wu, Jinbo Wu, Qingchao Luo, Jiesheng Wu

Alibaba Group

Abstract

Cloud object storage vendors are always in pursuit of better cost efficiency. Emerging Shingled Magnetic Recording (SMR) drives are becoming economically favorable in archival storage systems due to significantly improved areal density. However, for standard-class object storage, previous studies and our preliminary exploration revealed that the existing SMR drive solutions can experience severe throughput variations due to garbage collection (GC).

In this paper, we introduce SMRSTORE, an SMR-based storage engine for standard-class object storage without compromising performance or durability. The key features of SMRSTORE include directly implementing chunk store interfaces over SMR drives, using a complete log-structured design, and applying guided data placement to reduce GC for consistent performance. The evaluation shows that SMRSTORE delivers comparable performance as Ext4 on the Conventional Magnetic Recording (CMR) drives, and can be up to 2.16x faster than F2FS on SMR drives. By switching to SMR drives, we have decreased the total cost by up to 15% and provided performance on par with the prior system for customers. Currently, we have deployed SMRSTORE in standard-class Alibaba Cloud Object Storage Service (OSS) to store hundreds of PBs of data. We plan to use SMR drives for all classes of OSS in the near future.

1 Introduction

Object storage is a “killer app” in the cloud era. Users can use the service to persist and retrieve objects with high scalability, elasticity and reliability. Typical usage scenarios of object storage include Binary Large OBjects (BLOBs) storage [10,23], datalake [2] and cloud archive [1]. Object storage systems usually employ a large fleet of HDDs. Therefore, a key challenge of building a competitive cloud object storage is the cost efficiency.

Emerging Shingled Magnetic Recording (SMR) drives [5] are economically attractive [29] but they may not serve as a simple drop-in replacement for traditional CMR drives [12]. SMR drives, via overlapping tracks, have a higher areal density [14] (i.e., 25% more than CMR drives) and hence the better cost efficiency. However, shingling tracks has a byproduct—not allowing random writes [8,15]. This characteristic in return may require the upper-level software stack, such as storage engines, to make the corresponding adaptions.

A possible direction is to use Host Managed SMR (HM-SMR) drives where the host OS manages the I/Os and communicates with HM-SMR drives via the Zoned Block Device (ZBD) subsystem [6,7]. There are mainly three types of approaches in designing HM-SMR-based storage systems. First, Linux kernel can expose HM-SMR drives as standard block devices by employing a shingled translation layer (STL), such as dm-zoned [21]. Second, file systems with a log-structured [28] or copy-on-write design, can directly support HM-SMR drives(e.g., F2FS [17] and Btrfs [27]). Further, developers can modify their applications to accommodate HM-SMR drives (e.g., GearDB [32], SMORE [19], and SM-RDB [26]) or directly employ them in archival-class object storage systems such as Alibaba Archive Storage Service [1] and Huawei Object Store [18].

Unfortunately, these existing HM-SMR solutions can not be applied to standard-class Alibaba Cloud Object Storage Service (OSS) . First, setting the HM-SMR drive as a block device (i.e., via dm-zoned [21]) could suffer a significant throughput drop due to frequent buffer zones reclaiming after random updates (e.g., a 56.1% drop under a sustained write workload [22]). Second, employing log-structured file systems to manage HM-SMR drives can experience throughput variations due to GC in file systems. For example, our evaluation shows that the throughput of F2FS on a HM-SMR drive can drop 61.5% due to frequent F2FS GCs triggered by random deletions. Third, though archival-class and standard-class OSS share the same data abstraction (i.e., object), they have drastically different Service Level Objectives (SLOs). Therefore, a design that works well in the archival class may not deliver satisfying performances in the standard class.

Our benchmarks show that existing SMR translation layers or file systems could result in severe overhead possibly due to garbage collection. Moreover, log-structured design offers direct support to SMR drives and could achieves a high throughput when not affected by GC. Besides, GC, which is inevitable due to the append-only nature of SMR zones, could be alleviated via workload-aware data placement.

In this paper, we describe SMRSTORE, a high-performance HM-SMR storage engine co-designed with Alibaba Cloud

*Corresponding author. erc.xec@alibaba-inc.com

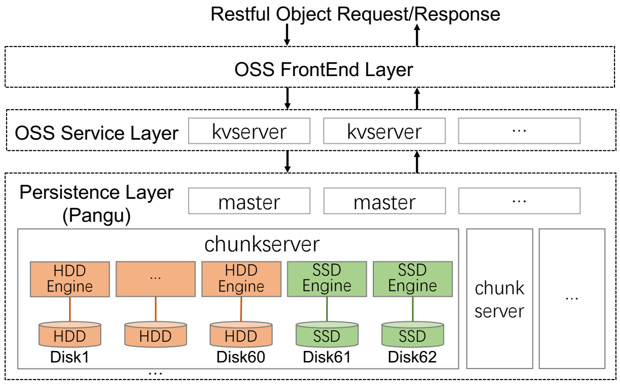


Figure 1: Architecture overview of OSS (§2.1). *The red shaded HHD engines refer to traditional ext4-based storage engines. The green shaded SSD engines refer to user-space storage engines.*

standard-class OSS. There are three key features in SMRStore. First, SMRStore is a user-space storage engine that does not require local file system support and directly implements chunk interfaces of Pangu distributed file system.

Second, SMRStore strictly follows a log-structured design to organize HM-SMR on-disk layout. In SMRStore, the basic building block is a variable-length customized log format called *Record*. We use records to persist data, and form various metadata structures (e.g., checkpoint and journal).

Third, we design a series of workload-aware zone allocation strategies to reduce the interleaving of different types of OSS data & metadata in zones. These effort help us to effectively lower the overhead of GC in HM-SMR drives.

We extensively evaluate SMRStore under various scenarios. The results show that Pangu chunkserver with SMRStore achieves more than 110MB/s throughput in high concurrent write workloads, 30% higher than the previous generation design (i.e., chunkserver with Ext4 on CMR drives). Moreover, on a storage server (60 HDDs and 2 cache SSDs), chunkserver with SMRStore provides steady 4GB/s write throughput in macro benchmarks, 2.16x higher than F2FS. Third, in OSS deployment, the performances of the HM-SMR cluster are comparable to the CMR cluster in all aspects.

The rest of the paper is organized as follows. We describe standard-class OSS in Alibaba and the HM-SMR drives in §2. Then, we analyze the pros and cons of existing solutions (§3). Further, we demonstrate the design choices (§4), the detail implementation of SMRStore (§5) and the evaluation (§6). We conclude with discussions on the limitation of SMRStore (§7), the related work (§8) and a short conclusion (§9).

2 Background

2.1 Alibaba Cloud OSS

Alibaba Cloud OSS offers four classes of services, including standard, infrequent access, archive, and cold archive (prices in descending order and retrieval time in ascending order).

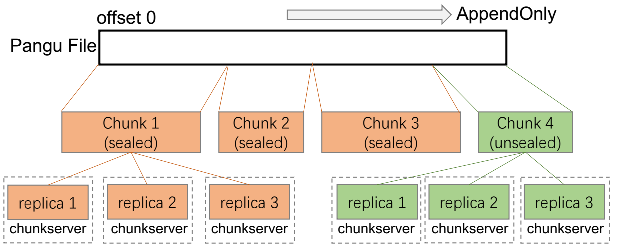


Figure 2: Semantics of Pangu file and chunk. *This figure shows a Pangu file consists of four chunks. Only chunk 4 (the last chunk) is not sealed (writable). Pangu keeps multiple replicas for each chunk across chunkservers to protect data against any failures.*

Standard-class OSS, usually for hosting hot data, offers the fastest SLOs with the highest economical cost.

Architecture. Figure 1 illustrates the three layers in Alibaba Cloud standard-class OSS stack, including an OSS frontend layer, an OSS service layer, and a persistence layer. The frontend layer pre-processes users' http requests and dispatches them to the service layer. The service layer, consisting of multiple KV servers, has two functionalities. First, the service layer writes the objects to Pangu files. Second, the service layer maintains the objects' metadata (the mapping from objects' names to locations within the corresponding Pangu files) using an LSM-tree based KV store [25], and writes these metadata to additional Pangu files. The persistence layer is our distributed file system Pangu.

Pangu overview. Pangu is a HDFS-like distributed file system and each Pangu cluster comprises a set of masters (handling Pangu files' metadata, not objects' metadata) and up to thousands of chunkservers (storing data of Pangu files). Each chunkserver exclusively owns a physical storage server to operate, consisting of 60 HDDs for persistence and two high performance SSDs for caching [1]. We leverage Linux kernel storage stack (Ext4 file system and libaio with O_DIRECT) for the HDD storage engines and build a user-space storage file system for the SSD engines.

Pangu data abstractions. Figure 2 illustrates two levels of abstractions in Pangu, file and chunk. Each file is append-only and can be further split into multiple chunks. Each chunk has a Chunk ID (a 24-byte UUID) and is replicated via copies or erasure coding. Pangu can create, write(append), read, delete, and seal a chunk. Similar to the "extent seal" in Windows Azure Storage [13], Pangu seals a chunk when: i) the size of chunk—including data and corresponding checksum—reaches the limit; ii) the application closes the Pangu file when writing is finished; iii) in the face of failures (e.g., network timeout). Due to case ii) and iii), the chunks can be of variable sizes. Note that only the last chunk of a Pangu file can be appended (not sealed) and only sealed chunks can be

[1] Pangu also supports other services (e.g., block storage and big data) and can have various modifications. In this paper, our discussion on Pangu and corresponding software/hardware setups only apply to OSS scenario.

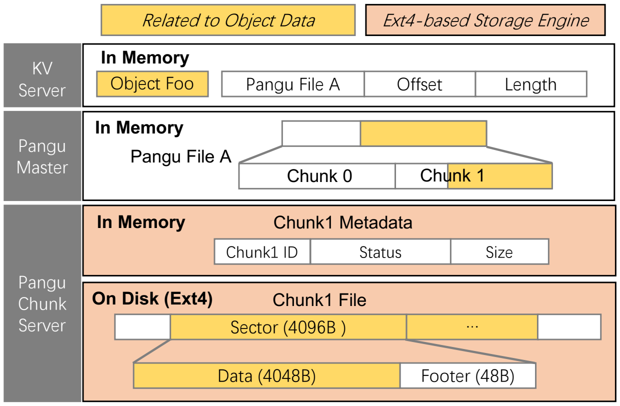


Figure 3: Dataflow with traditional storage engine (OSS (§2.1). *This figure illustrates the write path of an object in traditional CS-Ext4 stack (red shaded). The yellow shaded area means it is related to the object "Foo". The data is split as a series of 4048B segments where each is attached with a 48B footer for checksum.*

flushed from cache SSDs to HDDs for persistence. Storage engine does not provide any redundancy for failures. Instead, we rely on PANGU providing fault tolerance for each chunk across chunkservers by replication or erasure coding.

I/O Path. Figure 3 presents high-level write flows in OSS with the traditional Ext4-based storage engine. We use an example object called "Foo" to highlight the write flow. The KV server chooses the PANGU file *A* for storing "Foo". Then the KV server uses the PANGU SDK or contacts the PANGU master to locate the tail chunk (i.e.,chunk-1) and its respective chunkservers. Further, the chunkserver appends the object's data (with checksums) to the corresponding Ext4 file. To verify data integrity, we break the object's data into a series of 4048-Byte segments. We attach to each segment a 48 Byte footer which includes the checksum and segment locations in chunk. Note that each chunk in the chunkserver is an Ext4 file with the chunk ID as its filename.

Workloads. A KV server can open multiple PANGU files to store or retrieve the data and metadata of objects, and perform GC on deleted objects in PANGU files. From the perspective of chunkservers, we define an active PANGU file as a **stream**. The stream starts as the PANGU is opened by a KV server for read or write, and ends when the KV closes the PANGU file. Based on the operations (read or write) and types (metadata, data or GC), we can categorize the workloads issued by KV servers as five types of streams. Table 1 lists the characteristic of each type of streams. The "Concurrency" refers to the PANGU file concurrency, namely the number of PANGU opened files on a chunkserver to append data. The "Lifespan" refers to the expected lifespan of the data on disk (from being persisted to deleted), NOT the duration of the streams.

Type	Latency (ms)	Concur-rency	iosize (Byte)	Lifespan (Day)
OSS Data W	<1	~1500	512K-1M	<7
OSS Data R	<20	-	512K-1M	-
OSS Meta W	<1	~2000	4K-128K	<60
OSS Meta R	<20	-	4K-128K	-
OSS GC	<20	~100	512K-1M	<90

Table 1: Characteristics of streams on a chunkserver(§2.1).

- *OSS Data Write Stream.* Persisting object data requires low latency to achieve quick response. Therefore, object data are first written to SSD caches and later moved to HDDs. Normally, hundreds of PANGU chunks are opened for writing on a chunkserver, thereby yielding high concurrency.
- *OSS Data Read Stream.* OSS directly reads object data from HDDs to achieve high throughput. Due to space limits, object data for read are not cached in the SSDs.
- *OSS Metadata Write Stream.* OSS metadata stream includes objects' metadata formatted as Write-Ahead Logs and Sorted String Table Files from KV store. The metadata are first flushed to SSD cache and then migrated to HDDs. The metadata accounts for around 2% of the total capacity used (around 24TB per chunkserver).
- *OSS Metadata Read Stream.* The KV server maintains a cache for object index. In most cases, the metadata read directly hits the KV index cache and returns. If cache misses, OSS routes to SSTFiles in PANGU.
- OSS *GC Stream.* The garbage collection in OSS service layer (referred to as OSS GC) is to reclaim garbage space in PANGU files. A PANGU file can hold multiple objects. When a certain amount of objects are deleted in a PANGU file, OSS would re-allocate the rest to another PANGU file, and delete the old file. The chunks written by OSS GC streams account for more than 80% of the total capacity used in one chunkserver. OSS GC streams run in background and directly routed to HDDs for persistence.

2.2 Host-Managed HM-SMR

HM-SMR drives overlap the tracks to achieve higher areal density but consequently sacrifices random write support. Specifically, HM-SMR drives organize the address space as multiple fixed-size zones including sequential zones (referred to as zones) and a few (around 1%) conventional zones (referred to as czones). For example, the Seagate SMR drive we use in this paper has a capacity of 20TB and 74508 zones (including 800 czones). The size of each zone is 256MB, and it takes around 20ms, 24ms and 22ms for opening, closing and erasing a zone, respectively. Note that, for certain SMR HDD models (e.g., West Digital DC HC650), there is a limit on the number of zones to be opened concurrently.

Device mapper translation. A straightforward solution is to insert a shim layer, called shingled translation layer (STL), such as dm-zoned [21], to provide dynamic mapping from logical block address to physical sectors and hence achieve random-to-sequential translation. Apparently, the major advantage of this approach is allowing the users (e.g., chunkserver process) to adopt the HM-SMR drives as cost-efficient drop-in replacement for CMR drives.

SMR-aware file systems. The log-structured design file systems (e.g., F2FS) make them an ideal match for the append-only zone design of HM-SMR disks. For example, F2FS started to support zoned block devices since kernel 4.10. Users can mount a F2FS on an HM-SMR drive and utilize the F2FS GC mechanism to support random writes. Similarly, Btrfs, a file system based on copy-on-write principle, currently provided an experimental support for zoned block device in kernel 5.12.

End-to-End Co-design. Instead of relying on dm-zoned or general file systems, applications that perform mostly sequential writes can be modified to adopt HM-SMR. The benefits of end-to-end integration has been proved by several recent works, such as GearDB [32], ZenFs [11], SMORE [19], etc. Applications could eliminate the block/fs-level overhead and achieve predictable performance by managing on-disk data placement and garbage collection at application level [24,30].

3 Evaluating Existing Solutions

We evaluate running F2FS atop HM-SMR drives with microbenchmark and macrobenchmark (i.e., simulated OSS workloads). We compare the performances of chunkservers with Ext4 on CMR drives (referred to as CS-Ext4) and F2FS on HM-SMR drives (referred to as CS-F2FS).

3.1 Evaluation Configurations

	CMR Server	SMR Server
OS	Linux 4.19.91	
CPU	2*Intel(R) Xeon(R) Platinum 8331C CPU@2.50GHz 48 Physical Cores 96 Threads	
SSD	2*INTEL SSDPF21Q800GB	
Mem	512G	

HDD	60*ST16000NM001G-2KK103
	Rand. 4KB(IOPS): 113
	Seq. 512KB(MB/s):
	254.8(W) 254.5(R)
		60*ST20000NM001J-2U6101
		Rand. 4KB(IOPS): 121
		Seq. 512KB(MB/s):
		255.7(W) 255.6(R)

Table 2: Configurations of storage servers in evaluation. *A SMR server has the exact same setups with a CMR server, except the HDDs are 20TB SMR HDDs instead. The raw performance comparison with queue depth 1 random read and queue depth 32 sequential read/write is listed in the last row.*

Environment Setup. Table 2 lists the configurations of the storage servers in the evaluation. Note that F2FS does not support devices with a capacity larger than 16TB. Therefore, we format the disk with 6TB capacity. Moreover, in all cases we disable the disk write cache by hdparm [4] tool to prevent data loss upon crashes, a mandatory setting in OSS.

Workloads. For both micro- and macro-benchmarks, we use the Fio [3] (modified to use the PANGU SDK) as the workload generator. For microbenchmark, we start a chunkserver with one disk, and focus on testing write throughput with different I/O sizes in the clean state (no F2FS GC). We start a Fio with 4 numjobs, 4 iodepth, and 128 nrfiles to simulate a high write pressure.

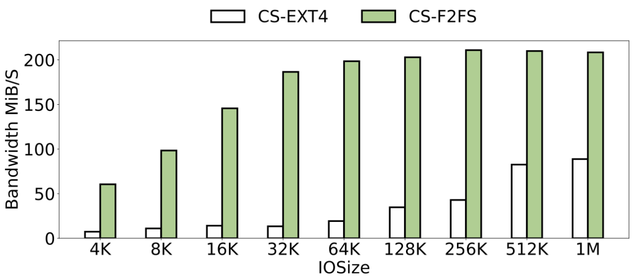


Figure 4: High Concurrency Write Throughput (§3.2). *The figure shows the write throughput of CS-Ext4 and CS-F2FS in microbenchmarks.*

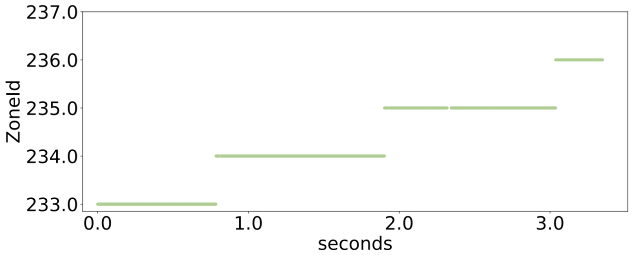


Figure 5: F2FS Access Pattern (§3.2). *The figure shows the accessed zoneIds by F2FS in a few seconds. F2FS writes all data into one zone in a period of time and switches to the next only when the zone is full.*

For macrobenchmark, we evaluate chunkserver with all disks loaded (60 HDDs and 2 cache SSDs) and run four Fio processes to simulate different types of write streams. Table 3 lists the detailed configurations. Note that we use two Fio processes to simulate two kinds OSS GC streams (i.e., OSS GC Wr 1 and 2). For OSS GC Wr 2, we use a smaller chunk size and rate (64MB and 20MB/s) to simulate the situations where the chunks are sealed before reaching the size limit (due to reaching the end of PANGU file or encountering I/O failures).

The macrobenchmark generates a stable 4GB/s throughput to simulate a typical high pressure workload. There are two phases in this test. In the first phase, we simply let the four streams to fill the HDDs and there is no file deletions. In the second phase, the utilization of capacity reaches around 80% (around 12 hours after the first phase started) and triggers the random deletions to maintain the utilization rate at around 80%. The average chunk deletion rate on a chunkserver ranges from 4 operations per second (ops/s) to 15 ops/s.

Stream Type	#Fio	Target	numjobs	iodepth	iosize	nrfiles	chunk size	rate
OSS GC Wr 1	1	HDDs	8	32	1MB	25	256MB	400MB/s
OSS GC Wr 2	1	HDDs	8	32	1MB	25	64MB	20MB/s
OSS Data Wr	1	SSDs	3	32	1MB	300	256MB	200MB/s
OSS Meta Wr	1	SSDs	1	8	4KB-128KB	500	4MB	80MB/s

Table 3: Macro benchmark setups (§3.1). OSS *GC Wr 1 refers to* OSS *GC streams with large chunks.* OSS *GC Wr 2 refers to* OSS *GC streams with small chunks.* OSS *Data Wr refers to* OSS *object data write streams.* OSS *Meta Wr refers to* OSS *metadata write streams.*

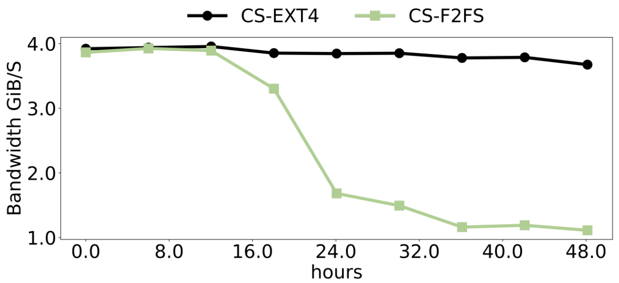


Figure 6: CS-Ext4 vs CS-F2FS in macrobenchmark (§3.2). *The test starts on empty disks and with steady 4GB/s throughput. At hour 12, the capacity utilization reaches 80% and random deletions occur.*

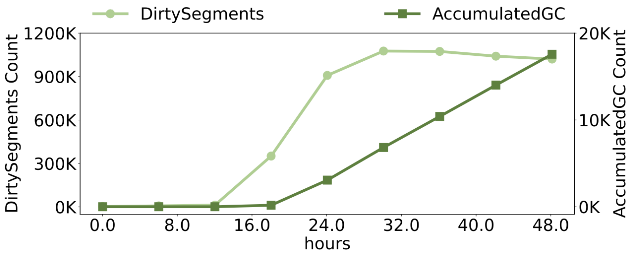


Figure 7: F2FS GC related metrics (§3.2.) *This figure illustrates the status of F2FS. The dirty segment count on the left axis reflects the generation of garbage space. The increasing accumulated GC count (right Y axis) indicates the continuing GC activities which are the immediate causes of the performance drop.*

3.2 Performance Comparison

Microbenchmark Performance. Figure 4 shows that CS-F2FS on HM-SMR drives achieves 1.3x - 12.9x higher throughput compared to CS-Ext4 on CMR drives. This is because F2FS writes from different streams to one zone at a time and thus always performs sequential writes. Figure 5 shows the accessing distribution of SMR ZoneIDs from the CS-F2FS. We can see F2FS fills up one SMR zone at a time (e.g., Zone 233 from second 0). This allocation strategy avoid overhead from jumping between zones.

Macrobenchmark performance. Figure 6 shows the throughput performances of CS-F2FS and CS-Ext4 along time. Initially, we can observe that both maintain stable throughput from hour 0 to 12. Then, after 12 hours, the CS-F2FS quickly drops and remains a low throughput for the rest of the time. This is because the random deletion starts and GC in F2FS kicks in to handle the increasing amount of obsolete data (see Figure 7). Note that F2FS puts chunks from different types of streams into one zone. Due to random deletions, severe F2FS GC can be frequently triggered and influence the OSS metadata/data streams, resulting in a performance drop.

We are aware that F2FS provides multi-head logging to separate streams on disk, but this technique cannot separate chunks from the same type of streams. In practice, Pangu file concurrency in each type of OSS stream can range from tens to hundreds, and F2FS would write all the chunks from the same type of stream into one zone. Therefore, random deletions on those chunks (a common scenario in standard-class OSS) still trigger severe F2FS GCs.

4 SMRStore Design Choices

No local file system. We build SMRStore to support chunk semantics (including `chunk_create`, `chunk_append`, `chunk_read`, `chunk_seal`, and `chunk_delete`) on SMR zoned namespace. There are three functionalities in SMRStore to support this feature. First, SMRStore directly manages the disk address space for persisting metadata (i.e., checkpoints and journalings) and data (i.e., the chunks). Second, SMRStore manages a mapping table between chunks and SMR zones to translate logical range in chunks (via chunkId, offset, and length) to the physical locations on disk (i.e., zoneId, offset, and length). Third, SMRStore orchestrates the lifecycle of zones and data placement strategies in the zones.

Everything is log. SMRStore stores both metadata and data as logs in SMR sequential zones. Specifically, SMRStore uses a basic structural unit, called *record*, to form different types of metadata and data. To avoid wasting space, record is of variable-length and enforces 4KB alignment with disk physical sector.

Guided data placement. Since SMR zones are append-only (except a few czones), and chunks from different Pangu files can be interleaved in SMR zones, deleting Pangu chunks can leave zones with obsolete data. This requires SMRStore to migrate valid data from old zones to new ones, termed as SMR GC in this paper. SMRStore reduces SMR GC by: i) only allowing chunks to be mixed in a zone if they are from the same type of streams (i.e., similar lifespans); ii) trying to allocate an exclusive zone for each large chunk if possible.

Note that SMR GC is different from the OSS GC. In OSS GC, after objects deleted by users, the corresponding Pangu

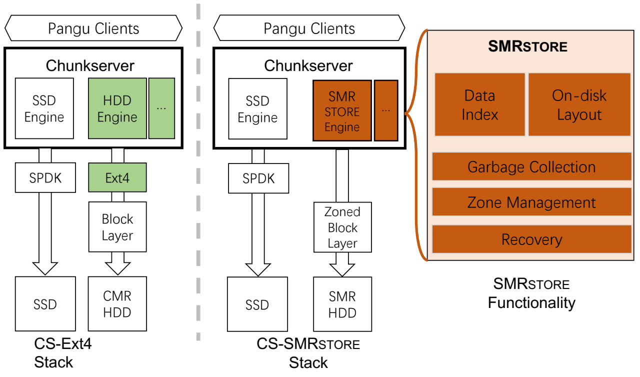


Figure 8: Overview of CS-Ext4 and CS-SMRStore (§5.1). SMRStore *is integrated in chunkserver, runs in the user space and communicates with HM-SMR drives directly by ZBD interface.*

files can be partially filled with obsolete objects. KV servers would create new Pangu files to store the valid objects collected from old Pangu files (i.e., generating OSS GC streams).

5 SMRStore Design & Implementation

5.1 Architecture Overview

Figure 8 shows a side-by-side comparison between running chunkserver with Ext4 on CMR disks (CS-Ext4), and with SMRStore on SMR disks (CS-SMRStore). The main difference is the addition of SMRStore to the chunkserver, sitting in the user space, and communicating with the SMR disks via Zoned Block Device (ZBD) subsystem. Next, we discuss the key functionalities of SMRStore:

- *On-disk data layout.* SMRStore divides an HM-SMR drive into three fixed-size areas, namely the superzone, the metazones, and the datazones. The SMRStore uses "record" as the basic unit for metazone and datazone.
- *Data index.* SMRStore employs three levels of in-memory data structures, including chunk metadata, index group, and record index, to map a chunk to a series of records on the disk.
- *Zone Management.* SMRStore uses a state machine to manage the lifecycle of zones, and keeps metadata (e.g., status) of each zone in the memory. Further, SMRStore adopts three workload-aware zone allocation strategies to achieve low SMR GC overhead.
- *Garbage Collection.* SMRStore periodically performs SMR GC to reclaim area with stale data at the granularity of zones. There are three steps in SMR GC procedure: victim zone selection, data migration, and metadata update.
- *Recovery.* Upon crashes, SMRStore restore through four steps: recovering meta zone table, loading the latest checkpoint, replaying journals, and completing the chunk metadata table by scanning opened data zones.

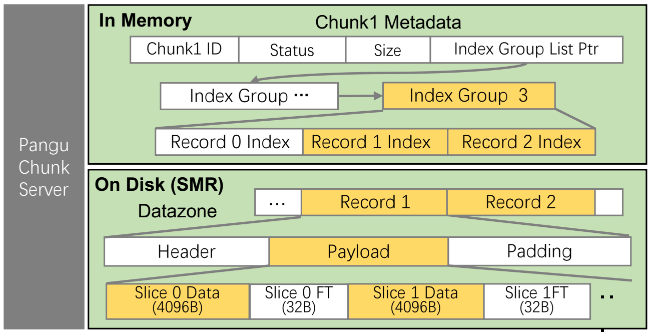


Figure 9: Dataflow in SMRStore engine. (§5.1). *Compared to Figure 3, the storage engine is* SMRStore *(green shaded) and the disk is an HM-SMR drive.* ***FT****: slice footer.*

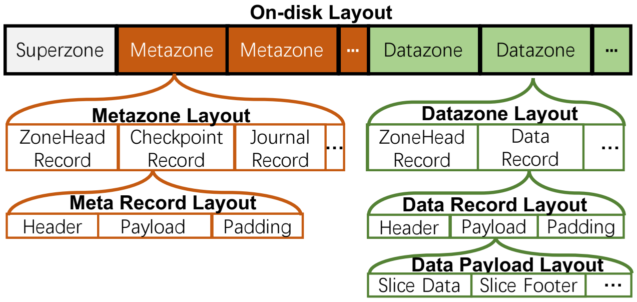


Figure 10: On-disk Data Layout of SMRStore (§5.2). SMRStore *divides a disk into three partitions. Both metadata and data are implemented based on the unified data structure called record. The record can be of variable in length and have different type. The payload of a data record is divided into several slices to support partial read.*

CS-SMRStore I/O Path. When replacing the storage engine with a SMRStore engine, the KV server and Pangu master follow the same procedures shown in Figure 3. As illustrated in Figure 9, SMRStore no longer relies on local system support and uses an in-memory chunk metadata table for mapping. SMRStore first locates the table entry and its index group linked list by using the chunk ID as index. SMRStore further identifies the targeted index group or creates a new one. Then, SMRStore appends data to the datazone (indicated by the targeted index group) as record(s), and updates the record index(es) in the corresponding index group.

5.2 On-Disk Data Layout

Overview. Figure 10 shows the three partitions of an HM-SMR drive under the SMRStore, including one superzone, multiple metazones, and multiple datazones. All partitions are fixed-sized and statically allocated. In other words, we place the superzone on the first SMR zone, the metazones occupy the next 400 SMR zones, and the rest of SMR zones are assigned as datazones. We do not allow metazones and datazones to be interleaved along disk address space to facilitate

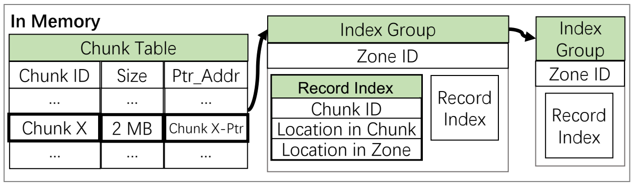


Figure 11: Data Index of SMRStore (§5.3). *In the chunk metadata table, each chunk has a pointer to an index group list. Each index group can have multile record indexes in a zone. The index groups and records are all sorted by the offset to the chunk.*

the metazones scanning during recovery.

Superzone. The superzone stores the information for initialization, including the format version, the format timestamp, and other system configurations.

Metazone. Inside the metazone, there are three types of metadata: the zonehead, checkpoint , and journal. Note that the metazones only store metadata of SMRStore not the metadata of OSS (i.e., data from OSS metadata stream). The metadata are composed by different types of records. The zonehead record stores the zone-related information, such as the zone type and the timestamp of zone allocation (used for recovery). The checkpoint is a full snapshot of in-memory data structures while the journals contain key operations of chunk and zone which we further introduce in §5.6.

Inside each record, there are also three fields: the header, the payload, and the padding. The header specifies the type of records (i.e., zonehead, checkpoint, or journal record), the length of the record and the CRC checksum of the payload. The payload contains the serialized metadata. An optional padding is appended at the end of the record as the SMR drive is 4KB-aligned.

Datazone. The datazones occupy the rest of the disk. In each zone, there are two types of records, the zonehead record and data record. The zonehead record is similar to the metazone zonehead record except the zone type.

Data record & slice. The payload of data record hosts user's data (i.e., a proportion of the chunk). The padding at the tail of a data record is used to bring it a multiple of 4096 bytes (i.e., 4KB-aligned). However, the payload field of data record is different from other types of records (see bottom right of Figure 10). To avoid read amplification, the payload is further divided into 4096-Byte slices, with a 32-Byte slice footer appended to each slice. The slice footer contains the chunk ID (24 bytes), the logical offset to chunk (4 bytes) and the checksum of slice data (4 bytes). Without payload slicing, reading a 4KB from a 512KB record would require SMRStore to fetch the whole record for verifying the payload with the record's checksum. Now, with slices, reading a 4KB only needs to read at most two slices, and SMRStore can use the footer in the slice for checksum verification.

5.3 Data Index

SMRStore uses an in-memory data structure, called record index, to manage the metadata of each record. The record index includes Chunk ID, the logical location of user's data in the chunk (i.e., chunk offset and size of user's data) and record's physical location in the datazone (i.e., offset in the datazone and size of the record).

A chunk usually can have multiple records that are distributed among several datazones. Note that SMRStore appends the data of a chunk to only one datazone at a time until that datazone is full. This guarantees two properties: i) the records in each datazone together must cover a consecutive range of the chunk; ii) the covered chunk ranges in each datazone are not overlapped with each other.

Therefore, we group the record indexes of a chunk in each datazone as an index group. Based on i), inside each index group, we can sort record indexes based on their chunk offsets. The index group also includes the corresponding datazone ID. Moreover, due to property ii), we can further sort the index groups of a chunk, based on the chunk offset of first record index in each group, as a list.

Then, SMRStore organizes the metadata of chunks as a table (see the left of Figure 11). Each entry of the table, indexed by the Chunk ID (a 24 byte UUID), contains the chunk size (the total length of the chunk on this disk), the chunk status (sealed or not, not illustrated in the figure), and the corresponding sorted list of index groups.

When receiving a read request (specified by the chunk ID, the chunk offset, and the data length), SMRStore can locate the chunk metadata with the ChunkID, find the target index group in the sorted list with chunk offset, and locate corresponding record index(es) with the chunk offset and data length.

For a write request, SMRStore always locates the last index group of the target chunk. If there are enough space left on the corresponding datazone, SMRStore appends the data to the datazone as a new record and adds the new record index to the index group. If not, SMRStore allocates a new datazone, appends the data, and adds the record index to the new index group.

5.4 Zone Management

Zone state machine. SMRStore employs a state machine to manage the status of datazones as shown in Figure 12. SMRStore maintains a pool of opened zones (55 zones by default) for fast allocation. SMRStore only resets the GARBAGE zones to FREE zones when the amount of FREE zones are not enough. Metazone follows the similar state machine except there is no pool of opened metazones.

Zone table. SMRStore maintains a zone table in the memory. Each entry of the zone table includes the zone ID, the zone status (OPENED, CLOSED, etc.), a list of live index groups, and a write pointer. We further introduce the usage

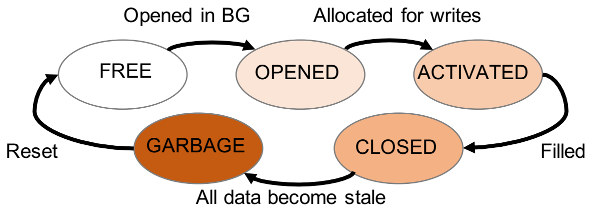


Figure 12: Zone State Transition of SMRStore (§5.4). SMRStore *maintains a pool of opened zones for fast allocation. When a zone is assigned to a new chunk, it transitions to ACTIVATED status. If a zone is closed, it will not be reopened for write before reset.*

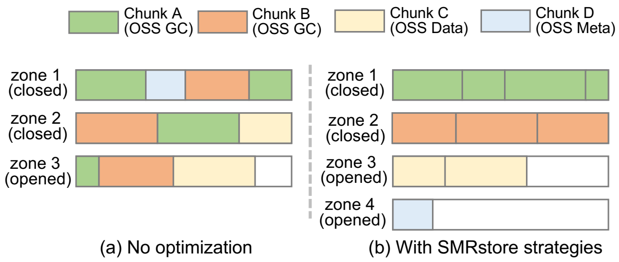


Figure 13: The effectiveness of SMRStore zone allocation strategies. (§5.4). *Chunk A-D come from four different* OSS *streams and shaded with corresponding colors, respectively. Subfigure(a) represents a possible scenario under the random allocation (no optimization) and (b) illustrates a possible layout with* SMRStore *strategies enabled. Each data block may be composed of one or more records.*

of per-zone live index groups list when discussing SMR GC (§5.5) and recovery (§5.6)

Zone allocation. Earlier in F2FS (see §3), we showed that allocating chunks from different types of OSS streams to the same datazone can result in high overhead led by frequent F2FS GC. One can allocate a single datazone for each chunk to reduce such GC. However, this can in return waste considerable space. For example, chunks from OSS metadata stream are usually just several megabytes large, much smaller than the size of a datazone (256MB).

Hence, a more practical solution is to only pursue the "one chunk per zone" for large chunks and let the small chunks with similar lifespans to be mixed together. A challenge here is that the size of a chunk is only determined after it is sealed. In other words, when allocating datazones for incoming OSS streams, SMRStore does not know the sizes of the chunks. Therefore, we design the following zone allocation strategies.

- ① *Separating streams by types.* Note that different OSS types of streams can have disparate characteristics (see Table 1). Therefore, we modify the OSS KV store to embed the types of the OSS streams (i.e., OSS Metadata, OSS Data or OSS GC) along with the data. SMRStore only allows chunks from the same type of streams to share a datazone.
- ② *Adapting chunk size limit for datazone.* Recall that a chunk is sealed when it reaches the size limit, the end of Pangu file or I/O failures. Hence, we configure the size limit of a chunk (including its checksum) to match the size of one datazone (256MB). A chunk may still be sealed well under 256MB (e.g., due to I/O errors). In that case, the left space would be shared with other chunks from the same type of streams if necessary. Note that we still use the default size limit (64MB) for chunks from OSS metadata stream as the corresponding Pangu files are usually small (several to tens of MBs each).
- ③ *Zone pool & round-robin allocation.* SMRStore pre-opens and preserves zones for different types of OSS streams. Specifically, we prepare 40, 10 and 5 opened zones for OSS GC, Data and Metadata stream, respectively. The rationale is that OSS GC stream is the main contributor of the I/O traffic. The OSS Metadata and Data streams have high Pangu file concurrency but can be throttled by the cache SSDs. Moreover, SMRStore allocates zones for new chunks in a round-robin fashion to reduce the chances of chunks to be mixed together.

In Figure 13, we use an example to showcase the effectiveness of our strategies. Consider there are four OSS streams—two OSS GC streams (green and red), one OSS Data stream (yellow) and one OSS Metadata stream (blue). If we do not enable any strategies, SMRStore would allocate datazones one by one for the incoming chunks. As a result, we can expect datazones to be interleaved as shown in Figure 13 (a), similar to the F2FS scenario in §3. In this case, for example if chunk *A* is deleted, all three datazones would have chunk *A*'s stale data and require further SMR GC to reclaim the space.

Now, in Figure 13(b), due to Strategy ①, chunks from different types of streams are no longer mixed together. Moreover, since we reconfigure the size limit of chunks (Strategy ②) and use round-robin allocation (Strategy ③), we can see that chunk *A* and *B* can both own a zone exclusively and fill the entire zone. The three strategies achieve our goal by allocating large-sized chunks with exclusive zones. Now, if chunk *A* is deleted, SMRStore can directly reset zone 1 (i.e., no SMR GC needed).

5.5 Garbage Collection

SMRStore performs garbage collection in three steps:

Victim zone selection. The SMRStore first choose a victim zone among the CLOSED ones to perform SMR GC. We use greedy algorithm to select a zone with most garbages.

Data migration. For the selected victim zone, by scanning live index group list from the zone table, SMRStore can identify valid data in this zone and migrate them to an available zone which is activated only for garbage collection. Moreover, SMRStore enables a throttle module that dynamically limits the throughput of SMR GC to alleviate interference to the foreground I/O.

Metadata update. During migration, SMRSTORE creates index groups with new record indexes for migrated data. After SMR GC finished, SMRSTORE replaces the old index groups in the linked list with the new ones. Finally, SMRSTORE updates the zone table by marking the victim zone as GARBAGE.

5.6 Recovery

SMRSTORE relies on journals and checkpoints to restore the in-memory data structures. In this section, we first introduce the detailed design of journal and checkpoint. Then, we discuss the four steps of recovery.

Checkpoint design. The checkpoint of SMRSTORE is a full snapshot of the in-memory data structures including the chunk metadata table (§5.3) and zone table (§5.4). SMRSTORE periodically creates a checkpoint and persists it into the metazones as a series of records. The zone table is usually small and can be stored in one record. The chunk metadata table is much larger (including all the index groups and record indexes, see Figure 11) and requires multiple records to store. Therefore, we also use two records to mark the start and end of a checkpoint, called checkpoint start/end record.

Journal design. In SMRSTORE, only the create, seal, delete operations of chunk, and the resetting of the zone need to be recorded by journals. Note that SMRSTORE does not journal write operation (i.e., chunk append) as this can severely impact the latency. Instead, we can restore the latest data locations by scanning the previously opened zones. SMRSTORE journals the zone reset operation to handle the case where the same zone may be opened, closed and reused multiple times between two checkpoints. Note that the checkpoint of SMRSTORE is non-blocking, hence the journal records and checkpoint records can be interleaved in the metazones.

Recovery process. The four steps of recovery are as follows:

- *Identifying the latest valid checkpoint.* The first step is to scan zonehead record of each metazone. Recall that, when opened, each metazone is assigned with a timestamp and stored in the zonehead record. Now, by sorting the timestamps, we can scan the metazones from the latest to the earliest to locate the most recent checkpoint end record and further obtain the corresponding checkpoint start record.
- *Loading latest checkpoint.* By scanning records between the checkpoint start and end record, SMRSTORE can recover zone table and chunk metadata table (including index groups and record indexes) from the most recent checkpoint.
- *Replaying journals.* Next, after the checkpoint start record, SMRSTORE replays each journal record till the checkpoint end record to update the zone table, and chunk metadata table.
- *Scanning datazones.* Recall that the journals do not log the write (i.e., `chunk_append()`) operations in order to reduce impacts on the write latency. Therefore, the last step of recovery is to check the datazones that have not been covered by the checkpoint and journals for yet-to-be-recovered writes. SMRSTORE checks the validity (i.e., allocated for writes before crash) of datazones by reading their zonehead records. For each valid datazone, SMRSTORE verifies the data record one by one with the per-record checksums. Finally, SMRSTORE updates the in-memory chunk metadata table (including index groups and record indexes).

6 Evaluation

Software/Hardware setup. We evaluate the end-to-end performance of three types of candidates, including the chunkserver with CMR drives (i.e., CS-Ext4), the chunkserver with F2FS on SMR drives (i.e., CS-F2FS), and SMRSTORE as the storage engine for chunkserver on SMR drives (i.e., CS-SMRSTORE). Additionally, we setup two alternative versions of CS-SMRSTORE. The CS-SMRSTORE-20T shows the performance with full-disk 20TB capacity and the CS-SMRSTORE-OneZone imitates the data placement strategy of F2FS (i.e., mixing data from different streams into one zone). Our node configurations are listed in Table 2.

Workloads setup. We use Fio (modified to use the PANGU SDK) to generate workloads. Our experiments evaluate the following aspects of SMRSTORE.

- *High concurrency micro benchmark.* We extend the microbenchmark in §3 to further evaluate the candidates under highly concurrent random read workloads.
- OSS *simulation macro benchmark.* We also repeat the multi-stream OSS simulation in §3 to evaluate the candidates with multiple write streams, random file deletion and high disk utilization rate.
- *Garbage collection performance.* We evaluate the SMR GC overhead in SMRSTORE and further examine the effectiveness of data placement strategies by comparing corresponding SMR GC overheads under different strategy setups.
- *Recovery.* To evaluate the recovery performance, we restart chunkserver on 20TB SMR drives with 60% capacity utilization, then analyze time consumption in recovery.
- *Resource consumption.* We compare the resources, such as CPU and memory usage, between CS-Ext4 and CS-SMRSTORE (i.e., the two generations of storage stack for standard-class OSS), under a similar setup.
- *Field deployment.* Both CS-Ext4 and CS-SMRSTORE are currently deployed in standard-class OSS. We summarize, demonstrate, and compare key performance statistics of a CS-Ext4 cluster and a CS-SMRSTORE cluster in the field.

6.1 High Concurrency Microbenchmark

In this microbenchmark, we evaluate the candidates on one disk (SMR or CMR) with two types of workloads: High Concurrency Write (HC-W) and High Concurrency Rand

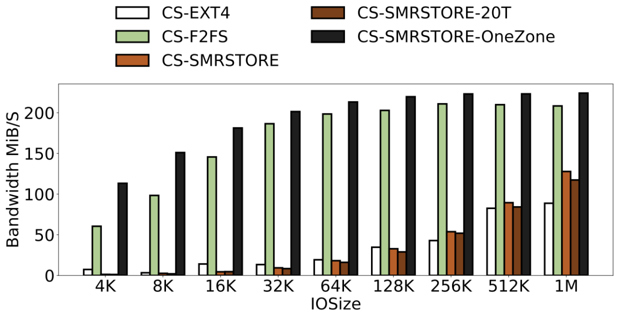


Figure 14: High Concurrency Write Throughput (§6.1). *This figure presents the comparison of write throughput between different storage engines. CS-F2FS (green) and CS-SMRSTORE-OneZone (black) achieve rather high throughputs as they place all incoming chunks onto the same zones, which can incur high F2FS/SMR GC overhead later.*

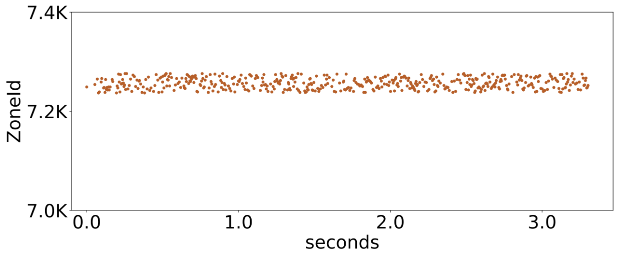


Figure 15: SMRSTORE Access Pattern (§6.1). *The figure presents the distribution of accessed zones under SMRSTORE during a few seconds. Each red dot represents the corresponding zone is accessed (zone ID on the Y axis). This shows the effectiveness of the round-robin allocation, rendering a clear contrast to the zone accessing in CS-F2FS (Figure 5).*

Read (HC-RR). Note that in this experiment, the disk is in the clean state and thus would not trigger F2FS or SMR GC.

Figure 14 shows the HC-W throughput of each candidate under different I/O sizes (from 4KB to 1MB). We can see that CS-SMRSTORE-OneZone and CS-F2FS always have much higher throughput. As discussed in §3.2, flushing data from different streams to enforce the “one zone at a time” policy can significantly benefit the throughput during the clean state (no deletion and F2FS/SMR GC).

For the rest three, their performance gradually increase with I/O size. We notice that, for small I/O size (i.e., <32KB), SMRSTORE shows low throughput. This is caused by the round-robin zone allocation strategy which tends to allocate a new zone for each new chunk to avoid mixed placement, thereby generating random writes for the disk (see Figure 15). As I/O size increases, the throughput of SMRSTORE gradually catches up at 128KB, finally reaches 110MB/s and exceeds CS-Ext4 by 30% at 1MB I/O size. This is acceptable as most writes in standard-class OSS are larger than 128KB (see Table 1).

Figure 16 shows the performance comparison of candidates with HC-RR. We can see CS-SMRSTORE manages to deliver comparable performance to the CS-Ext4. Moreover, in both HC-W and HC-RR experiments, we can observe that the full-disk version (i.e., CS-SMRSTORE-20T) does not suffer severe performance drops.

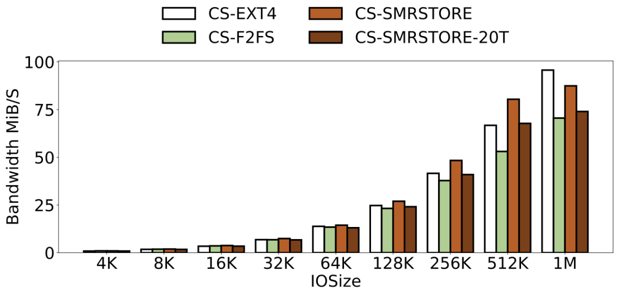


Figure 16: High Concurrency RandRead Throughput (§6.1).

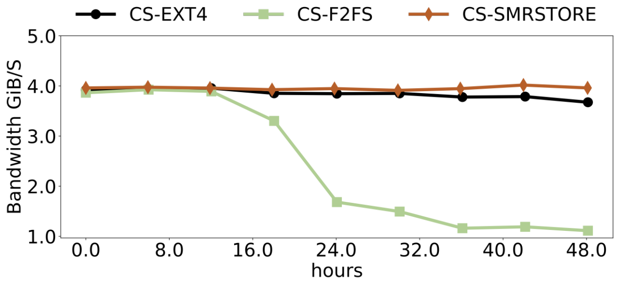


Figure 17: Throughput Comparison of Multi-Stream Benchmark (§6.2).

6.2 Multi-Stream Benchmark

Next, same as the multi-stream experiment in §3.2, we evaluate the candidates under a more realistic setup with multiple data streams, random deletion, and subsequent F2FS/SMR GC. We reuse the set of parameters as Table 3. In Figure 17, all candidates begin with a stable throughput of around 4GB/s. After reaching 80% capacity, random deletion starts, and then the GC kicks in. Recall our discussion in §3.2, CS-F2FS experiences a considerable performance drop due to frequent F2FS GC led by mixed data allocation. CS-Ext4 is hardly affected by the random deletion as Ext4 does not incur GC. Finally, CS-SMRSTORE continues to offer high throughput under random deletion. The main reason is that CS-SMRSTORE adopts several strategies to reduce the frequency and overhead of SMR GC.

Now, we take a closer look to understand the reason behind CS-SMRSTORE performance. In Figure 18, we plot the CDF of zone utilization under SMRSTORE. We can see that most zones are 100% used (i.e., the 100 on the X axis) and only a few zones are occupied with small chunks, thereby indicating less frequent SMR GC.

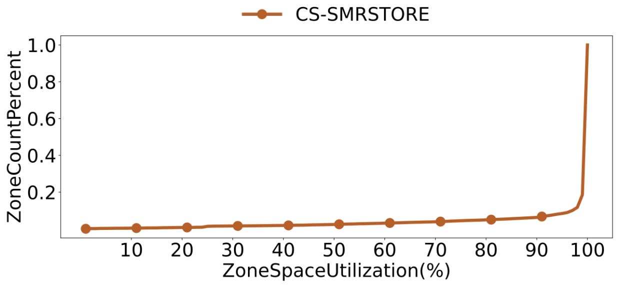


Figure 18: Zone utilizations (CDF) of CS-SMRSTORE (§6.2).

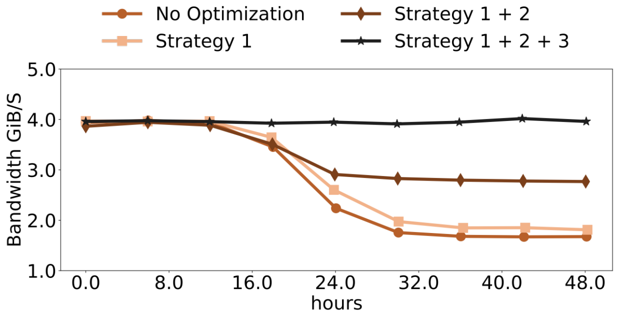


Figure 19: Throughput with different data placement strategies (§6.3). *No optimization: no separating, 64MB chunk size, random allocation on 55 opened zones. Strategy 1: separating streams by types. Strategy 2: adapting chunk size limit for datazone. Strategy 3: zone pool & round-robin allocation.*

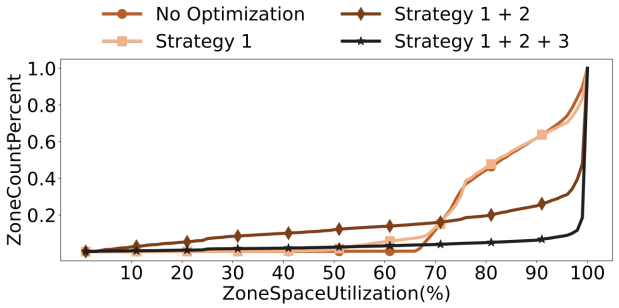


Figure 20: Zone Space Utilizations (CDF) Comparison (§6.3). *The results show that* SMRSTORE *can maintain a high space efficiency by enabling three end-to-end data placement strategies.*

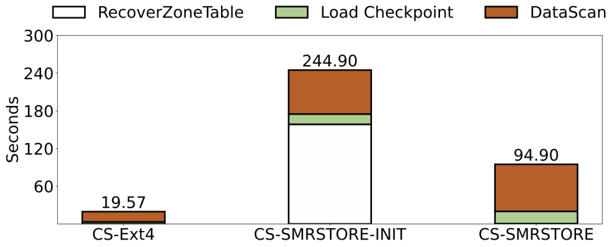


Figure 21: Recovery Performance (§6.4). *The figure shows the breakdown of recovery time. CS-*SMRSTORE*-INIT refers to the initial version of* SMRSTORE *without fixed metazone partition. Recovering zone table refers to the step "Identifying the latest valid checkpoint (§5.6)". Replaying journals is negligible and not shown.*

6.3 Effectiveness of Placement Strategy

The concentrated distribution of high utilization zones is a joint effort of different data placement strategies. Figure 19 shows the various combinations of individual strategies and corresponding effectiveness on write throughput. Here, we run the same multi-stream experiment with four different combinations.

From Figure 19, when only 'separating streams by types' is enabled, the SMR GC overhead is quite obvious and the performance is close to that of no optimization on allocation. Moreover, 'adapting chunk size limit' or 'zone pool & round-robin allocation' each contributes around half of the speedup and such phenomenons are also reflected by the zone utilizations CDFs in Figure 20.

6.4 Recovery Performance

In this experiment, we measure the time consumption in the recovery of a 20TB SMR drive with 60% capacity occupied. Figure 21 shows that CS-Ext4 with a 16TB CMR drive costs less than 20 seconds. CS-Ext4 only has two steps in recovery, loading checkpoint (which takes 3.27 seconds) and data scanning (which takes 16.3 seconds). CS-SMRSTORE completes the recovery with 94.4 seconds which takes around 19 seconds to load the checkpoint, less than 1 second to replay a few journals, and the remaining 75 seconds are for scanning the previously opened zones.

Note we also include a previous implementation, the CS-SMRSTORE-INIT which takes more than 4 minutes to recover. The major reason is that in this version, the on-disk layout is dynamic, meaning the metazones and datazones can be interleaved. As a result, SMRSTORE needs to scan all zone headers (both metazone and datazone) for recovery. Therefore, we switch to static zone allocation.

6.5 Resource Consumption

Memory. In a single server (60 HDDs and 2 SSD caches), the CS-SMRSTORE occupies 49.3GB of memory, around two times more than CS-Ext4. Memory growth is mainly contributed by the in-memory data structures of SMRSTORE. Specifically, the metadata of each chunk occupies around 200 bytes, and each record index in memory needs 8 bytes. The record indexes can be further compressed and we decide not to discuss in this paper due to space limit.

CPU. The CS-SMRSTORE uses around 19 cores which are 26.7% more than CS-EXT4. We use 8 cores for 8 partitions of the two cache SSDs (polling with spdk). We use another 4 cores for user-space network threads. SMRSTORE uses another 7 cores for processing requests, memory copy, checksum calculation, and background GC tasks of 60 SMR drives. With increasing areal density and comparable performance,

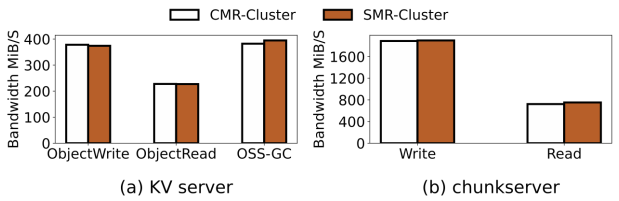


Figure 22: Performance comparison in OSS benchmark (§6.6). *Figure(a) compares key metrics of KV servers, including throughput of object write, object read and* OSS *GC. Figure(b) compares the corresponding read and write throughput of chunkservers.*

the extra overhead on CPU and memory usage is acceptable.

Space efficiency. Apart from persisting data, SMRSTORE further requires extra space for record headers, record paddings, and slice footers. For large IOs (512KB-1MB)—a common scenario in SMRSTORE (i.e., OSS GC/data stream, see Table 1)—SMRSTORE requires another 1-2% space of the IO size. The percentage increases for smaller writes but they are rather uncommon for HDDs due to IO merging in cache SSDs.

6.6 Field Deployment

In the OSS full stack benchmark, all of the key metrics in the SMR cluster are on par with the CMR cluster. The two clusters are both deployed with 13 KV store servers, 13 chunkservers, and 780 HDDs in total. Figure 22 shows that, at OSS service layer, each KV server in the SMR cluster achieves 374.2MB/s object write throughput, 227.7MB/s object read throughput, and 394.8MB/s OSS GC throughput. Each chunkserver in the SMR cluster provides 1898.6MB/s write throughput and 752.8MB/s read throughput. Similarly, in the CMR cluster, each chunkserver provides 1888.3MB/s write throughput and 723MB/s read throughput. This suggests, from an end-to-end perspective, we are able to replace CMR drives in standard-class OSS with SMR drives with no performance penalty thanks to SMRSTORE.

7 Limitation & Future Work

CZone. SMRSTORE follows a strictly log-structured design and thus does not require random writes support from czones. The use of the czones is under discussion. We could use czones as szones by maintaining a writer pointer in memory and a sequence number for each czone. The sequence number is used to identify valid records when the czone is reused.

Ad hoc to Alibaba standard OSS. At the moment, SMRSTORE is dedicated to serve standard-class OSS in Alibaba Cloud. However, SMRSTORE can easily adopt other zoned block devices , such as ZNS SSD. In fact, adapting SMRSTORE to ZNS SSD devices is in progress and will serve other services (e.g., Alibaba EBS).

Garbage collection. The expected on-disk lifespans of OSS data, OSS metadata and OSS GC are different from one OSS cluster to another. Certain clusters can have regular patterns on object creations and deletions while others perform more randomly. Currently, we are exploring more efficient SMR GC algorithms to better serve a variety of OSS workloads based on the accumulated statistics.

8 Related Work

Enabling HM-SMR drives. There are mainly three fashions of solutions in enabling HM-SMR, including adding a shim layer between the host and the ZBD subsystem [20,21], adopting local file systems to provide support [16,17], and modifying applications to efficiently utilize SMR devices [19,26,32]. SMRSTORE differs from above from two aspects. First, SMRSTORE completely discards random write by building everything as logs and hence avoid the potential constraints led by using the limited conventional zones or the tax imposed by random-to-sequential translation. Second, SMRSTORE significantly minimizes SMR GC overhead by end-to-end data placement strategies with the guidance of workloads.

Storage engine designs. To avoid the indirect overheads of general-purpose file systems [17,27], storage engines of cloud storage systems [18] and distributed file systems [31]) tend to evolve towards to user space, special purposed [9], and end-to-end integration [11,32]. SMRSTORE follows and further explores this path by building in the user space and implementing the semantics of PANGU chunks, which is much simpler than general file semantics (e.g., directory operations, file hardlink). Further, the range of the end-to-end integration in SMRSTORE is much wider than host-device, which includes OSS service layer, PANGU distributed file system layer, the storage engine persistence layer, and a novel but backward-incompatible device (i.e., HM-SMR drive). The results of SMRSTORE showcase the benefits can inspire future storage system designs under similar circumstances.

9 Conclusion

This paper describes our efforts in understanding, designing, evaluating, and deploying HM-SMR disks for standard-class OSS in Alibaba. By directly bridging the semantics between PANGU and HM-SMR zoned namespace, enforcing an all-logs layout and adopting guided placement strategies, SMRSTORE achieves our goal by deploying HM-SMR drives in standard-class OSS and providing comparable performance against CMR disks yet with much better cost efficiency.

Acknowledgments

The authors thank our shepherd Prof. Peter Desnoyers and anonymous reviewers for their meticulous reviews and insightful suggestions. We also thank the OSS and the PANGU team for their tremendous support on the SMRSTORE project and this paper. We sincerely thank Yikang Xu who pioneered the SMRSTORE prototype development. This research was partly supported by Alibaba AIR program and NSFC(62102424).

References

[1] Archival-class OSS on Alibaba Cloud. https://www.alibabacloud.com/solutions/backup_archive.

[2] Data Lake on Alibaba Cloud. https://www.alibabacloud.com/solutions/data-lake.

[3] Fio. https://github.com/axboe/fio.

[4] hdparm. https://www.man7.org/linux/man-pages/man8/hdparm.8.html.

[5] Shingled Magnetic Recording. https://zonedstorage.io/docs/introduction/smr.

[6] INCITS T13 Technical Committee. Information technology - Zoned Device ATA Command Set (ZAC). Draft Standard T13/BSR INCITS 537, 2015.

[7] INCITS T10 Technical Committee. Information technology-Zoned Block Commands (ZBC). Draft Standard T10/BSR INCITS 536, 2017.

[8] A. Aghayev and P. Desnoyers. Skylight—A window on shingled disk operation. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[9] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[10] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proceedings of 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[11] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. Moal, G. R. Ganger, and G. Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[12] E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. T'so. Disks for Data Centers. https://research.google/pubs/pub44830.pdf, 2016.

[13] 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 *Proceedings of the 21th ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[14] T. R. Feldman and G. A. Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *Usenix Magazine*, 2013.

[15] G. Gibson and G. Ganger. Principles of operation for shingled disk devices. *Canregie Mellon Parallel Data Laboratory, CMU-PDL-11-107*, 2011.

[16] C. Jin, W.-Y. Xi, Z.-Y. Ching, F. Huo, and C.-T. Lim. HiSMRfs: A high performance file system for shingled storage array. In *Proceedings of 30th Symposium on Mass Storage Systems and Technologies (MSST)*, 2014.

[17] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A new file system for flash storage. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[18] Q. Luo. Implement object storage with smr based key-value store. In *Proceedings of Storage Developer Conference (SDC)*, 2015.

[19] P. Macko, X. Ge, J. Haskins, J. Kelley, D. Slik, K. A. Smith, and M. G. Smith. SMORE: A Cold Data Object Store for SMR Drives (Extended Version). https://arxiv.org/abs/1705.09701, 2017.

[20] A. Manzanares, N. Watkins, C. Guyot, D. LeMoal, C. Maltzahn, and Z. Bandic. ZEA, a data management approach for SMR. In *Proceedings of 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[21] D. L. Moal. dm-zoned: Zoned Block Device device mapper. https://lwn.net/Articles/714387/, 2017.

[22] D. L. Moal. Linux SMR Support Status. https://events.static.linuxfound.org/sites/events/files/slides/lemoal-Linux-SMR-vault-2017.pdf, 2017.

[23] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's warm BLOB storage system. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[24] G. Oh, J. Yang, and S. Ahn. Efficient Key-Value Data Placement for ZNS SSD. *Applied Sciences*, 2021.

[25] Z. Pang, Q. Lu, S. Chen, R. Wang, Y. Xu, and J. Wu. ArkDB: A Key-Value Engine for Scalable Cloud Storage Services. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, 2021.

[26] R. Pitchumani, J. Hughes, and E. L. Miller. SMRDB: Key-Value Data Store for Shingled Magnetic Recording Disks. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR)*, 2015.

[27] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 2013.

[28] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 1992.

[29] A. Suresh, G. A. Gibson, and G. R. Ganger. Shingled Magnetic Recording for Big Data Applications. Technical Report CMU-PDL-11-107, 2012.

[30] 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 *Proceedings of 20th USENIX Conference on File and Storage Technologies (FAST)*, 2022.

[31] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, High-Performance distributed file system. In *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[32] T. Yao, J. Wan, P. Huang, Y. Zhang, Z. Liu, C. Xie, and X. He. GearDB: A GC-free Key-Value Store on HM-SMR Drives with Gear Compaction. In *Proceedings of 17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.

Multi-view Feature-based SSD Failure Prediction: What, When, and Why

Yuqi Zhang[†], *Wenwen Hao*[†], *Ben Niu*[‡], *Kangkang Liu*[‡], *Shuyang Wang*[†], *Na Liu*[†], *Xing He*[†],
Yongwong Gwon[*], *Chankyu Koh*[*]
[†]*Samsung R&D Institute China Xi'an, Samsung Electronics*
[‡]*Tencent* [*]*Samsung Electronics*

Abstract

Solid state drives (SSDs) play an important role in large-scale data centers. SSD failures affect the stability of storage systems and cause additional maintenance overhead. To predict and handle SSD failures in advance, this paper proposes a multi-view and multi-task random forest (MVTRF) scheme. MVTRF predicts SSD failures based on multi-view features extracted from both long-term and short-term monitoring data of SSDs. Particularly, multi-task learning is adopted to simultaneously predict what type of failure it is and when it will occur through the same model. We also extract the key decisions of MVTRF to analyze why the failure will occur. These details of failure would be useful for verifying and handling SSD failures. The proposed MVTRF is evaluated on the large-scale real data from data centers. The experimental results show that MVTRF has higher failure prediction accuracy and improves precision by 46.1% and recall by 57.4% on average compared with the existing schemes. The results also demonstrate the effectiveness of MVTRF on failure type and time prediction and failure cause identification, which helps to improve the efficiency of failure handling.

1 Introduction

Compared with hard disk drives (HDDs), NAND flash-based solid state drives (SSDs) have higher performance and lower power consumption [8, 19] and thus have become popular in enterprise storage systems and large data centers. However, to reduce costs, the storage density of SSDs is increasing, which reduces the endurance and reliability of SSDs [6, 25, 48]. Large-scale data centers usually have hundreds of thousands or even millions of SSDs. Such a large-scale deployment of SSDs poses a challenge to data center reliability. Although redundancy mechanisms (such as replication [38] and RAID [32]) have been used to protect data from loss, SSD failure still causes two major problems. First, even if the data center adopts a redundant protection scheme, SSD failure also affects the performance of the storage system and the stability of online services. Second, SSD failure leads to additional maintenance costs due to failure location, failure recovery, etc. Therefore, SSD failure prediction, as a proactive fault tolerance mechanism, has received increasing attention recently. Compared with the passive redundancy mechanisms, it can identify and proactively handle potential SSD failures in advance, thereby improving the reliability of the storage system and reducing the costs of failure location and recovery. In the large-scale storage system, it is significant to monitor the symptoms of SSD failures and predict failures in advance.

A common monitoring solution for modern storage devices (HDDs and SSDs) is S.M.A.R.T (Self-Monitoring, Analysis, and Reporting Technology), which can monitor and record the internal reliability-related attributes of drives. SMART logs are usually captured regularly, for example, there are one or several SMART logs captured per day for each device (in this paper, a log refers to a snapshot of SSD monitoring attributes). Since SMART logs originate from HDDs, many previous works [4, 9, 14, 21, 23, 24, 29, 42, 46, 49, 51] have studied HDD failure prediction based on SMART logs, and only some works [27, 30, 45, 50] focus on SSDs. In recent years, to better monitor SSDs, some SSD manufacturers have customized more attributes about SSD reliability and failure. Based on these custom attributes, some works [3,7,16] predict SSD failures more effectively.

For SSD failure prediction algorithms, most current schemes are based on supervised learning. They regard failure prediction as a binary classification problem (healthy SSD and failed SSD), and build classification models (such as random forest and neural network) to identify failed SSDs [3, 16, 27, 30, 45]. Some other works [7, 50] adopt anomaly detection approaches (such as isolated forest and autoencoder) to predict SSD failures, based on unsupervised learning. These schemes are primarily designed to learn the pattern of healthy SSDs. When the monitoring log of an SSD is very different from that of most healthy SSDs, it is considered that a failure may occur.

The previous works still face the following challenges. First, most of them [3,7,27,30,45] predict SSD failures based on one or several short-term monitoring logs, and pay less attention to the long-term logs of SSDs. However, through our analysis, some SSD failures may not be reflected in short-term local information, but hidden in long-term information. A few works [16,50] use sequence models such as long short-term memory (LSTM) [17] to directly learn from long-term data, but the sequence lengths of SSD monitoring data are too long and the lengths vary greatly, which affect the performance of sequence models. For long-term data, their trends and distributions are actually important for judging SSD failures (see Section 2.2). Second, although the failure prediction has screened the possible failed SSDs, it lacks instructive suggestions for verifying and handling failures. The operator only knows that a failure may occur, but not know what it is, when and why it will occur. Predicting or analyzing more information such as failure type, lifespan (remaining working time before failure) and failure cause is helpful for operators to verify whether it is an internal SSD failure, and judge what measures to take and whether it is urgent. For example, operators would deal with different types of failures with different urgency and measures (see Section 2.1).

In order to solve these challenges, we propose a multi-view and multi-task random forest (MVTRF) scheme. First, in addition to the short-term raw data, we generate histogram statistics and sequence-related features to reflect the long-term pattern of monitoring data. MVTRF adopts multiple input and groups decision trees to learn these multi-view features in parallel. It can predict SSD failures with both short-term and long-term information. Second, MVTRF employs multi-task learning to jointly learn the failure pattern through the associated failure type classification and remaining lifespan prediction. With these two tasks, the operator knows what type of failure it is and when it will occur, and can take corresponding actions. Finally, according to the decision process, we extract key decisions from MVTRF to reveal why the failure occurs and help operators verify and deal with SSD failures quickly. Experiments on real data from data centers show that MVTRF is effective and outperforms existing schemes. Our contributions are summarized as follows:

- We design histogram features and sequence-related features to characterize the distribution and trend of long-term monitoring data. MVTRF is proposed to jointly learn failure patterns from these features and short-term raw data, thereby improving the prediction accuracy.

- We propose SSD failure type prediction and combine remaining lifespan prediction to suggest proactive measures, in addition to failure prediction. Multi-task learning is adopted for these three tasks, since joint learning of related tasks can improve the model performance for each task.

- We propose a similar decision extraction (SDE) approach to extract the key decisions of MVTRF, so as to identify the symptoms and causes of SSD failures, and provide more information for verifying and handling failures.

2 Data Analysis and Motivation

2.1 Dataset

The large-scale SSD monitoring datasets of Samsung PM1733 and PM9A3 SSDs were collected from the data center of Tencent cloud. The datasets include more than 70 million monitoring logs within nine months from more than 300,000 SSDs with different lifespans in Tencent's data center. The log information consists of SSD serial number, server serial number, timestamp and SSD internal attribute values. Besides SMART attributes, Samsung has customized more internal attributes to enhance SSDs' self-monitoring capability, which makes it possible to predict and analyze more failure information. There are a total of 40 internal attributes for PM1733 and 85 for PM9A3. All these attributes, including standard SMART attributes and custom attributes, are called Telemetry attributes in this paper, and some of them are shown below.

- media_errors: the number of unrecovered data integrity errors detected by the controller
- controller_busy_time: the amount of time the controller spends on I/O commands
- temperature: the current temperature of internal composite
- read_recovery_attempts: total count of uncorrectable NAND reads that require retrying
- wear_leveling_max: maximum erase cycle of internal blocks
- nand_bytes_written: the number of NAND sectors written ($1\ count = 32MB$)

The failure lists of both PM1733 and PM9A3 were also provided by Tencent. The lists contain the information of SSD failures collected by Tencent operators, including the serial number of failed SSDs, failure's report date, failure description, and handling time and measures. There are totally 409 failure records in the lists. After checking by operators, most of them were SSD failures, and a few of them were failures of other devices such as the server backplane. Since manually checking and verifying each failure is a burden, operators need additional failure information (such as failure causes) to verify failures more efficiently.

By analyzing the failure description and handling measures in Tencent's failure lists, we found that failures can be divided into eight types, and different measures were taken at different times to deal with different types of failures. These failure types are called Check Failed, Cancelling I/O, Media Error, SSD Drop, Fail Mode, PLP, Read Only, and Reliability Degradation, and the relevant descriptions are shown in Table 1. Based on the measures and time to handle different failures, we also give a corresponding reference in terms of urgency.

Table 1: Eight SSD failure types.

Failure type	Description	Urgency
Check Failed	Health or performance check failed	High
Cancelling I/O	NVMe cancelling I/O	Medium
PLP	Power loss protection test failed	Medium
SSD Drop	SSD cannot be detected by host	Medium
Fail Mode	Device fail mode	Medium
Media Error	Some data cannot be read correctly	Medium
Read Only	Unable to write data to SSD	Medium
Reliability Degradation	NVMe reliability degradation	Low

For example, the SSDs with Check Failed were processed in an average of four days, and almost all of them were directly replaced, so its urgency is high. In contrast, the SSDs with Reliability Degradation were processed in an average of 19 days, and a small number of them were replaced. Reliability Degradation only means that there may be a problem with the SSD, but no real failure has occurred, while Check Failed generally means that the SSD has an unspecified serious failure with the impact on the performance of storage system. Some definite failure types, such as Media Error and Read Only, have definite effect and may be mitigated by redundancy mechanisms, and the processing urgency is medium.

> **Finding 1:** The failure needs to be checked manually to confirm whether it is an internal SSD failure, and the urgency and measures to deal with the failure may vary depending on the confirmed SSD failure phenomenon and type. Detailed failure information is significant for failure handling.

2.2 Failure Analysis

To gain insight into SSD failures for failure prediction, we analyzed the failed SSDs in Tencent datasets based on Telemetry attributes. First, the distribution of Telemetry attributes of failed SSDs and healthy SSDs were analyzed to mine their differences. We evenly divided the value range of each attribute from minimum to maximum into multiple buckets, and used histograms to compare the data distribution of failed SSDs and healthy SSDs in each bucket.

Figure 1 and Figure 2 compare the data distribution of failed SSDs and healthy SSDs with nand_bytes_written and temperature attributes, respectively. The horizontal coordinate is the bucket index, and the vertical coordinate is the proportion of data that falls in the bucket. Figure 1 shows that most nand_bytes_written values of the failed SSDs and healthy SSDs fall in the buckets 1–7. However, the values of failed SSDs have a larger proportion than healthy SSDs in the later buckets. Figure 2 shows that the data distribution of failed SSDs and healthy SSDs differs greatly in the buckets 20–23 of temperature attribute, but the distribution before bucket 17 is more similar. Overall, the Telemetry values of failed SSDs and healthy SSDs are somewhat different, but the distributions in some ranges are similar.

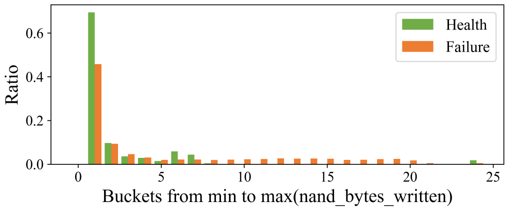


Figure 1: Distribution of nand_bytes_written of failed SSDs and healthy SSDs.

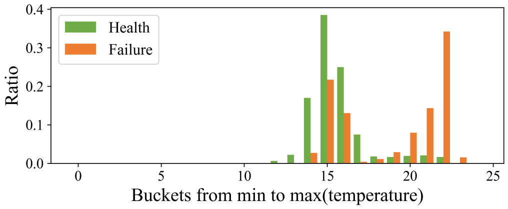


Figure 2: Distribution of temperature of failed SSDs and healthy SSDs.

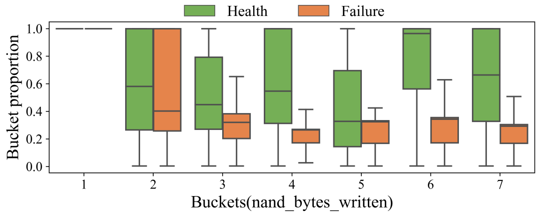


Figure 3: Bucket proportion of long-term data on buckets 1–7 of nand_bytes_written for failed SSDs and healthy SSDs.

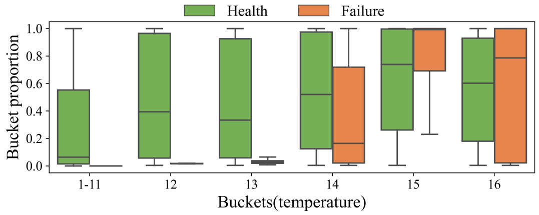


Figure 4: Bucket proportion of long-term data on buckets 1–16 of temperature for failed SSDs and healthy SSDs.

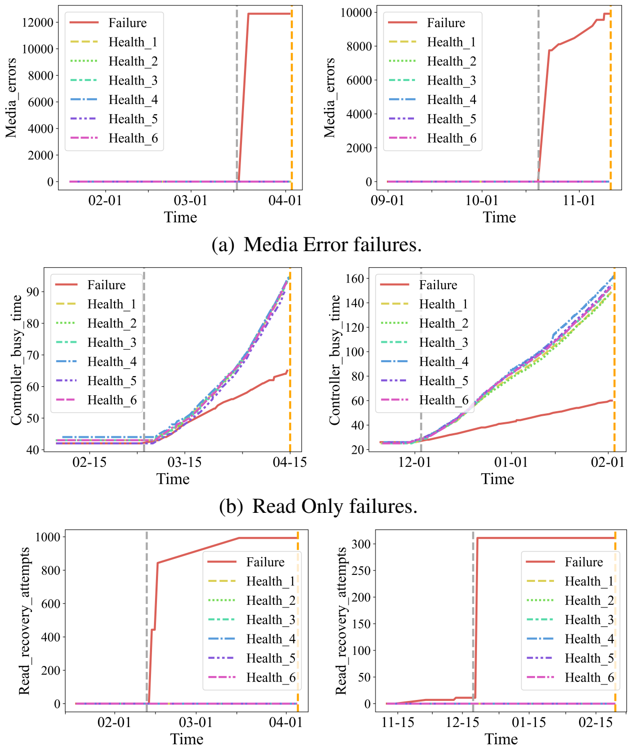
To further distinguish the similar distributions of attributes for failed SSDs and healthy SSDs, we explored the distribution differences of statistics of long-term Telemetry data (each SSD has multiple Telemetry logs over time). Multiple values of each attribute of each SSD over a long time fall into different buckets, and we calculated the proportion of the number of these values in each bucket to the number of values in all buckets, which is called the bucket proportion. Then, we used boxplots (the line in the middle of the box is the median, the lower edge of the box is first quartile, and the upper edge is third quartile) to compare the distribution of bucket proportions for failed SSDs and healthy SSDs.

For the nand_bytes_written attribute, Figure 3 shows the bucket proportion of long-term data for buckets 1–7 whose distributions are similar in Figure 1. The horizontal coordinate is still the bucket index, and the vertical coordinate is the bucket proportion of long-term data. It shows that on these buckets with similar distributions of values, the distribution of bucket proportions for long-term data of failed SSDs and healthy SSDs was different. On buckets 3–7 with small nand_bytes_written, the bucket proportions for long-term data of healthy SSDs were significantly larger than those of failed SSDs. It shows that healthy SSDs suffered from fewer writes over the long term, and thus they were less prone to failure. For the temperature attribute, Figure 4 shows the bucket proportion of long-term data before bucket 17 whose distributions were relatively similar in Figure 2. On buckets 1–13 with low temperature, the bucket proportions for long-term data of healthy SSDs were obviously larger and this indicates that low temperature is good for SSD health. In conclusion, based on the statistics of long-term SSD data, the difference between failed SSDs and healthy SSDs tended to be amplified.

Finding 2: There were some differences in the distribution of Telemetry attributes between failed SSDs and healthy SSDs, and the difference was more significant based on the statistics of long-term Telemetry data of each SSD (i.e., bucket proportion).

The Telemetry attributes of each SSD varied over the long term. Next, we analyzed the long-term changing trends of Telemetry attributes to explore the differences between failed SSDs and healthy SSDs. Since the workload was usually similar for most SSDs on the same server, we compared the changing trends of attributes of the failed SSD with other healthy SSDs on the same server before the failure occurred. Figure 5 shows the changing trend of main abnormal attributes of failed SSDs with different failure types. The horizontal coordinate represents the collection time, and the vertical coordinate represents the attribute value. Figure 5 shows that the attribute trends of healthy SSDs on the same server were similar, while the trend of failed SSD was different over the long term. Moreover, the curve of a failed SSD could involve multiple stages such as slow change, rapid change, and stability.

For the Media Error failure type, Figure 5(a) shows the changing trend of the media_errors attribute of two failed SSDs and the healthy SSDs on the same server. Although there are differences in the value range of two failed SSDs, they both showed a rapid growth trend in about 20 days before the failure occurred. Rapid growth of media_errors usually indicates an unrecoverable component problem inside the SSD and is one of the symptoms of SSD failure. Figure 5(b) shows the changing trend of the controller_busy_time attribute for the Read Only failure type. Compared with the healthy SSDs, both failed SSDs show smaller growth rate of the controller_busy_time attribute, and this trend occurred one to two months before the failure. This trend indicates that the SSD successfully processed fewer I/Os and experienced performance anomalies, and finally the SSD went into read-only mode. Figure 5(c) shows that the SSDs with Check Failed went through a rapid rise for the read_recovery_attempts attribute in about two months before the failure occurred. Too many read retries generally indicate there exists a problem inside the SSD. In general, the same failure type may have similar changing trends of the same Telemetry attribute, but different failure types usually showed different symptoms which may have appeared at different times.



(a) Media Error failures.

(b) Read Only failures.

(c) Check Failed failures.

Figure 5: Attribute trends before the failures. For each failure type, the attribute trends of two failed SSDs and their respective server's healthy SSDs are shown. The gray and orange vertical dashed lines represent the date of symptom onset and the date of failure report respectively.

Finding 3: Due to similar workload and environment, SSDs on the same server usually have similar attribute trends, but failed SSDs may have different trends. The attribute value may change over a long time before SSD fails, and the change may go through multiple stages.

Finding 4: The same types of failures may have similar symptoms in the long-term trends of attributes, and the symptoms may appear at similar times before failures. Different failure types usually exhibit different failure symptoms in attribute trends. This makes it possible to predict the failure type and remaining lifespan of SSDs.

3 Design and Implementation

3.1 Overview

The overall architecture of our multi-view and multi-task random forest (MVTRF) scheme is shown in Figure 6. Based on the analysis in Section 2, our MVTRF design mainly follows three ideas: 1) the distribution and trend related features of long-term data are designed to capture long-term failure patterns; 2) features from different views are combined with group learning and joint decision to predict SSD failures accurately; and 3) detailed failure information is predicted and extracted to improve the efficiency of failure handling.

Specifically, Figure 6 shows that the MVTRF scheme is divided into two parts: offline training and online prediction. Offline training mainly involves two steps. The first is feature extraction. We perform preprocessing and data cleaning on the collected large-scale Telemetry data, and extract raw features, histogram features and sequence-related features. Raw features focus on the values of short-term SSD data, while histogram features and sequence-related features focus on the distribution and trend of long-term SSD data, and they are introduced in detail in Section 3.2. The second step is MVTRF training. The extracted features are trained in groups with MVTRF to obtain information from different views. To predict detailed failure information, we also introduce multi-task learning to simultaneously perform multiple prediction tasks through a single model, including the prediction of health or failure, failure type, and remaining lifespan.

Online prediction involves the following four steps. The first is feature extraction. The three features are extracted from the online data in the same way as offline training. The second is MVTRF prediction. Based on the extracted features, the trained MVTRF model combines decisions from different views to predict whether the SSD will fail, as well as the specific failure type and remaining lifespan. The third is failure cause identification. When an SSD failure is predicted, the key decisions in the judgment process of MVTRF model are extracted to analyze the possible causes of the failure. Through multi-task prediction and failure-cause identification, MVTRF not only identifies the failed SSD, but also answers what the failure is, when and why it will occur. Based on the information above, the fourth step is to verify the failure and take corresponding measures. Furthermore, we regularly train the model offline (e.g., training a new model monthly or quarterly) and update it online to ensure that the model can adapt to data changes. Next, we will introduce multi-view feature extraction, MVTRF, failure-cause identification and failure handling in detail.

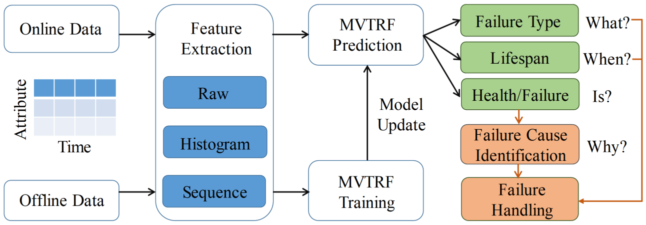


Figure 6: Overall architecture.

3.2 Multi-view Feature Extraction

According to our observations of the symptoms of SSD failures in Section 2.2, we found that SSD failures were not only reflected in the abnormal value of short-term data, but also hidden in the distribution and trend of long-term data. It is an option to directly feed long-term data into sequence models such as LSTM. However, due to different usage periods and irregular collection, the number of Telemetry logs of different SSDs varies greatly (from a few to several thousands in our datasets). It is difficult for sequence models to process sequence data with such different lengths [20]. Moreover, overly lengthy sequences also affect the performance of sequence models (for example, LSTM has the vanishing gradient problem in the case of long sequences [47]), and lead to excessive computational complexity and overhead.

To avoid using long-term data directly, we extract features from long-term data to represent its distribution and trend. The analysis in Section 2.2 shows that the bucket statistics of long-term data help to distinguish between failed SSDs and healthy SSDs, therefore we first introduce histogram features based on bucket statistics. Then, from Section 2.2, we observed that the fluctuation and trend of long-term data also implies the failure symptoms, and thus we introduce sequence-related features that can characterize the degree of sequence fluctuation and change. Histogram features and sequence-related features extract key information from long-term data and discard redundant information. These features and short-term raw data constitute multi-view information for SSD failure prediction. Specifically, when the T-th Telemetry data of an SSD is collected, we extract raw features, histogram features and sequence-related features as follows.

3.2.1 Raw Features

After preprocessing and data cleaning, the data of a Telemetry log are the raw features. We discard attributes with exactly the same value in offline training, and do the same in online prediction. Assuming that there are N attributes remaining after data cleaning, the raw features of the T-th Telemetry data of SSD are defined as $D_T = \{a_{1T}, a_{2T}, ..., a_{nT}, ..., a_{NT}\}$, where $a_{1T}, a_{2T}, ..., a_{nT}, ..., a_{NT}$ are the values of N attributes. We mainly use raw features to capture short-term abnormal value of attributes, so they come from a single Telemetry log by default.

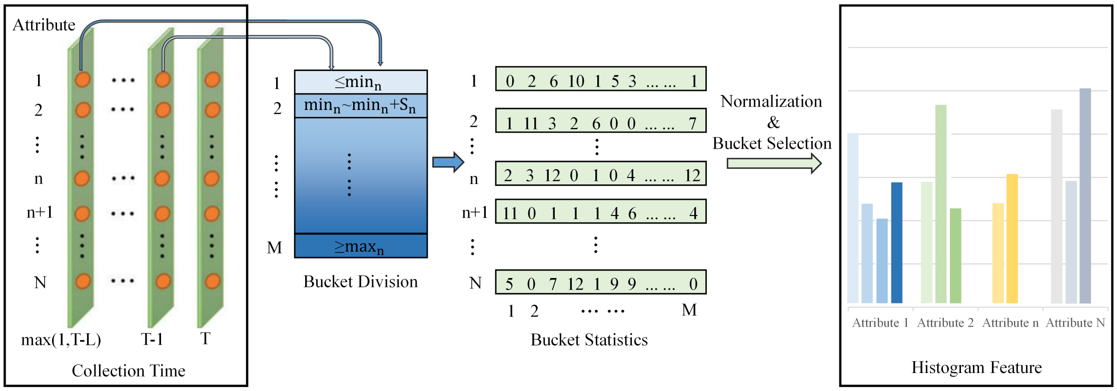


Figure 7: Overall process of generating histogram features.

3.2.2 Histogram Features

Histogram features are proposed to represent the distribution of Telemetry attributes over the long term. They are obtained by bucket statistics on the long-term raw features D_{T-L}–D_T of the SSD. L defaults to 256 and the time span of 256 logs is generally more than three months, which can cover the time span of failure symptoms analyzed in Section 2.2. The overall process of generating histogram features is shown in Figure 7. First, the minimum and maximum values of each attribute of all data are calculated during offline training, and the min and max of the *n-th* attribute are defined as min_n, max_n. Then, the min to max range of each attribute is divided into M buckets (100 by default), and the M ranges of the *n-th* attribute are defined as $\{(-\infty, min_n]$, $(min_n, min_n+S_n]$,...,$(min_n+(M-3)\times S_n$, $min_n+(M-2)\times S_n)$, $[max_n, +\infty)\}$, where $S_n = (max_n - min_n)/(M-2)$. Since the min and max of many attributes have special meaning, the min and max buckets are independent. Afterwards, we divide the features of each attribute of D_{T-L}–D_T into each bucket and count them. The statistics of the *n-th* attribute on M buckets is defined as $\{C_{n1}, C_{n2}, ..., C_{nm}, ..., C_{nM}\}$, where C_{nm} is the count of this attribute falling in the range of bucket m and $\sum_{m=1}^{M} C_{nm} = L$. In particular, when the number of SSD logs is less than L, all raw features (D_1–D_T) of the SSD are counted in buckets. In order to avoid the influence of this special case, we divide the bucket counts by the number of logs to get the proportions, and the formula for normalizing C_{nm} to the proportion P_{nm} is as follows.

$$P_{nm} = \begin{cases} \frac{C_{nm}}{T}, & T < L \\ \frac{C_{nm}}{L}, & T \geq L \end{cases} \tag{1}$$

Then the normalized *M-dimensional* feature $\{P_{n1}, P_{n2}, ..., P_{nm}, ..., P_{nM}\}$ of the *n-th* attribute is obtained, and $\sum_{m=1}^{M} P_{nm} = 1$. Through normalization, we solve the problem of large differences in the number of SSD logs. Next, we concatenate the *M-dimensional* features of all N attributes to get the histogram features whose dimension is $N \times M$. Since some buckets are less meaningful for failure prediction (e.g., the bucket for $wear\_leveling\_max = 0$), we adopt recursive feature elimination with cross-validation (RFECV) [28] to remove some buckets. During offline training, the RFECV algorithm forms multiple bucket subsets by recursively eliminating the least important buckets, and then selects the bucket subset with highest discrimination between failed SSDs and healthy SSDs through cross-validation. During online prediction, we only need to calculate the values of these selected buckets as the final histogram features. This not only reduces the noise from buckets with low discrimination, but also decreases the feature dimension and computational complexity.

3.2.3 Sequence-related Features

Sequence-related features are proposed to represent the fluctuation and trend of long-term raw features D_{T-L}–D_T of SSD. As stated in Finding 3 (see Section 2.2), the attribute trends of failed SSDs may change over a long time, and there may be multiple change stages. We introduce the coefficient of variation [2] to characterize the fluctuation of the attribute, and introduce kurtosis [10] and slope to characterize the trend of the attribute. To capture the multiple changing stages that may exist in long-term data, we also divide D_{T-L}–D_T into G segments equally in the time dimension (G is 4 by default), and calculate the coefficient of variation, kurtosis and slope separately for each segment. Assuming that the *g-th* segment starts at t_s and ends at t_e ($T-L \leq t_s < t_e \leq T$) and the raw features are D_{t_s}–D_{t_e}, the sequence-related features are calculated as follows.

Coefficient of variation. The coefficient of variation can measure the dispersion degree of the attribute over a long period of time. Relative to variance or standard deviation, the coefficient of variation can eliminate the effect of different scales for different attributes and different SSDs. We calculate the coefficient of variation for each segment window of each

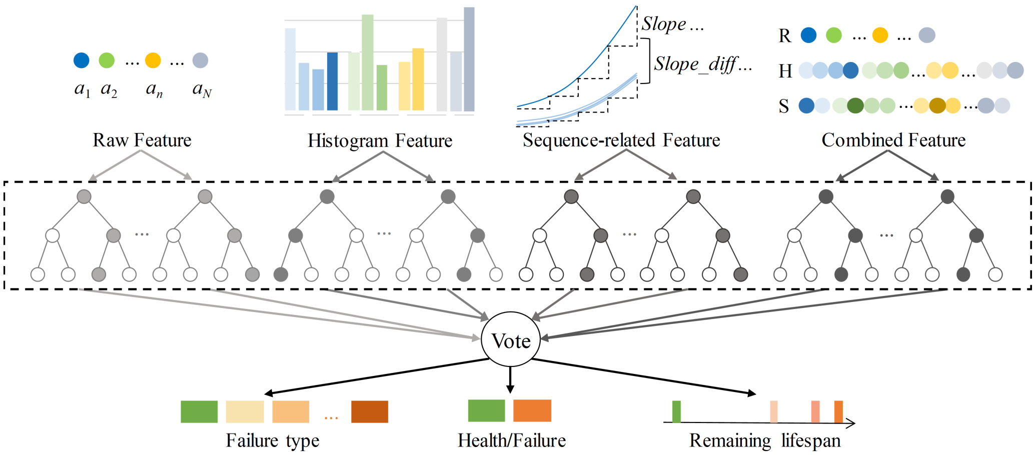


Figure 8: The structure of MVTRF.

attribute of the long-term raw features D_{T-L}–D_T respectively, and the calculation formula of the coefficient of variation $CVAR_{ng}$ for the g-th segment of the n-th attribute is as follows:

$$CVAR_{ng} = \frac{\sqrt{\frac{G}{L}\sum_{t=t_s}^{t_e}(a_{nt}-\mu_{ng})^2}}{\mu_{ng}} \tag{2}$$

where $\mu_{ng} = \frac{G}{L}\sum_{t=t_s}^{t_e} a_{nt}$.

Kurtosis. Kurtosis reflects the steepness of an attribute's distribution over the long term. The calculation formula of the kurtosis $KURT_{ng}$ for the g-th segment of the n-th attribute is shown below:

$$KURT_{ng} = \frac{\frac{G}{L}\sum_{t=t_s}^{t_e}(a_{nt}-\mu_{ng})^4}{(\frac{G}{L}\sum_{t=t_s}^{t_e}(a_{nt}-\mu_{ng})^2)^2} - 3 \tag{3}$$

Slope. Slope can reflect the changing trend of an attribute over time. The slope $SLOPE_{ng}$ for the g-th segment of the n-th attribute is calculated as follows:

$$SLOPE_{ng} = \frac{a_{nt_e} - a_{nt_s}}{t_e - t_s} \tag{4}$$

In addition, when the number of raw features of an SSD is less than L, the above features are calculated based on all the raw features of the SSD (i.e., D_1–D_T, and $L = T$ in the above formula), thereby avoiding the impact of various sequence lengths.

As stated in Finding 3, the trends of some attributes of failed SSDs may be quite different from those of other healthy SSDs on the same server. Therefore, for the above $CVAR$, $KURT$, and $SLOPE$, we calculate the difference between their values of an SSD and the average values of the same feature of other SSDs on the same server, defined as $CVAR\_diff$, $KURT\_diff$, $SLOPE\_diff$. SSDs in the same server usually have similar workloads, so differences of attribute fluctuations and trends between these SSDs can provide more information for failure prediction. Next, we concatenate the $CVAR$, $KURT$, $SLOPE$ and $CVAR\_diff$, $KURT\_diff$, $SLOPE\_diff$ of G windows of all N attributes to obtain sequence-related features of the SSD, with a dimension of $N \times G \times 6$. Finally, RFECV is also used to select more effective features from these features, similar to the approach in Section 3.2.2.

3.3 MVTRF

To learn the pattern of the extracted features, we chose random forests [5] as our base model for three reasons. First, existing studies have demonstrated good performance of random forests on SSD failure prediction [3, 27, 30, 45]. Second, a random forest is composed of multiple decision trees, and each decision tree divides the samples into different classes through a series of judgments on features. Its interpretability is good, which is helpful in further identifying failure causes through the judgment process (see Section 3.4). Third, the computational complexity of random forests is lower compared with neural network-related models, which is beneficial in reducing overhead during offline training and online prediction.

Section 3.2 introduced raw features, histogram features and sequence-related features. Each of them actually characterizes the state of SSDs from a different view, and we concatenate these three features together to form combined features with a global view. It is an option to adopt combined features as the input of all decision trees of random forest. However, it would be more reliable to predict SSD failures from these different views independently and then make decisions together. Therefore, we designed MVTRF with different sets of decision trees to learn different types of features in parallel. As shown in Figure 8, all decision trees of a random forest are equally divided into four sets, which learn raw features, histogram features, sequence-related features and combined features respectively. Then, all decision trees of the four sets vote to get the final prediction result. The class with the most votes is the predicted class, and the vote share is the confidence probability. In this way, we combine features from different views to obtain the final judgment.

As stated in Finding 1 (see Section 2.1), more failure information can help operators take actions, and thus we recommend predicting the failure type and remaining lifespan when predicting the SSD failure. We adopt multi-task learning [1] to allow the single model to learn these three prediction tasks simultaneously. Multi-task learning and prediction with a single model has the following two advantages over using three independent models to learn and predict three tasks. First, our three tasks are related to each other. For example, Finding 4 in Section 2.2 shows the correlation between failure type and the time window of failure symptom. Joint learning of related tasks tends to improve the prediction accuracy of the model for each task. Second, learning and predicting three tasks simultaneously via a single model can reduce the time and overhead of training and prediction.

The specific definitions of the three tasks are as follows. 1) Failure prediction. We define it as a binary classification task. The data of healthy SSDs and failed SSDs are labeled 0 and 1 respectively. 2) Failure type prediction. We define it as a multi-classification task. The data of healthy SSDs and failed SSDs are labeled 0 and 1–O respectively. Our datasets have eight failure types, so $O = 8$. 3) Remaining lifespan prediction. Regression is more suitable for this task, but in order to maintain unity with the above two tasks, we also define it as a multi-classification task. The data more than one week from the failure are labeled 0, the data from one day to one week from the failure are labeled 1, the data within one day from the failure are labeled 2, and the data around the time of failure are labeled 3. Through multi-task learning, the prediction accuracy of each task is improved and more information is available for recommending proactive measures.

3.4 Cause Identification and Failure Handling

In a production environment, some SSD anomalies may actually be caused by failures of other devices, such as the server backplane. When a failure is predicted, operators need to understand the symptoms and causes of the failure to confirm exactly what device is failing. In fact, one of the reasons to use the random forest algorithm lies in its interpretability. Random forests are based on decision trees which are essentially a series of threshold decisions. It is in line with human thinking, that is, the final result is obtained through the combination of multiple judgments. By analyzing the decision process, we can reveal why there is a failure, thereby identifying the symptoms and causes of failure. However, a random forest is an ensemble of multiple decision trees, and it is difficult to analyze so many decision processes. Therefore, we propose similar decision extraction (SDE) to obtain key decisions from multiple decision trees in MVTRF to reflect the overall decision process and find the failure causes.

Figure 9 shows an example of how SDE works and there are three steps involved. First, each decision is chosen by the decision tree due to its distinguishing ability, and we extract similar decisions that appear more frequently in multiple decision trees as key decisions. Two decisions are considered to be similar when they meet the following conditions: 1) the features and decision logic (i.e., $\leq$ or $>$) for the two decisions are the same; and 2) the decision thresholds of both decisions are similar, and the difference between the two thresholds is within $\propto$ (10% by default). We look for similar decisions in other decision trees for each decision, and the number of similar decisions is used as the weight of this decision.

After calculating the weights of all decisions, the second step is to remove redundant similar decisions. Drawing on the idea of Non-Maximum Suppression [31], SDE retains decisions with higher weights as key decisions and discards similar decisions with lower weights. The main process is as follows. 1) Sort the weights of all decisions; 2) Select the decision with the highest weight from the unprocessed decisions; 3) Remove other decisions similar to this decision; and 4) Repeat operations 2 and 3 above until the weight of the selected decision is less than half of the global highest weight. In this way, redundant similar decisions are represented by the key decisions with higher weights. Finally, the weights of key decisions with the same features and decision logic can be integrated, and the most strict threshold (i.e., the maximum value for $>$ and the minimum value for $\leq$) is retained to show the outlier.

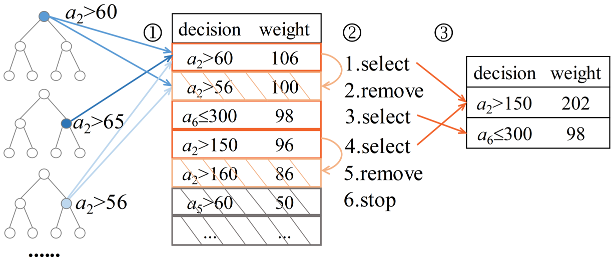


Figure 9: SDE. ①: Statistics of similar decisions within 10% threshold difference; ②: Non-maximum suppression on similar decisions; ③: Integrating key decisions with the same features and decision logic.

The key decisions extracted by SDE can reveal the failure cause and thus help to confirm whether it is an internal failure of the SSD. The key decisions of many failures involve SSD internal errors (e.g., excessive media errors, bad blocks or program failures), indicating that SSDs are failing. When key decisions involve the communication or environment, such as PCI errors or temperature, operators also need to check external devices (such as backplane) or the environment in addition to the SSD. The failure causes revealed by key decisions can significantly improve the efficiency of operators in verifying failures.

When an SSD failure is confirmed, the measures taken are based on the predicted failure type and remaining lifespan. As described in Section 2.1, different failure types may have different processing urgency. For SSDs with a high-urgency failure type, operators can replace them directly. The failures with low or medium urgency and long remaining lifespan can be further analyzed by operators, for example, by regular full-disk scans using scrub technology [27]. Depending on the urgency and remaining lifespan, the scan interval can also be adjusted accordingly. In this way, the impact on healthy SSDs can be significantly reduced while real failures are dealt with in time.

4 Evaluation

We evaluated our MVTRF scheme on real datasets from data centers. The following gives dataset setup and the evaluation metrics.

Dataset setup: For failure prediction, MVTRF was compared with existing schemes on three datasets. Besides the PM1733 and PM9A3 Tencent Telemetry datasets introduced in Section 2.1, the Alibaba public SMART dataset [45] was also used to evaluate the generalizability of MVTRF. This public dataset has multiple SSD models, but the number of failed SSDs for some models is inconsistent with the description of their paper, such as the MA1 and MC1 models. Except these models, we selected the MB1 model with the most failed SSDs for the experiment, as more samples can reduce the test error. There were 42,594 healthy SSDs and 1,807 failed SSDs with two-year SMART data of 16 standard attributes for the MB1 model.

We evaluated the performance of schemes in real scenarios, i.e., the history data were used to train models and new data online were used to predict SSD failures. Similar to the previous work [45], each dataset was divided into a training set, a validation set and a test set in chronological order. The training set was used to train the model, the validation set was used to tune model's hyper-parameters by evaluating the model during training, and the test set was used for the final evaluation of the model. For each dataset, we conducted two or three independent experiments on different data partitions, as detailed in Table 2. The average results of the independent experiments were deemed as the final results.

To further evaluate the generalizability of MVTRF for failure prediction on a new batch of SSDs, we performed a five-fold cross-validation on SSDs of PM1733 Tencent dataset. The further discussion and analysis in Section 4.2 to Section 4.4 were also performed on the PM1733 dataset.

Metrics: We used precision, recall, F0.5-Score and ROC_AUC to evaluate the prediction accuracy.

Precision: The proportion of correctly predicted failed SSDs (true alarms) to all predicted failed SSDs (both true alarms and false alarms).

Table 2: Data partitions for three datasets.

Dataset	Experiment round	Train set (month)	Val set (month)	Test set (month)
Samsung PM1733 (Tencent)	1	1–7th	8th	9th
	2	1–6th	7th	8th
Samsung PM9A3 (Tencent)	1	1–7th	8th	9th
	2	1–6th	7th	8th
MB1 (Alibaba) [45] (detailed model unkown)	1	1–22th	23th	24th
	2	1–21th	22th	23th
	3	1–20th	21th	22th

Recall: The proportion of correctly predicted failed SSDs to all actual failed SSDs, also called the true positive rate (TPR).

*F*0.5-*Score*: $\frac{(1+0.5^2)\times Precision\times Recall}{0.5^2\times Precision+Recall}$. It is the harmonic average of precision and recall, where precision is weighted higher. To avoid more false alarms for SSD failure prediction in practice, operators pay more attention to precision [45], and thus we use F0.5-Score to more comprehensively evaluate the effectiveness of schemes in a production environment.

ROC_AUC: For the above three indicators, the discrimination threshold of binary classification was fixed. In practice, different discrimination thresholds may be used. For example, to predict more failed SSDs, the discrimination threshold can be set lower, although there may be more false alarms at the same time. Therefore, we introduce the area under the curve of receiver operating characteristic (ROC) [12] to reflect the diagnostic ability of the binary classification model at different discrimination thresholds. The ROC curve is created by plotting the TPR versus the false positive rate (FPR, the proportion of false alarms to all healthy SSDs) at various thresholds. The area under the ROC curve (ROC_AUC) is a single score that can reflect the ability of the model to distinguish between failed SSDs and healthy SSDs across discrimination thresholds [7].

4.1 Comparison with Existing Schemes

In this section, we compare the proposed MVTRF with Random Forest, Neural Network, Autoencoder, and Ensemble LSTM on failure prediction. The descriptions of these existing methods are as follows. 1) Random Forest (RF): The raw features of a single monitoring log are used as the input of the random forest to predict SSD failures [3], which is the same as the single-task RF with raw features discussed in Section 4.3. 2) Neural Network (NN): SSD failures are predicted based on raw features using a neural network [3]. 3) Autoencoder (AE): The raw features of healthy SSDs are used as the input and they are reconstructed through an encoder and decoder. The reconstruction loss (i.e., the Euclidean distance between the input and reconstructed output) is used to predict SSD failures [7]. 4) Ensemble LSTM (LSTM): LSTM is used to capture failure symptoms from sequence data (the

Table 3: Comparison of MVTRF with existing methods for failure prediction on three datasets.

Methods	PM1733 Tencent				PM9A3 Tencent				MB1 Alibaba				Average			
	P	R	F	AUC	P	R	F	AUC	P	R	F	AUC	P	R	F	AUC
RF [3]	0.58	0.31	0.48	0.69	0.75	0.33	0.60	0.75	0.56	0.37	0.51	0.87	0.63	0.34	0.53	0.77
NN [3]	0.63	0.14	0.36	0.58	**0.85**	0.31	0.58	0.61	0.72	0.46	0.64	**0.89**	0.73	0.30	0.53	0.69
AE [7]	0.54	0.14	0.34	0.77	0.54	0.33	0.40	0.78	0.36	0.46	0.31	0.88	0.48	0.31	0.35	0.81
LSTM [16]	0.36	**0.40**	0.36	0.69	0.52	0.25	0.28	0.62	0.63	0.61	0.62	0.87	0.50	0.42	0.42	0.73
MVTRF(Ours)	**0.90**	**0.40**	**0.72**	**0.81**	0.70	**0.42**	**0.61**	**0.83**	**0.89**	**0.76**	**0.86**	0.86	**0.83**	**0.53**	**0.73**	**0.83**

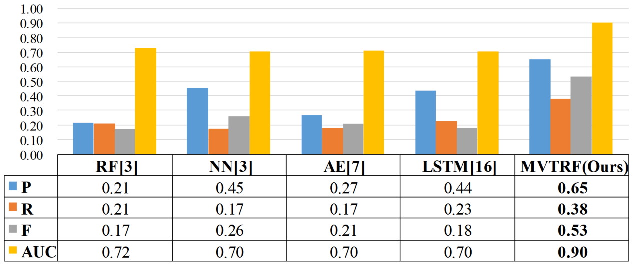
(P: precision; R: recall; F: F0.5-Score; AUC: ROC_AUC)

sequence length is also set to 256 for comparison), and multiple LSTMs are integrated to jointly predict SSD failures [16]. We re-implemented these algorithms, since the source code was not available.

Table 3 shows the results of these methods on the three datasets and the average results. RF, NN and AE are based on the raw features of a single monitoring log and cannot find failure patterns in long-term data, so they produce lower recall. AE predicts SSD failures only by learning the pattern of healthy SSDs and its average precision (0.48) is the lowest. However, its average ROC_AUC reaches 0.81, indicating that AE can better distinguish between failed SSDs and healthy SSDs at lower discrimination thresholds. LSTM achieves the average recall of 0.42 and outperforms the previous methods. This is because LSTM directly takes long-term sequence data as input and can capture more long-term failure symptoms. However, its precision and ROC_AUC is low, since the excessively long sequence length and the difference in lengths bring noise to the LSTM model.

For the average results of three datasets, our MVTRF improves precision by 46.1%, recall by 57.4%, F0.5-Score by 64.5%, and ROC_AUC by 11.1% on average compared with the four existing methods. We extract histogram features and sequence-related features from long-term sequence data to reflect the distribution and trend, thereby reducing noise and redundant information. MVTRF learns these features and raw features separately and predicts SSD failures by combining different views, which is more accurate and comprehensive. In addition, MVTRF performs better on the MB1 Alibaba dataset with a longer time span and more failed SSDs, which is conducive to the learning of long-term failure patterns. Although the three datasets have different SSD models (PM1733, PM9A3, and MB1), monitoring attributes (40 Telemetry attributes, 85 Telemetry attributes, and 16 SMART attributes), and time spans (9 months or two years), our MVTRF shows better performance on all three datasets, which demonstrates its robustness and generalizability.

Furthermore, a five-fold cross-validation on the PM1733 Tencent dataset was performed to further evaluate the effectiveness and generalizability of MVTRF in terms of failure prediction on a new batch of SSDs. Similar to previous work [3], the dataset was divided into five parts according to the serial numbers of the SSDs, and there were five independent experiments accordingly. In each experiment, four parts were selected for training and validation and one for testing. Therefore, SSDs in the test set do not appear in the training set for each experiment, and the test sets of the five experiments contain all SSDs. Fig. 10 shows that the cross-validation results were roughly consistent with the results in Table 3, with some reduction in prediction accuracy. The data patterns of unseen SSDs may be slightly different, which has some impact on the prediction. Compared with the existing methods, our MVTRF showed great improvements in four metrics. It implies that MVTRF is also more effective in failure prediction of unseen SSDs.



	RF[3]	NN[3]	AE[7]	LSTM[16]	MVTRF(Ours)
■ P	0.21	0.45	0.27	0.44	**0.65**
■ R	0.21	0.17	0.17	0.23	**0.38**
■ F	0.17	0.26	0.21	0.18	**0.53**
■ AUC	0.72	0.70	0.70	0.70	**0.90**

Figure 10: Cross-validation on PM1733 dataset. (P: precision; R: recall; F: F0.5-Score; AUC: ROC_AUC)

4.2 Discussion on Multi-view Features

Besides the raw features, this paper proposes histogram features and sequence-related features based on long-term data. These features reflect the state of SSDs from different views. By concatenating these three features, the combined features have a more comprehensive view. We first trained RF with each feature separately and compared their prediction accuracy to analyze the impact of different features on SSD failure prediction. Then, RF, NN, and AE with combined features and our MVTRF were also compared together to evaluate the effectiveness of MVTRF.

Table 4 shows the results on the PM1733 Tencent dataset. Raw features focus on abnormal attribute values, which are easy to judge, and thus their recall is relatively high. However, the short-term raw features cannot capture some failure symptoms in the long-term information, therefore the ROC_AUC was the lowest (0.69), implying that it is difficult to find more failed SSDs at lower discrimination thresholds. The histogram features and sequence-related features reflect the distribution and trend of long-term data, and more failure symptoms can be found, so their ROC_AUC is higher. The combined feature contains the above three features. Since it contains multi-view information, RF with combined features performed well in each indicator. However, NN and AE with the same features did not perform so well. The combined features are comprehensive but also contain too much information, and this leads to the overfitting problem of these two models in training, while RF reduces overfitting through the joint decision of various decision trees [41]. Finally, MVTRF reached 0.90, 0.40 and 0.72 in precision, recall and F0.5-Score respectively. It enables different sets of decision trees to capture failure symptoms from different views, thereby further reducing overfitting caused by mixed excess information during training.

Table 4: Comparison of MVTRF and existing methods with different features.

Method	Precision	Recall	F0.5-Score	ROC_AUC
RF + Raw	0.61	0.34	0.52	0.69
RF + Histogram	**1.00**	0.17	0.48	0.72
RF + Sequence	0.50	0.25	0.38	**0.81**
RF + Combined	0.83	0.37	0.66	0.78
NN + Combined	0.79	0.17	0.39	0.74
AE + Combined	0.88	0.14	0.40	0.77
MVTRF	0.90	**0.40**	**0.72**	**0.81**

4.3 Multi-task Learning and Prediction

In addition to failure prediction, this paper introduces the tasks of failure type prediction and remaining lifespan prediction (see Section 3.3). Since joint learning of related tasks is often beneficial for each task, we perform multi-task learning and prediction through a single model. On the baseline RF with raw features and our MVTRF, the impact of multi-task learning on each task was evaluated. Table 5 compares the performance of two models under single-task learning and multi-task learning for three tasks. For failure prediction, the performance was better and the F0.5-Score of two models improved by 0.05 on average with multi-task learning and prediction. For failure type prediction and remaining lifespan prediction, we used the accuracy rate to evaluate the performance, since both tasks are multi-classification tasks and they only make sense when SSD failures are correctly predicted. The accuracy rate is defined as the proportion of SSDs with correctly predicted failure type (or remaining lifespan) to all correctly predicted failed SSDs. After using multi-task learning, Table 5 shows that the accuracy rate of two models for failure type prediction and remaining lifespan prediction increased by 0.04 and 0.09 on average, respectively. In conclusion, multi-task learning and prediction boosted the model's performance on three tasks.

Table 5 shows that our MVTRF with multi-task learning achieved an accuracy rate of 0.95 in failure type prediction and 0.55 in remaining lifespan prediction. It demonstrates that both predictions are effective. According to the urgency of different failure types and the remaining lifespan, operators can decide whether to directly replace the SSD or further analyze it, so that the failures can be handled in a timely and accurate manner.

Table 5: Comparison of single-task learning and multi-task learning.

Method		P	R	F	AUC	Type Acc	Lifespan Acc
RF + Raw	**Single-task**	0.58	0.31	0.48	0.69	0.88	0.44
	Multi-task	0.61	0.34	0.52	0.69	0.93	0.53
MVTRF	**Single-task**	0.83	0.37	0.66	0.79	0.93	0.47
	Multi-task	**0.90**	**0.40**	**0.72**	**0.81**	**0.95**	**0.55**

(P: precision; R: recall; F: F0.5-Score; AUC: ROC_AUC; Acc: accuracy rate)

Another benefit of using a single model for multi-task learning is that it can reduce model training and prediction time compared with using three models to predict three tasks. Table 6 shows the dimensions of different features, and compares the total time required for separate training/prediction and joint training/prediction on the three tasks based on these features. Table 6 reveals that adopting multi-task learning can reduce training/prediction time in most cases. It also shows that the training/prediction time of MVTRF mainly depends on the training/prediction time of the combined features with the highest dimension. In addition, MVTRF with multi-task learning completes the prediction of one million Telemetry data within three minutes and thus can fully support the online real-time prediction of large-scale SSDs.

Table 6: Total training/prediction time of single-task model and multi-task model.

Method	Feature NO.	Training time(s)		Prediction time(s)	
		Single	Multi	Single	Multi
RF + Raw	26	1230.9	599.8	36.5	62.8
RF + Histogram	102	1852.1	978.5	171.6	111.2
RF + Sequence	104	2867.9	1378.1	65.2	69.7
RF + Combined	232	3171.9	1707.2	232.2	130.9
MVTRF	464	3262.7	1775.2	245.2	143.0

(Train and predict on one million data.)

4.4 Similar Decision Extraction

According to the decision process of MVTRF model, we propose SDE to obtain key decisions and find the failure causes (see Section 3.4). Table 7 shows the key decisions extracted from the decision process of a failed SSD. SDE extracts five key decisions from a total of original 3,825 decisions and gives them weights (as described in Section 3.4, the weight is the number of similar decisions). The extracted key decisions can certainly identify this failed SSD, but may lead to false alarms due to the large reduction in joint decisions. We reapplied these key decisions to all data to evaluate their effectiveness based on the false alarms introduced by them. Table 7 shows that the decision with the highest weight only had three false alarms, which indicates that the extracted key decisions have a strong distinguishing ability. Then, all false alarms were eliminated by combining subsequent key decisions. It can be concluded that the decisions extracted by the SDE approach are critical and they can represent the major decision process. According to the key decisions, we figured out that the direct cause of this failure was the rapid increase of media errors ($\mathit{media\_errors\_slop} > 126.44$ and $\mathit{media\_errors} > 6015.5$), and thus it was verified to be an internal failure of the SSD. In addition, the changes of temperature and wear leveling may be potential factors ($\mathit{temperature\_kurt} <= -1.11$ and $\mathit{wear\_leveling\_max\_kurt} > -0.047$).

Table 7: Key decisions of an SSD failure. False alarms were reduced with the combination of key decisions.

	Key decision	Feature type	Weight	False alarms
①	$\mathit{media\_errors\_slope} > 126.44$	Sequence	122	3 (①)
②	$\mathit{media\_errors\_bkt}0 <= 0.99$	Histogram	120	3 (① - ②)
③	$\mathit{temperature\_kurt} <= -1.11$	Sequence	117	1 (① - ③)
④	$\mathit{media\_errors} > 6015.5$	Raw	113	1 (① - ④)
⑤	$\mathit{wear\_leveling\_max\_kurt} > -0.047$	Sequence	96	0 (① - ⑤)

We also extracted several sets of key decisions from the judgment process of all failed SSDs to evaluate the overall discriminative ability of key decisions, as shown in Table 8. It shows there were 53,663 decisions in total for failed SSDs, and our SDE approach extracts 49 key decisions. Reapplying these key decisions to all data achieved the same precision and recall as all original decisions. The 49 key decisions performed almost the same as the original 53,663 decisions in distinguishing failed SSDs and healthy SSDs, which illustrates the effectiveness of the proposed SDE approach. Then, the failure causes can be identified and analyzed based on these decisions, which lays the foundation for verifying and handling SSD failures.

Table 8: Comparison of key decisions with all decisions.

	Decision NO.	Precision	Recall
All decisions	53663	0.90	0.40
Key decisions	49	0.90	0.40

5 Related Work

Many previous studies have investigated and analyzed the impact of drive errors and failures on large data centers [13, 15, 34, 35, 37, 43, 44]. In order to take proactive measures (such as replacing drives) before failures occur, drive failure prediction has received extensive attention and research. Since HDDs have been widely used for a long time, there are many works on HDD failure prediction [9, 11, 18, 21, 26, 36, 39, 42, 46, 49, 51, 52]. Most of these works [9, 18, 21, 26, 39, 42, 51, 52] are based on short-term monitoring data, as the symptoms of HDD failure generally appear days or hours leading up to the failure [24]. Unlike SSDs that are based on electrical signals, HDDs are mechanically based, and their problems would quickly develop into serious failures.

In recent years, with the popularization of SSDs, more and more research studies have been done on SSD failure prediction [3, 7, 16, 22, 27, 30, 33, 40, 45, 50]. Alter [3] et al. adopted classification algorithms to predict SSD failures based on machine learning algorithms, including logistic regression, support vector machine, random forest, and neural network. They also analyzed the failure characteristics of SSDs in different periods. Chandranil et al. [7] introduced the unsupervised anomaly detection algorithms, isolation forest and autoencoder, to predict SSD failures. These algorithms only learn the patterns of healthy SSDs and consider the ones with large pattern differences to be failed SSDs. Hao et al. [16] introduced LSTM, a recurrent neural network, to capture failure symptoms from the sequences of monitoring data. In addition, they proposed Ensemble LSTM to enhance the prediction accuracy through ensemble learning. Xu [45] et al. studied the impact of feature selection algorithms on SSD failure prediction. They proposed a feature selection approach, Wear-out-updating Ensemble Feature Ranking (WEFR), to improve the performance of random forest algorithm by selecting SMART attributes with strong representational ability.

6 Conclusions

In this paper, we propose multi-view and multi-task random forest (MVTRF) to predict SSD failures and other failure information based on short-term and long-term monitoring data. We observed that some failure symptoms are hidden in the distribution and trend of long-term data, and thus histogram features and sequence-related features were introduced. MVTRF learns these features and short-term data in parallel through multiple sets of decision trees, thereby integrating multi-view information to find more failures and reduce false alarms. In addition, we adopted multi-task learning to allow a single model to learn and predict detailed failure information, including failure type and remaining lifespan. We also propose similar decision extraction (SDE) to obtain the key decisions from MVTRF to identify and analyze the failure causes. These details help operators to quickly verify the failure and recommend appropriate actions to handle it more efficiently. Our evaluation on real data from data centers showed that MVTRF significantly improves the accuracy of failure prediction and can predict the failure type and remaining lifespan of SSDs simultaneously and effectively.

References

[1] Multiclass and multioutput algorithms. https://scikit-learn.org/stable/modules/multiclass.html.

[2] Hervé Abdi. Coefficient of variation. *Encyclopedia of research design*, 1:169–171, 2010. https://www.utdallas.edu/~herve/abdi-cv2010-pretty.pdf.

[3] Jacob Alter, Ji Xue, Alma Dimnaku, and Evgenia Smirni. SSD Failures in the Field: Symptoms, Causes, and Prediction Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3295500.3356172.

[4] Mirela Madalina Botezatu, Ioana Giurgiu, Jasmina Bogojeska, and Dorothea Wiesmann. Predicting Disk Replacement towards Reliable Data Centers. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 39–48, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2939672.2939699.

[5] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. https://doi.org/10.1023/A:1010933404324.

[6] Yu Cai, Yixin Luo, Saugata Ghose, and Onur Mutlu. Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 438–449, June 2015. https://doi.org/10.1109/DSN.2015.49.

[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 '20, page 120–133, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3419111.3421300.

[8] Saeideh Alinezhad Chamazcoti, Bardia Safaei, and Seyed Ghassem Miremadi. Can Erasure Codes Damage Reliability in SSD-Based Storage Systems? *IEEE Transactions on Emerging Topics in Computing*, 7(3):435–446, July 2019. https://doi.org/10.1109/TETC.2017.2693424.

[9] Iago C. Chaves, Manoel Rui P. de Paula, Lucas G.M. Leite, Lucas P. Queiroz, Joao Paulo P. Gomes, and Javam C. Machado. BaNHFaP: A Bayesian Network Based Failure Prediction Approach for Hard Disk Drives. In *2016 5th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 427–432, October 2016. https://doi.org/10.1109/BRACIS.2016.083.

[10] L. T DECARLO. On the meaning and use of kurtosis. *Psychological methods*, 2(3):292–307, 1997. https://doi.org/10.1037/1082-989X.2.3.292.

[11] Yan Ding, Yunan Zhai, Yujuan Zhai, and Jia Zhao. Explore deep auto-coder and big data learning to hard drive failure prediction: a two-level semi-supervised model. *Connect. Sci.*, 34(1):449–471, 2022. https://doi.org/10.1080/09540091.2021.2008320.

[12] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006. https://doi.org/10.1016/j.patrec.2005.10.010.

[13] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 139–152, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2785956.2787496.

[14] Shujie Han, Patrick P. C. Lee, Zhirong Shen, Cheng He, Yi Liu, and Tao Huang. Toward Adaptive Disk Failure Prediction via Stream Mining. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 628–638, November 2020. https://doi.org/10.1109/ICDCS47774.2020.00044.

[15] Shujie Han, Patrick P. C. Lee, Fan Xu, Yi Liu, Cheng He, and Jiongzhou Liu. An In-Depth Study of Correlated Failures in Production SSD-Based Data Centers. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 417–429. USENIX Association, February 2021. https://www.usenix.org/conference/fast21/presentation/han.

[16] Wenwen Hao, Ben Niu, Yin Luo, Kangkang Liu, and Na Liu. Improving accuracy and adaptability of SSD failure prediction in hyper-scale data centers. *SIGMETRICS Perform. Eval. Rev.*, 49(4):99–104, June 2022. https://doi.org/10.1145/3543146.3543169.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997. https://doi.org/10.1162/neco.1997.9.8.1735.

[18] G.F. Hughes, J.F. Murray, K. Kreutz-Delgado, and C. Elkan. Improved disk-drive failure warnings. *IEEE*

Transactions on Reliability, 51(3):350–357, September 2002. https://doi.org/10.1109/TR.2002.802886.

[19] Massimo Iaculo, Francesco Falanga, and Ornella Vitale. Introduction to SSD. *Memory Mass Storage*, pages 213–236, 2011. https://doi.org/10.1007/978-3-642-14752-4_5.

[20] Pedro Lara-Benítez, Manuel Carranza-García, and José C.Riquelme. An Experimental Review on Deep Learning Architectures for Time Series Forecasting. *CoRR*, abs/2103.12057, 2021. https://arxiv.org/abs/2103.12057.

[21] Jing Li, Xinpu Ji, Yuhan Jia, Bingpeng Zhu, Gang Wang, Zhongwei Li, and Xiaoguang Liu. Hard Drive Failure Prediction Using Classification and Regression Trees. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 383–394, June 2014. https://doi.org/10.1109/DSN.2014.44.

[22] Peng Li, Wei Dang, Congmin Lyu, Min Xie, Quanyang Bao, Xiaofeng Ji, and Jianhua Zhou. Reliability Characterization and Failure Prediction of 3D TLC SSDs in Large-Scale Storage Systems. *IEEE Transactions on Device and Materials Reliability*, 21(2):224–235, June 2021. https://doi.org/10.1109/TDMR.2021.3063164.

[23] Fernando Dione dos Santos Lima, Gabriel Maia Rocha Amaral, Lucas Gonçalves de Moura Leite, João Paulo Pordeus Gomes, and Javam de Castro Machado. Predicting Failures in Hard Drives with LSTM Networks. In *2017 Brazilian Conference on Intelligent Systems (BRACIS)*, pages 222–227, October 2017. https://doi.org/10.1109/BRACIS.2017.72.

[24] Sidi Lu, Bing Luo, Tirthak Patel, Yongtao Yao, Devesh Tiwari, and Weisong Shi. Making Disk Failure Predictions SMARTer! In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 151–167, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/fast20/presentation/lu.

[25] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. Improving 3D NAND Flash Memory Lifetime by Tolerating Early Retention Loss and Process Variation. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(3), December 2018. https://doi.org/10.1145/3224432.

[26] Ao Ma, Rachel Traylor, Fred Douglis, Mark Chamness, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures. *ACM Trans. Storage*, 11(4), November 2015. https://doi.org/10.1145/2820615.

[27] Farzaneh Mahdisoltani, Ioan Stefanovici, and Bianca Schroeder. Proactive error prediction to improve storage system reliability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 391–402, Santa Clara, CA, July 2017. USENIX Association. https://www.usenix.org/conference/atc17/technical-sessions/presentation/mahdisoltani.

[28] Puneet Misra and Arun Singh Yadav. Improving the Classification Accuracy using Recursive Feature Elimination with Cross-Validation. *Int. J. Emerg. Technol*, 11(3):659–665, 2020. http://www.puneetmisra.com/admin/uploads/journals/5f136d202b8ba1.18644117.pdf.

[29] Joseph F. Murray, Gordon F. Hughes, and Kenneth Kreutz-Delgado. Machine Learning Methods for Predicting Failures in Hard Drives: A Multiple-Instance Application. *Journal of Machine Learning Research*, 6(27):783–816, 2005. http://jmlr.org/papers/v6/murray05a.html.

[30] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR '16, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2928275.2928278.

[31] A. Neubeck and L. Van Gool. Efficient Non-Maximum Suppression. In *18th International Conference on Pattern Recognition (ICPR'06)*, volume 3, pages 850–855, August 2006. https://doi.org/10.1109/ICPR.2006.479.

[32] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, page 109–116, New York, NY, USA, 1988. Association for Computing Machinery. https://doi.org/10.1145/50202.50214.

[33] Jay Sarkar, Cory Peterson, and Amir Sanayei. Machine-learned assessment and prediction of robust solid state storage system reliability physics. In *2018 IEEE International Reliability Physics Symposium (IRPS)*, pages 3C.6–1–3C.6–8, March 2018. https://doi.org/10.1109/IRPS.2018.8353565.

[34] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 67–80, Santa Clara, CA, February 2016. USENIX Association. https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder.

[35] Bianca Schroeder, Arif Merchant, and Raghav Lagisetty. Reliability of nand-Based SSDs: What Field Studies Tell Us. *Proceedings of the IEEE*, 105(9):1751–1769, September 2017. https://doi.org/10.1109/JPROC.2017.2735969.

[36] Jing Shen, Yongjian Ren, Jian Wan, and Yunlong Lan. Hard Disk Drive Failure Prediction for Mobile Edge Computing Based on an LSTM Recurrent Neural Network. *Mobile Information Systems*, 2021:1–12, February 2021. https://doi.org/10.1155/2021/8878364.

[37] Guosai Wang, Lifei Zhang, and Wei Xu. What Can We Learn from Four Years of Data Center Hardware Failures? In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36, June 2017. https://doi.org/10.1109/DSN.2017.26.

[38] 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)*, pages 413–424, Boston, MA, June 2012. USENIX Association. https://www.usenix.org/conference/atc12/technical-sessions/presentation/wang.

[39] Yu Wang, Eden W. M. Ma, Tommy W. S. Chow, and Kwok-Leung Tsui. A Two-Step Parametric Method for Failure Prediction in Hard Disk Drives. *IEEE Transactions on Industrial Informatics*, 10(1):419–430, February 2014. https://doi.org/10.1109/TII.2013.2264060.

[40] Debao Wei, Liyan Qiao, Mengqi Hao, Hua Feng, and Xiyuan Peng. Reliability prediction model of NAND flash memory based on random forest algorithm. *Microelectronics Reliability*, 100-101:113371, 2019. https://www.sciencedirect.com/science/article/pii/S002627141930472X.

[41] Graham Williams. Random forests. In *Data Mining with Rattle and R*, pages 245–268. Springer, 2011. https://doi.org/10.1007/978-1-4419-9890-3_12.

[42] Jiang Xiao, Zhuang Xiong, Song Wu, Yusheng Yi, Hai Jin, and Kan Hu. Disk Failure Prediction in Data Centers via Online Learning. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3225058.3225106.

[43] Erci Xu, Mai Zheng, Feng Qin, Jiesheng Wu, and Yikang Xu. Understanding SSD Reliability in Large-Scale Cloud Systems. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 45–53, November 2018. https://doi.org/10.1109/PDSW-DISCS.2018.00010.

[44] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and Actions: What We Learned from 10K SSD-Related Storage System Failures. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 961–976, Renton, WA, July 2019. USENIX Association. https://www.usenix.org/conference/atc19/presentation/xu.

[45] 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 *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 263–270, June 2021. https://doi.org/10.1109/DSN48987.2021.00039.

[46] Yong Xu, Kaixin Sui, Randolph Yao, Hongyu Zhang, Qingwei Lin, Yingnong Dang, Peng Li, Keceng Jiang, Wenchi Zhang, Jian-Guang Lou, Murali Chintalapati, and Dongmei Zhang. Improving Service Availability of Cloud Systems by Predicting Disk Error. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 481–494, Boston, MA, July 2018. USENIX Association. https://www.usenix.org/conference/atc18/presentation/xu-yong.

[47] Jinpei Yan, Yong Qi, Qifan Rao, and Tom Chen. LSTM-Based Hierarchical Denoising Network for Android Malware Detection. *Sec. and Commun. Netw.*, 2018, January 2018. https://doi.org/10.1155/2018/5249190.

[48] Ji Hyuck Yun, Jin Hyuk Yoon, Eyee Hyun Nam, and Sang Lyul Min. An Abstract Fault Model for NAND Flash Memory. *IEEE Embedded Systems Letters*, 4(4):86–89, December 2012. https://doi.org/10.1109/LES.2012.2213235.

[49] Ying Zhao, Xiang Liu, Siqing Gan, and Weimin Zheng. Predicting Disk Failures with HMM- and HSMM-Based Approaches. In *Proceedings of the 10th Industrial Conference on Advances in Data Mining: Applications and Theoretical Aspects*, ICDM'10, page 390–404, Berlin, Heidelberg, 2010. Springer-Verlag. https://doi.org/10.1007/978-3-642-14400-4_30.

[50] 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 *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, pages 1–10, June 2021. https://doi.org/10.1109/IWQOS52092.2021.9521302.

[51] Bingpeng Zhu, Gang Wang, Xiaoguang Liu, Dianming Hu, Sheng Lin, and Jingwei Ma. Proactive drive failure prediction for large scale storage systems. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, May 2013. https://doi.org/10.1109/MSST.2013.6558427.

[52] Marwin Züfle, Florian Erhard, and Samuel Kounev. Machine Learning Model Update Strategies for Hard Disk Drive Failure Prediction. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1379–1386, December 2021. https://doi.org/10.1109/ICMLA52953.2021.00223.

Fast Application Launch on Personal Computing/Communication Devices

Junhee Ryu[1], Dongeun Lee[2], Kang G. Shin[3], and Kyungtae Kang[4]
[1]*SK hynix,* [2]*Texas A&M University - Commerce,* [3]*University of Michigan,* [4]*Hanyang University*

Abstract

We present Paralfetch, a novel prefetcher to speed up app launches on personal computing/communication devices by: 1) accurate collection of launch-related disk read requests, 2) pre-scheduling of these requests to improve I/O throughput during prefetching, and 3) overlapping app execution with disk prefetching for hiding disk access time from the app execution. We have implemented Paralfetch under Linux kernels on a desktop/laptop PC, a Raspberry Pi 3 board, and an Android smartphone. Tests with popular apps show that Paralfetch significantly reduces app launch times on flash-based drives, and outperforms *GSoC Prefetch* and FAST, which are representative app prefetchers available for Linux-based systems.

1 Introduction

Quick app launches are of great importance to user experience on personal computing/communication devices such as laptop/tablet PCs, single-board computers, and smartphones [17, 18, 22, 24, 26, 34]. The latency of launching an app mainly depends on the performance of the underlying CPU and flash-based disks. Despite continuing improvements in the performance of these components, the launch latencies, especially of large apps and games, still remain an important problem for three reasons.

First, the performance of flash storage does not always meet users' expectations/desire. For example, it has been predicted [53] that in 2025 around 50% of the data on flash will be stored in QLC (quad-level cell) flash, which has 2.1× slower read and 5.7× slower write times than TLC (triple-level cell) flash [4]. The use of affordable QLC SSDs was found to extend the launch latency of the popular Blade and Soul game from 91s to 114s [46], and that of Horizon Zero Dawn from 15.7s to 21.4s [47], compared to high-end SSDs. Many Windows apps take a similar amount of time [48] to launch from the Samsung QLC SSD as they do from the Intel X25-M G2 SSD, which was released in 2009. Furthermore, recent entry-class SSDs widely adopt DRAM-less architecture [35], which leads to additional flash accesses for translating logical-to-physical addresses. A Raspberry Pi is also widely used to run desktop applications [57], but it only supports the sluggish MicroSD.

Second, the complexity of apps is continuously growing due to the addition of new features and functionality to software [50]. Unfortunately, complex software also requires higher-level programming languages and libraries, generating slower code, thus extending their launch latencies [54].

Third, although parallelism is effectively utilized in modern multicore CPUs and solid-state disks [8], app launches can seldom exploit existing sources of parallelism. It has also been shown [25] that CPUs and disks are seldom utilized simultaneously during a launch because synchronous disk reads are dominant. Making better use of parallelism is, therefore, a major consideration in the design of app prefetchers [24].

Launch latencies depend on the previous state of the system, especially the disk cache. A *cold start* occurs when the disk cache does not hold any data required by the app, either because it is the first time the app has been launched, or because all of the app's data has been evicted since its last run. A *system cold start* is a special case of cold start, which occurs when no user-launched app is already running. A *warm start* occurs when the app being launched has been running recently, so the disk cache still holds all, or most, of the data that it needs. A warm start is much faster than a cold start, because no, or very little, data has to be fetched from the disk. This avoids the concomitant file system operations, thus saving CPU time as well as disk time.

An app prefetcher [6, 7, 9, 11, 28, 36, 40] can reduce the time needed for a cold start: during *learning phase*, which corresponds to the first launch of an app, the prefetcher collects launch-related blocks and/or their access sequences (the term *launch sequence* is used interchangeably). This is usually achieved by monitoring disk reads and/or page faults. A *prefetching phase* occurs during subsequent launches of the app, in which case this launch sequence is used for disk prefetching to accelerate loading.

Different prefetching strategies are required for the different seek characteristics of mechanical and flash disks. These storage devices have different performance bottlenecks which have been addressed in well-known ways. *Threaded prefetching* is designed for SSDs. A dedicated thread is used to prefetch blocks in the order of their collection during monitoring. The prefetching thread runs concurrently with the app, reducing the launch time. On the other hand, *Sorted prefetching* is designed for HDDs. Data is read from the disk in logical block address (LBA) order to reduce seek times [5, 19, 20], which account for most of the launch time. Sorted prefetching is not done concurrently with the app because the app's disk I/O would disrupt prefetching in the LBA sequence.

In this paper, we define three fundamental challenges in reaping the potential speed-up with an app prefetcher, and then explain how Paralfetch addresses these issues that previous approaches fail to achieve. Overall, this paper makes the following main contributions:

- **Accurate tracking of launch-related blocks (§3.1):** Most monitoring methods fail to locate a significant number of blocks during the learning phase [23]. In threaded prefetching on SSDs, an access tracer should collect not only accessed blocks but their access order. To do this, a viable solution is to monitor at the disk I/O level after performing the invalidation of unused entries in the disk cache. Unfortunately, metadata and data blocks would not be detected by imperfect OS-level disk cache invalidation. To address this problem, Paralfetch introduces a file-system-level block dependency check and low-overhead page-fault monitoring.
- **Pre-scheduling of these blocks to increase prefetch throughput (§3.2):** Although the I/O involved in prefetching frequently becomes a bottleneck in threaded prefetching on commodity SSDs, prior work does not address this issue. We observe I/O dependencies between prefetch blocks to significantly hinder the asynchrony of I/O requests, reducing prefetch throughput. We address this problem with a new I/O reordering method called *metadata shift* that places more I/O requests between dependent I/O requests, issuing more I/O requests asynchronously. A *range merge* is also introduced to combine nearby I/O requests into one large request, improving I/O throughput.
- **Tailored overlapping of application execution with prefetching (§3.3)**: We find that aggressive prefetching with excessive pre-scheduling can actually increase launch latencies because of I/O contention between the app and prefetching threads. Modern SSDs' reordering of outstanding I/O operations can aggravate this contention [41]. We vary the amount of I/O optimization in response to a prefetching bottleneck. This avoids the I/O contention caused by an excessive optimization, and thus helps Paralfetch find a better optimization level.
- **Implementation (§4) and evaluation (§5) of Paralfetch:** We evaluate Paralfetch in the launch of common apps on a laptop PC, a Raspberry Pi 3, and an Android smartphone. With the aforementioned features, Paralfetch achieves launch performance close to the warm start: On a PC, Paralfetch reduced the average system cold start time (favoring competitors) of 16 benchmark apps by 48.0%, this number corresponds to 11% and 22% further reductions from FAST and GSoC Prefetch, respectively. Paralfetch also reduced the average app launch time on a Raspberry Pi 3 by 31%, and on an Android phone by 11%. Paralfetch is publicly available[1]

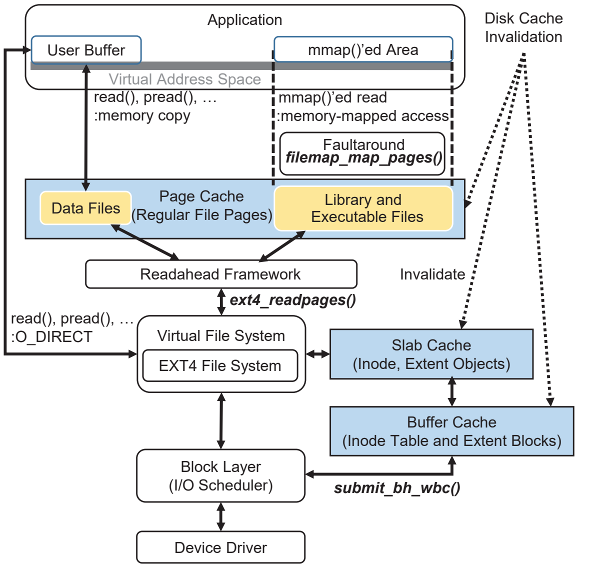


Figure 1: I/O Stack in Linux. Linux includes three disk caches: page cache for regular files, slab (or slub) cache for metadata objects, and buffer cache for metadata blocks. The slab is used as an object-granular metadata cache for buffer cache. read system call explicitly fills page cache based on its arguments, while page cache for mmaped files is populated through page fault mechanism. Readahead framework is responsible for filling the contents of page cache, and it determines how many blocks to be prefetched based on the access sequentiality. Note that metadata blocks can be prefetched by EXT4 file system.

2 Background and Motivation

2.1 Targets of Paralfetch

Linux-based systems using EXT4 file system. We implemented and tested a Paralfetch prototype on EXT4 file system on a laptop with SSD, a Raspberry Pi 3 with microSD card, and a Pixel smartphone with universal flash storage (UFS).

Large apps with highly deterministic I/O. Other applications do not benefit much from Paralfetch: I/O requests from text-based apps such as cp, gcc and find largely depend on input parameters that can change with every launch; and apps such as pwd and ssh are too small to amortize prefetch overhead, and are usually warm started.

2.2 Disk Caching in Linux

Figure 1 provides a summary of the Linux I/O stack from disk caching perspectives.

Page cache and buffer cache. The Linux kernel provides two cache mechanisms for disk blocks in terms of API and unit size [15]: The *page cache* holds file pages, whereas the *buffer cache* contains data blocks corresponding to block devices. The contents and lookup spaces of these caches are managed using a radix tree for each regular file or block device file.

[1] https://github.com/optios/paralfetch

In EXT4 file system, blocks of data from regular files are cached in the page cache, while the buffer cache is used for caching metadata blocks (*e.g.*, inode table blocks, directory blocks, and extent blocks). The contents of regular files can be prefetched using a combination of *device number*, *inode number*, *offset*, and *size*. On the other hand, metadata blocks can be prefetched using a combination of *device number* and *block number*. It should be noted that there are no prefetching-level dependencies among buffer-cached (metadata) blocks, whereas I/O requests for page-cached (data) blocks are delayed until relevant metadata blocks are cached.

Slab for caching file system metadata at object granularity. Metadata objects in EXT4 file system, namely, the inode, directory entry, and extent, are smaller than a file system block but must nevertheless be managed individually so that important objects are kept in memory, even when the memory is under pressure. Therefore, the Linux slab object allocator caches these objects without reference to the contexts of the buffer cache. Thus an `inode` can be simultaneously stored in both the slab and buffer caches.

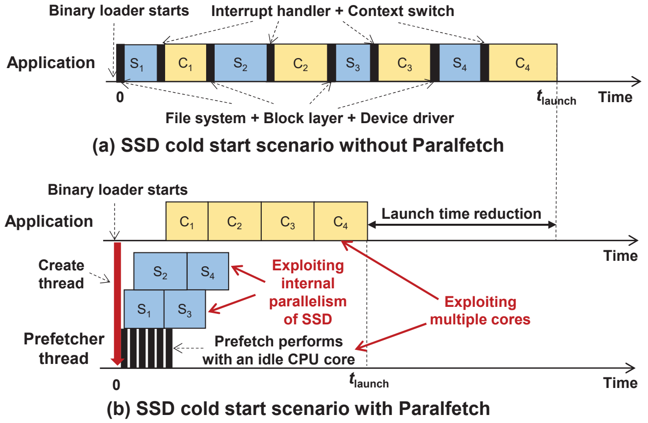
Page cache accessing methods. A process can copy the contents of the page cache into a user buffer using a `read` or a file-related syscall. Alternatively, a process can map the extent of a file to its virtual address space using the `mmap` syscall. In the latter case, attempting to access an unmapped address in the page table causes a page fault. To reduce the number of page faults, Linux employs an interesting feature, called *faultaround* [49], which pre-faults a 64KB-aligned chunk of the address space around the fault address.

Disk cache invalidation. The Linux kernel provides functions to invalidate disk caches. A user or process with root permission can invalidate these caches by writing a predefined value ("1" for the page and buffer caches, "2" for the slab cache, and "3" for all these) into the `/proc/sys/vm/drop_caches` proc file. This method can only invalidate unused entries with zero reference counts.

2.3 Representative App Prefetchers

Windows prefetcher [37]**.** Since XP, Windows has included a prefetcher for launch and system boot. The *Windows prefetcher* is customized for HDDs, but it can also be used with SSDs, although user configuration is required to make best use of more capable SSDs. In its learning phase, the copies of file-backed memory pages which are required by an application are identified by the Windows working-set manager. The generated information, which is file-level data, determines the disk blocks to be prefetched during subsequent application launches. By defragmenting these blocks to make their file-level prefetch blocks correspond to their LBA order, the Windows prefetcher optimizes the disk head movements of HDD. This time-consuming process is scheduled to happen every three days.

GSoC Prefetch [29], which was selected for the Google Summer of Code 2007, is a Linux-based prefetcher for HDDs. It obtains launch-related block information in its learning phase by first clearing the bit in every OS-managed page descriptor (not page table) which indicates that the page has been referenced. After a predefined monitoring time (10 seconds by default), GSoC Prefetch traces those referenced pages with 'referenced' bits on. It then extracts a file identifier (device number, inode number, and offset) from each of the traced pages. Next, GSoC Prefetch sorts the pages based on these identifiers and stores the sorted pages in a file. On subsequent launches, launch-related blocks are prefetched in the order recorded in that file. This reduces both seek and rotational latencies in HDDs. GSoC Prefetch has a defragmentation tool similar to that in the Windows prefetcher.


Binary loader starts
Interrupt handler + Context switch
Application
S1
C1
S2
C2
S3
C3
S4
C4
0
t_launch
Time
File system + Block layer + Device driver
Binary loader starts
Application
C1
C2
C3
C4
Launch time reduction
Time
Create thread
S2
S4
S1
S3
Exploiting internal parallelism of SSD
Exploiting multiple cores
Prefetcher thread
Prefetch performs with an idle CPU core
0
t_launch
Time

(a) SSD cold start scenario without Paralfetch

(b) SSD cold start scenario with Paralfetch

Figure 2: SSD cold start scenarios with and without Paralfetch. S_i is the i^{th} block requested from the SSD, and C_i is the corresponding CPU computation. Paralfetch expedites an application launch by exploiting parallelism of each resource (i.e., multicore activation and internal parallelism on SSDs) and utilizing these resources concurrently.

FAST [24] is a recent Linux-based prefetcher for SSDs. It starts by clearing the slab, buffer, and page caches. Then, `FAST` begins its learning phase, during which it creates a prefetch program by monitoring the LBAs of blocks using the `blktrace` tool and converting them to prefetchable system calls with arguments. On subsequent launches, `FAST` executes this prefetch program at the same time as the application. Disk blocks are prefetched in order without any I/O optimization.

2.4 Cold Start with Paralfetch

Figure 2a shows a cold start scenario without `Paralfetch`, and Figure 2b shows the same scenario in which `Paralfetch` runs the application concurrently with a prefetch thread. The computations run on multiple CPU cores, in parallel with the SSD accesses, which are issued in a way that exploits the internal parallelism of the SSD. This is effected by issuing concurrent asynchronous I/O requests using the command queuing (CQ) feature. If an SSD does not support CQ, `Paralfetch` merges I/O requests, which have consecutive LBAs and are close in the block access sequence, so as to promote internal parallelism.

Table 1: Metadata and data block requests required to launch applications with missing metadata blocks. Note that 'regular' files include `mmap`ed files, and that files `mmap`ed by running applications are not subject to disk cache invalidation. The last column shows the number of I/O operations that were not captured by Paralfetch, which varies from run to run.

Application		Read requests traced by `Paralfetch`		Number of missing	Number of accessed files		
		Metadata accesses	File data accesses	metadata blocks	regular	`mmap`ed	Number of
		(total size in KB)	(total size in KB)	detected	files	files	missing I/Os
Ubuntu Linux (Laptop PC)	Android Studio	1,330 (6,844)	3,845 (197,932)	58	954	10	38
	Chromium Browser	612 (3,048)	1,135 (130,728)	37	629	108	34
	Eclipse	565 (3,348)	1,669 (67,256)	28	744	328	49
	GIMP	489 (2,620)	1,026 (38,512)	20	975	474	28
	LibreOffice Impress	590 (2,900)	706 (83,004)	37	438	232	32
	LibreOffice Writer	552 (2,800)	729 (83,824)	25	476	227	33
	Okular	1,093 (5,720)	426 (23,640)	41	349	238	36
	Scribus	840 (5,984)	1,560 (141,056)	35	1,230	682	21
	VLC Player	682 (5,420)	444 (20,192)	41	375	104	32
	Xilinx ISE	573 (3,024)	1,028 (176,504)	42	657	273	33
Raspbian OS (Raspberry Pi 3)	Chromium Browser	496 (1,984)	2,017 (138,600)	40	473	68	41
	Frozen Bubble	605 (2,420)	3,769 (32,992)	25	3,425	26	12
	GIMP	618 (2,472)	1,863 (46,664)	38	991	296	47
	LibreOffice Writer	596 (2,384)	911 (35,164)	33	395	154	36
	Scratch 2	332 (1,328)	839 (48,580)	40	294	73	19
	Xpdf	127 (508)	169 (7,236)	15	75	21	11
	0 A.D.	206 (509)	669 (86,272)	19	162	139	21
Android 8.0 (Google Pixel XL)	Asphalt 8	131 (988)	838 (217,240)	49	179	N/A	11
	Dragon Quest 8	95 (852)	4,339 (333,812)	46	335	N/A	12
	FIFA 16 UT	76 (772)	805 (166,120)	39	265	N/A	47
	GTA SA	104 (560)	377 (82,928)	41	95	N/A	36
	Truck Pro	96 (792)	1,792 (115,732)	41	175	N/A	19
	Devil May Cry	237 (1,728)	1,904 (316,004)	45	407	N/A	19
	The War of Mine	127 (696)	517 (128,300)	43	101	N/A	11

3 Paralfetch Design and Preliminary Results

3.1 Accurate Tracing

The benefit from an application prefetching is limited by the tracing accuracy with which launch-related blocks are traced. In particular, accurate tracing is essential to prevent a launching application's wait for missing blocks from disk when several concurrent threads are causing lots of I/O contention. Note that the threaded prefetching can marginally benefit from Windows prefetcher and GSoC Prefetch which cannot trace the block access sequence because they rely on a snapshot of the working set or of the referenced pages after a launch.

There are also issues with the tracing method used by GSoC Prefetch: it only traces pages for regular files, and missing metadata limits the benefit of prefetching; a significant number of pages are also accessed more than once during a launch. This latter issue is particularly problematic because, when a page with the 'referenced' bit set on is accessed for the second time, Linux OS turns off the 'referenced' bit and promotes the page from the inactive list to the active list. As a result, some pages are never traced. In the case of Eclipse, we found 2,782 file-backed pages not traced.

Potentially, the highest accuracy would be achieved by monitoring page faults and data accesses at all disk caching layers (*e.g.*, slab, buffer, and page caches). But such exhaustive tracing would produce significantly more data than I/O-level monitoring (37× during an Eclipse launch), incurring unacceptable memory and computation overheads. Furthermore, a log of I/O operations obtained by monitoring disk cache accesses is likely to include many useless cached entries created by I/O operations of background tasks.

This issue is successfully mitigated by monitoring I/O requests: In the learning phase, `Paralfetch` invalidates unused entries in the disk cache, so that `Paralfetch` collects a proper set of blocks for subsequent launches of the application. It then records I/O requests for blocks not found in these caches by instrumenting file system functions with I/O logging codes, and these requests are used to prefetch those additional blocks during launches. In this paper, we use the term *log entry* to refer to a log of I/O request collected during a launch, while the term *prefetch entry* refers to an entry used for prefetching disk blocks. The latter includes arguments for prefetching function calls.

Unfortunately, as mentioned earlier, the invalidation of disk caches (slab, buffer, and page caches) is not perfect because only unused entries can be invalidated; a working set of blocks for running applications is always retained. This issue has been overlooked in previous schemes (including `FAST`), i.e., their evaluation was restricted to system cold start scenarios. Table 1 classifies traced blocks with `Paralfetch`. Note that metadata blocks and `mmap`ed file blocks are potential missing blocks when using `FAST`. Since usually many user and system processes run in the background, this issue can significantly degrade tracing accuracy. For example, 225 files of this kind were accessed by both LibreOffice Impress and LibreOffice Writer (on a laptop) during a launch of either. Thus, an attempt to trace launch blocks for LibreOffice Writer just after LibreOffice Impress launched (and started running in the background) returns only 700 log entries (27,688 KB) compared to 1,281 log entries (83,824 KB) during a system cold start. We conducted further experiments by substituting Android Studio, Chromium Browser, Eclipse, and GIMP for LibreOffice Impress. Surprisingly, imperfect cache invalidation still resulted in many missing data and associated metadata blocks: 5.0%, 12.0%, 14.4%, and 6.6% of the total in each case. The launch time impact of missing blocks is significant as shown in §5.2.

We have therefore developed two methods to detect missing metadata and data blocks.

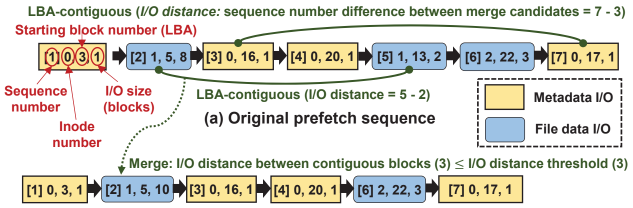
1) Finding missing metadata blocks. We first introduce a file system-level dependency check, called *missing metadata block detection*, which identifies launch-related metadata blocks (*i.e.*, inode and extent blocks) that have not been traced due to the imperfect invalidation of the slab and buffer cache, but nevertheless share a dependency with traced data blocks. To address this issue, Paralfetch implements a function (§4.2) that tracks associated metadata blocks for each log entry for a regular file. Table 1 shows that 15 – 58 missing metadata blocks were found during launches, and these numbers vary with the number of irreclaimable entries in the disk caches under use by running applications. When these missing blocks are found, Paralfetch inserts new log entries for them just before other log entries of associated data blocks.

2) Page fault monitoring. Page cache invalidation is also imperfect because file-backed pages which are dirty, under writeback, or accessed through mmap, are not invalidated. To trace pages which are dirty or under writeback, Paralfetch flushes them out via a sync operation before the disk cache is cleared. However, pages accessed through mmap, such as shared library files, are more challenging. When these are shared with running applications, tracing accuracy is compromised. To address this issue, we arranged for Paralfetch to trace previously untraced blocks accessed through mmap calls by instrumenting the faultaround [49] handler with page fault tracing code. The handler proactively maps 16 boundary-aligned (page-cached) pages around the page-faulted address.

3.2 Prefetch Scheduling

Upon completion of collection of disk I/O requests during an application launch, Paralfetch pre-schedules these requests to speed up the prefetching phase, *merging* and *reordering* requests so as to exploit the internal parallelism of an SSD.

Range merging. Merging small I/O requests into a single large request enhances the throughput of an SSD [12, 27, 32, 43]. Figure 3b shows a range merge in which two requests for blocks with consecutive LBAs that are within a predefined *I/O distance* threshold are combined where the I/O distance is defined as difference in the locations of blocks in the launch sequence. This threshold prevents the merging of far-apart log entries in the launch sequence, as they can hinder timely prefetching of subsequent blocks. Overly-aggressive merging can be bad especially for applications with CPU-bound launches, in which I/O optimization is less influential in meeting prefetching deadlines. Figure 4 shows plots of prefetch time against the I/O distance threshold on SSD, UFS flash, and MicroSD. The performance gain from range merging tails off as the threshold increases mainly because EXT4 tries to locate metadata and data blocks for related files close together in terms of LBA.


LBA-contiguous (I/O distance: sequence number difference between merge candidates = 7 - 3)
Starting block number (LBA)
[1] 0, 3, 1
[2] 1, 5, 8
[3] 0, 16, 1
[4] 0, 20, 1
[5] 1, 13, 2
[6] 2, 22, 3
[7] 0, 17, 1
Sequence number
Inode number
I/O size (blocks)
LBA-contiguous (I/O distance = 5 - 2)
Metadata I/O
File data I/O
Merge: I/O distance between contiguous blocks (3) ≤ I/O distance threshold (3)
[1] 0, 3, 1
[2] 1, 5, 10
[3] 0, 16, 1
[4] 0, 20, 1
[6] 2, 22, 3
[7] 0, 17, 1

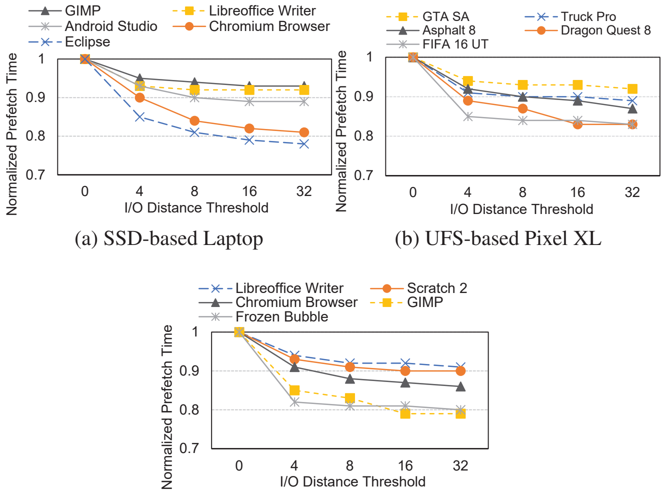
(a) Original prefetch sequence

(b) Prefetch sequence after range merge *with an* I/O distance threshold of 3

Figure 3: Range merge. Merging nearby I/O operations into a single large operation improves throughput while keeping changes to the I/O order within a predefined limit so that the target application and prefetch thread can run concurrently. Range merge combines LBA-contiguous I/O requests of the same type (*e.g.*, metadata or data block) into the preceding one.

Metadata shifting. Every file system has its own particular I/O dependencies for prefetching between metadata and data blocks (and between metadata blocks). In EXT4, a request for a data block can only be issued after the associated metadata block, which contains the LBA of that data block, has been read. The metadata for a data block is often requested just before the corresponding data block.

Thus this dependency tends to limit the number of commands that can be queued, and this in turn limits the effectiveness of command queuing, which yields maximum benefit when there are many commands in the queue which can potentially be executed in parallel [39].



(a) SSD-based Laptop

(b) UFS-based Pixel XL

(c) MicroSD-based Raspberry Pi 3

Figure 4: Normalized prefetching times with varying I/O distance thresholds.

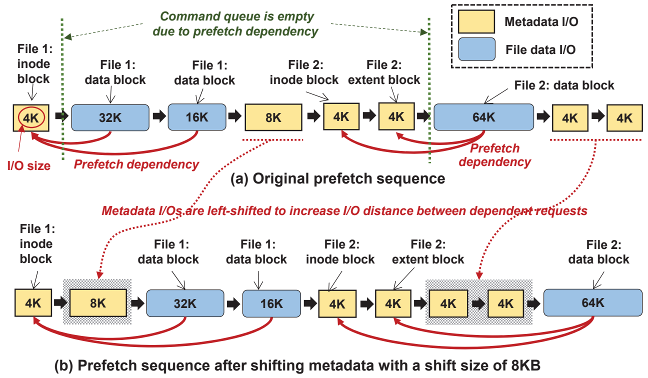


Figure 5: Metadata shifting to boost the outstanding I/O size in the command queue of an SSD controller. An I/O request for data blocks should wait for the associated metadata blocks to be read. By left-shifting I/O requests for metadata, more I/O requests can be issued asynchronously. The shift size controls the extent to which metadata blocks can be left-shifted.

This issue can be addressed by bringing forward requests for metadata blocks. This is facilitated in EXT4, where there are no read dependencies among buffer-cached (metadata) blocks, while I/O requests for page-cached data blocks can only be issued after associated metadata blocks are buffer-cached. Figure 5a shows the processing of an example prefetch thread, in which dependencies on metadata blocks cause the command queue to become empty on two occasions. Figure 5b shows how Paralfetch brings forward metadata block requests in the prefetch thread to increase the interval between requests for dependent blocks. Figure 6a shows that the average prefetching time on a CQ-enabled SSD was reduced by 21.6% through shifting metadata requests forward by 128 KB, when combined with the tracing of missing metadata blocks.

An SSD without CQ support can also benefit from shifted metadata (Figure 6c): requests to the I/O scheduler can be issued in advance, so that the storage driver receives a request earlier from the I/O scheduler queue, rather than later by the application; and an MMC/SD driver (for eMMC flash and SD cards) overlaps flash access for the current I/O request with DMA preparation for the next I/O request. A metadata shift of 4 KB reduced prefetch times by 19.3% on the Raspberry Pi 3 using a MicroSD.

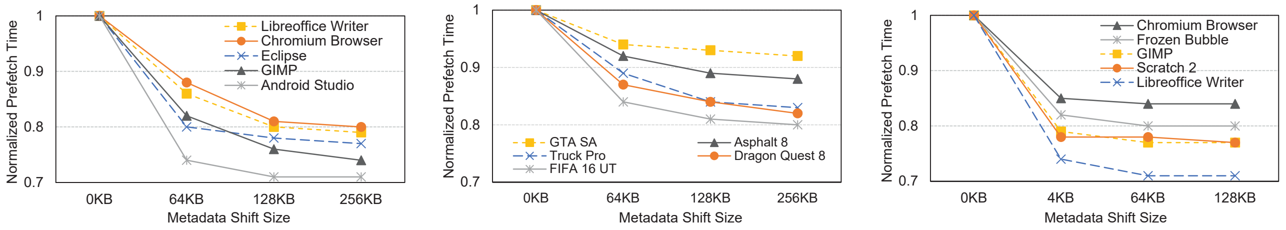
Correctness. The read requests from the prefetch thread go through disk caches, and hence reordering and merging of a launch sequence have no implications on correctness. Even if a prefetch entry is outdated, it only affects the launch performance.

3.3 Parallelized Execution: Overlapping Application Execution with Disk Prefetching

Timely prefetching can better overlap application execution with prefetching. Reordering or merging blocks far apart could improve prefetch throughtput but could also hinder timely prefetching. Experimental results in Figures 7 and 8 substantiate the claim by showing prefetching throughput does not always correspond to launch performance. Paralfetch avoids this pitfall by tailoring metadata shift and range merge dynamically. A challenge is how to find near-optimal threshold values in an automatic manner. To address this, Paralfetch employs *dynamic scheduling* which reschedules prefetch entries with an increased I/O distance threshold and/or metadata shift size when a prefetching bottleneck is detected.

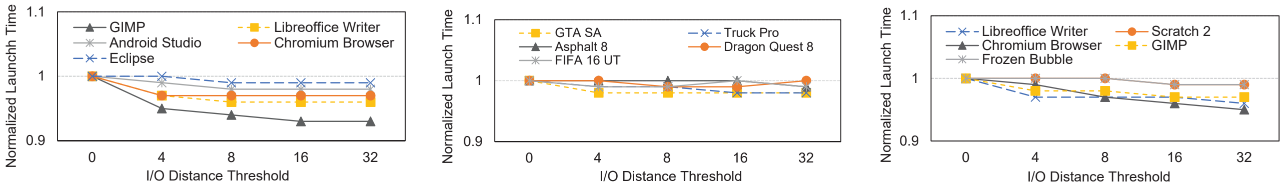
The ability of shifting metadata and merging nearby requests to reduce prefetching time on SSD-based systems is limited by contentions between I/O requests from the prefetch thread and I/O requests which must be issued by the application because they were omitted from the prefetch thread. As shown in Table 1, we found that an average of 2.8% of requested blocks were not traced despite the improved tracing features of Paralfetch. These missing blocks are inevitably requested by the application, which waits until the blocks are loaded from the disk. Contention between the application and the prefetch thread becomes critical when there are too many I/O requests in the I/O scheduler or command queue [13] in an SSD. This can occur when metadata blocks are shifted too far, or when an oversize I/O request is created by range merging with a large threshold. From an experiment with Eclipse, we found that the effect of missing blocks on latency was increased by 3.2× and 8.7× when the largest allowable shifts were 128KB and 256KB, respectively.

To avoid the need to optimize the thresholds for metadata shifting and range merge over a number of trial runs, Paralfetch gradually increases the threshold if prefetching



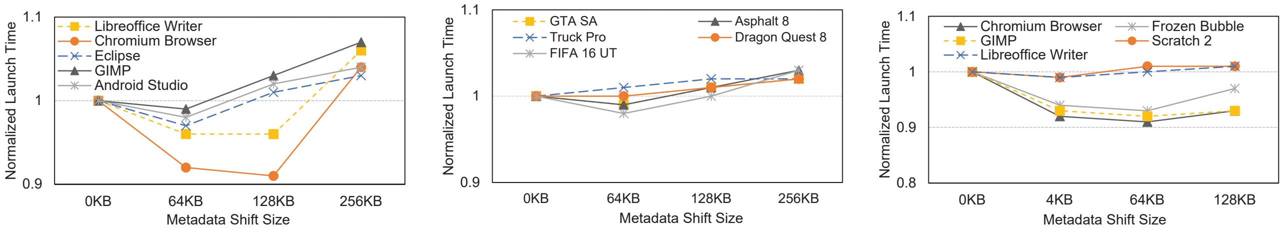
(a) SSD-based Laptop (b) UFS-based Pixel XL (c) MicroSD-based Raspberry Pi 3

Figure 6: Normalized prefetching times for different metadata shift sizes.



(a) SSD-based Laptop (b) UFS-based Pixel XL (c) MicroSD-based Raspberry Pi 3

Figure 7: Normalized launch times with varying I/O distance thresholds.



(a) SSD-based Laptop (b) UFS-based Pixel XL (c) MicroSD-based Raspberry Pi 3

Figure 8: Normalized launch times for different metadata shift sizes.

Table 2: Default configuration for prefetch optimization.

	SSD without CQ feature	SSD with CQ feature
I/O distance threshold for range merging	Starts at 8 and can be increased	8
Metadata shift size (KB) for metadta shifting	4	Starts at 64 and can be increased

is not effective. Next, we describe how to control the extent of dynamic scheduling and how to measure the effectiveness of prefetching.

Optimizing prefetch entries with dynamic scheduling. Initially, Paralfetch uses default thresholds for metadata shifting and range merge shown in Table 2. It subsequently increases the threshold for only one of these methods, depending on the availability of CQ support. The metadata shifting threshold is increased in increments of 16KB and the I/O distance threshold in increments of 4.

The best combination of scheduling methods depends on the type of disk. For example, on a CQ-supported SSD, range merge gains little beyond a threshold of 8, which can, therefore, be used as a default during the learning phase. Similarly, metadata shifting yields little benefit on MicroSD-based devices without CQ support beyond a threshold of 4KB.

Detecting prefetch bottleneck. An application experiences more context switches when it has to wait for the blocks requested by the prefetch thread, implying that the prefetch thread is not prefetching in time. Specifically, the prefetch thread collects the number of context switches made by the launching application during the prefetching period. Paralfetch ends dynamic scheduling if the quantity of context switches is below a user-defined threshold (by default, 5% of the number of prefetch entries). The overall disk read size is checked by Paralfetch in order to remove the results from the warm cache.

4 Implementation of Paralfetch

This section details the workflow of Paralfetch and the interaction among its main components described in Figure 9.

4.1 Launch Phase Management

Native Linux: The next launch type for each application is determined by reading the 9-th byte of the header of its executable and linkable format (ELF) binary file. This byte (referred to as the *phase byte*) is normally used for memory alignment (padding), and has a default value of 0. It is set to PHASE_LEARNING (3) for a learning phase and PHASE_THREADED_PREFETCHING (1) for a prefetching phase. A user can also set this value to PHASE_DISABLE (9) to disable prefetching altogether, for small applications or utilities that frequently experience warm starts. The phase byte is passed to the ELF binary loader (load_elf_binary).

Paralfetch supports two modes for launch phase management. In manual mode, a user explicitly selects applications that will use Paralfetch, by calling pfsetmode, which takes a value for the phase byte and an ELF binary path as arguments. pfsetmode can be also invoked from a desktop icon (*i.e.*, mouse right-click menu). In contrast, Paralfetch is applied to all installed applications in automatic mode, which is similar to the management method used in FAST.

Android: zygote is a process that creates a native Android application in Java by forking and loading the main class of a program [30]. zygote invokes the handleChildProc method to create and run a new Android application. To re-

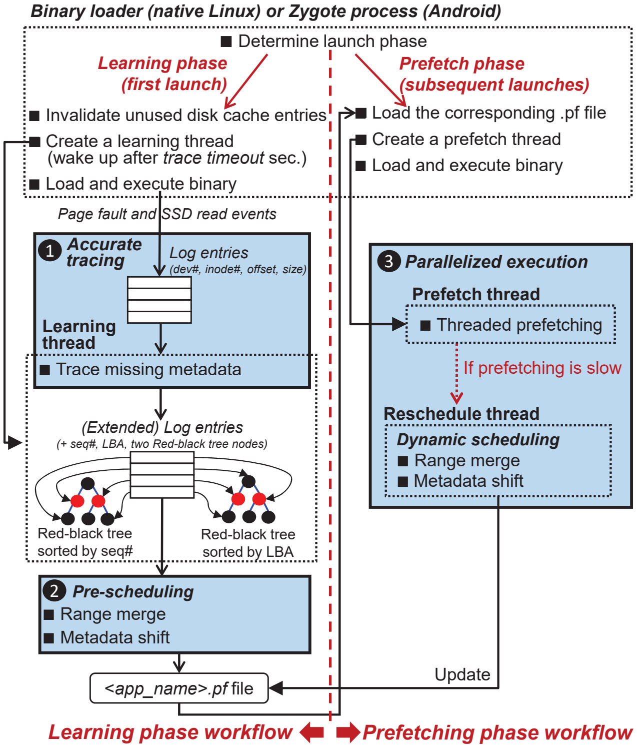


Figure 9: Paralfetch workflow. Boxes with dotted edges denote threads, and boxes with solid edges identify the three major components of Paralfetch. During a learning phase, Paralfetch records an I/O log as a form of log entry. Upon the completion of the launch, collected log entries are passed to missing metadata detector, generating additional log entries for missing metadata. Then, the log entries are passed to pre-scheduling functions as a form of red-black tree. The details of pre-scheduling are described in Algorithm 1 and 2.

duce launch times, zygote preloads classes and resource files used by many applications, quickly creating a process which shares these preloaded classes. Unlike native Linux processes, a native Android process remains in the background even after a user quits the application, and can be resumed by moving the process to the foreground (the resuming procedure). However, when free memory is in short supply, Android wakes up the low memory killer (LMK) to reclaim memory space by removing less important processes completely.

To interface Paralfetch with the Android platform, we created a file named fetch_app using sysfs, which provides a communication interface between the Linux kernel and a user process. On Android, Paralfetch uses automatic launch management mode, in which Paralfetch tailors each launch to the type of application. When the main class name of an application is written to the fetch_app file, Paralfetch determines how to perform the launch phase based on the following rules: if there is no corresponding <class_name>.pf file[2] in the /persist/paralfetch directory, then Paralfetch starts a learning phase for that application; but if the file exists, then Paralfetch performs prefetching. To implement this, we augmented the handleChildProc method to write the main class name of the application being launched to the fetch_app file. Paralfetch does not begin a prefetching for the resuming procedure that does not invoke handleChildProc.

4.2 Learning Phase

I/O logging. To collect blocks required for a launch, Paralfetch first invalidates unused entries in the slab (for file system objects), buffer and page caches, and temporarily disables the inode read-ahead functionality of EXT4 so as to prevent I/O contention resulting from unnecessary inode blocks being read during the prefetching phase. Next, Paralfetch sets a trace timeout, with the default value of 30 seconds, and also sets the trace_flag to true to activate logging. Then, Paralfetch resumes loading and execution of the application. During the execution, the I/O requests for buffer-cached blocks caused by disk cache misses are logged by code introduced into the metadata access function (submit_bh_wbc). Similarly, code introduced into the functions ext4_readpage, ext4_readpages, and filemap_map_pages logs read requests associated with page-cached blocks.

Page fault monitoring. The filemap_map_pages function is called by the OS when a page fault occurs. It pre-faults the 16 boundary-aligned pages which contain the faulting page, provided that these pages are in the page cache [49]. Performing this reduces the overhead of tracing page faults.

Tracing missing metadata blocks. Block tracing ends when the trace times out, and the launch is deemed to be complete when fewer than 10 block read requests occur in a second [25]. We refer to the corresponding block of an application as the *completion block*. To detect missing metadata blocks, we implemented the ext4_fiedep function, a variant of the ext4_fiemap function that must in any case access the metadata blocks associated with file blocks during the mapping of logical-to-physical extents. Unlike the original version that returns file extents for arguments (*i.e.*, a file and query range of the file), the ext4_fiedep function returns a list of associated metadata blocks along with file extents.

As shown in Figure 9, Paralfetch builds two red-black binary search trees for log entries that are used for prefetch scheduling: Paralfetch reads log entries in their access order and inserts each of them to the trees. It invokes the ext4_fiedep function for each log entry for a regular file. If the corresponding metadata blocks are missing from the tree, Paralfetch allocates and inserts new log entries for them right before the entry for the corresponding data blocks.

This operation consumes little CPU time (17 ms for Android Studio) and incurs no disk I/Os because the procedure runs in the warm cache condition (*i.e.*, after the completion of a launch process).

[2] <class_name>.pf file is equivalent to <app_name>.pf in native Linux.

Algorithm 1: Metadata Shift Procedure

```
Input: Log entries sorted by their access order ($rbtree\_seq$),
       Metadata shift size ($ms\_size$)
Result: Metadata-shifted log entries (accessed via $rbtree\_seq$)
1  $log \leftarrow$ first_log_entry($rbtree\_seq$)
2  $out\_meta\_size \leftarrow 0$
3  while $log \neq NULL$ do
4      if $is\_metadata\_log\_entry(log)$ then
5          move_to_MS_queue($log$)
6          $out\_meta\_size \leftarrow out\_meta\_size + log.size$
           /* expired entries ($log.expire$ <= $out\_meta\_size$)
              in wait queue are moved to MS queue             */
7          move_expired_wait_queue_entries_to_MS_queue()
8      else
9          $log.expire = out\_meta\_size + ms\_size$
10         move_to_wait_queue($log$)
11     $log \leftarrow$ next_log_entry_seq($log$)
12 drain_wait_queue_entries_to_ms_queue()
13 rebuild_rbtree_seq_to_correspond_to_ms_queue_order()
```

Algorithm 2: Range Merge Procedure

```
Input: Log entries sorted by their LBA ($rbtree\_lba$) and access
       order ($rbtree\_seq$), IO distance threshold ($dist\_thr$)
Result: Range-merged log entries (accessed via $rbtree\_seq$)
1  $curr \leftarrow$ first_log_entry($rbtree\_lba$)
2  $next \leftarrow$ next_log_entry_lba($curr$)
3  while $next \neq NULL$ do
4      if $curr.inode\_num = next.inode\_num$ &
5      $curr.start\_lba + curr.size = next.start\_lba$ &
6      $next.seq\_num - curr.seq\_num <= dist\_thr$ then
7          $curr.size \leftarrow curr.size + next.size$
8          unlink_log_entry_from_rbtrees_lba_and_seq($next$)
9          remove_log_entry($next$)
10         $next \leftarrow$ next_log_entry_lba($curr$)
11         continue
12     $curr \leftarrow next$
13     $next \leftarrow$ next_log_entry_lba($curr$)
```

Pre-scheduling. Paralfetch schedules the collected log entries. Algorithm 1 describes the procedure of metadata shift: Paralfetch accesses log entries in their access order (lines 1, 3, 11). A log entry for metadata blocks moves right away to the *MS* queue[3] (lines 4–5), while a log entry for data blocks remains in the wait queue until enough subsequent metadata blocks (at least the metadata shift size) are moved to the MS queue (lines 9–10) in order to left-shift metadata I/O requests When enough metadata blocks are left-shifted, the accompanying wait queue log entries are transferred to the MS queue (line 7). Finally, the red-black tree *rbtree_seq* is rebuilt with the metadata-shifted order (line 13) once the remaining log items in the wait queue are transferred to the MS queue (line 12).

To perform range merge (as described in Algorithm 2), Paralfetch accesses log entries in their LBA-sorted order. This makes it easy to detect log entries that have consecutive LBAs (line 5) of the same inode (line 4). Range merge then combines consecutive I/O operations (lines 7–9) that are within a predefined threshold for I/O distance in the launch sequence (line 6).

Different thresholds of metadata shift and range merge are used for SSDs with and without command queuing (CQ). To discover whether an SSD supports CQ, the Paralfetch initialization process, executed by the systemd daemon or a startup script (*e.g.*, rc.local), examines sysfs files. For example, the CQ support for an SATA SSD is determined by the value of /sys/block/*<root device>*/device/queue_depth.

Storing scheduled log entries. Scheduled log entries (*i.e.*, prefetch entries) are stored in the file <app_name>.pf (*e.g.*, eclipse.pf for Eclipse). This file consists of a 24-byte Paralfetch header, followed by prefetch entries. The header contains the version number, the inode number of the executable file, the metadata for dynamic scheduling, the number of obsolete entries, and the number of prefetch entries. Each prefetch entry contains the device number, the inode number, its offset and size. The inode number for a metadata block is set to 0. The size of each prefetch entry is 20(24) bytes on a 32(64)-bit system.

4.3 Prefetching Phase

During the prefetching phase, Paralfetch creates the prefetch thread, following the sequence stored in the <app_name>.pf file.

For EXT4 file system, Paralfetch uses the __breadahead function to prefetch metadata blocks, and the force_page_cache_readahead function to prefetch data blocks for regular files. While these functions try to perform block caching asynchronously (or in a non-blocking manner), data blocks can be prefetched asynchronously only when the associated metadata blocks are ready. Paralfetch uses explicit I/O plugging [3] to merge contiguous metadata (bio) requests into a single request, which is then delivered to the dispatch queue of device drivers. This reduces the amount of computation required for dispatching and completing I/O requests.

Changing from prefetching back to the learning phase. The set of blocks required for the first launch of some applications is significantly different from that required for subsequent launches. For example, Eclipse and GIMP only configure their environments on their first launch: Paralfetch detects this behavior by counting I/O requests issued by an application during its launch, which is easily done by counting synchronous readahead requests [38] in the Linux readahead framework [33]. If the count is greater than 10% of the total number of prefetch entries, Paralfetch returns to the learning phase.

5 Evaluation

5.1 Methodology

Launch time measurement. Like [24], we measure the launch time of an application between two events: in the

[3] The MS queue stores the metadata-shifted order of log entries.

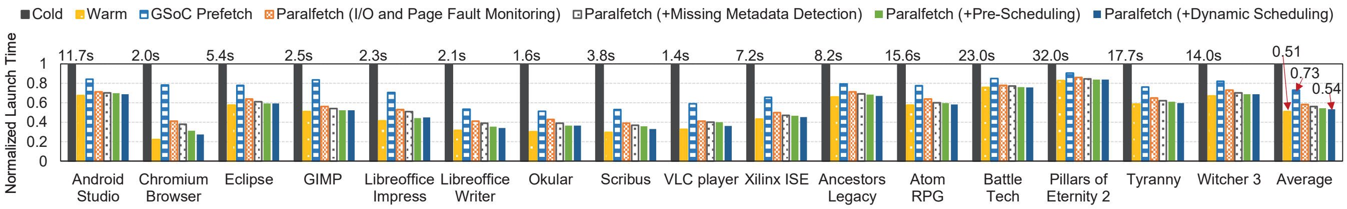


Figure 10: Launch times on a laptop equipped with a QLC SSD, normalized to cold start times. Optimizations for Paralfetch are incrementally applied.

case of Linux, the launch is deemed to start when the load_elf_binary function is called, and to finish when the completion block request has itself completed. To identify the latter event, we remove the completion block request from the prefetch file, allowing it to be issued by the application. After a warm start, we call posix_fadvise with the argument POSIX_FADV_DONTNEED to evict the completion block request from the page cache.

Comparisons with other prefetchers. We ported the GSoC Prefetcher to the Linux kernel 5.4.51 and set its trace timeout to the value used by Paralfetch. We temporarily modified Paralfetch to bring its operation in line with three key features of the GSoC Prefetcher: 1) the way in which it traces referenced file pages during an application launch, 2) its method of pre-scheduling disk I/O using inode numbers and in-file offsets as sort key, and 3) the way in which it holds an application until prefetching is completed, rather than allowing the application and the perfetcher thread to compete.

FAST only supports EXT3 file system, so we temporarily modified Paralfetch's function for detecting missing metadata to support EXT3. We could only compare FAST with Paralfetch on a PC because the Android and Raspbian OS do not support EXT3 file system.

5.2 On a PC

We conducted experiments on a laptop PC equipped with an Intel Core i5-8265 CPU and 16 GB of RAM, running Linux kernel 5.4.51. This PC has a 1 TB Samsung 860 QVO QLC SSD, which uses native command queuing. We tested Paralfetch, GSoC Prefetch and FAST on 16 applications, 6 of which were games. The 10 non-game applications were Android Studio, Chromium Browser, Eclipse, GIMP, LibreOffice Impress, LibreOffice Writer, Okular, Scribus, VLC player, and Xilinx ISE; and the 6 games were Ancestors Legacy, Atom RPG, Battle Tech, Pillars of Eternity 2, Tyranny, Witcher 3.

QLC SSDs typically employ a small pseudo-SLC (single-level cell) cache. To reduce the effects of this cache, we conducted evaluation after installing all benchmark apps.

Comparison with the GSoC prefetcher. Figure 10 shows Paralfetch to reduce the average launch time of these 16 applications by 44.2% with pre-scheduling alone. After four launches of each application, a 1.8% more reduction was achieved on average by using dynamic scheduling to increase prefetch throughput.

It should be noted that the naïve use of excessive metadata shift (of 256KB) led to a 3.8% increase in average launch time: as previously shown in Table 1, Paralfetch fails to trace a few launch blocks. A launching application should wait for these missing blocks to be read while a large number of outstanding I/O requests due to excessive metadata shift increase the waiting time.

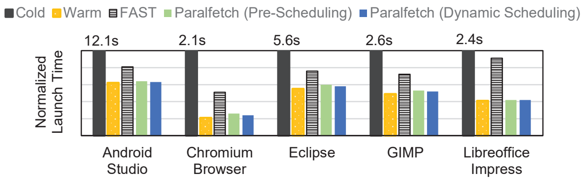


Figure 11: Comparison of Paralfetch and FAST launch times on a laptop PC, normalized to cold start times. Tracing of each application is performed when LibreOffice Writer is running in the background. The results show that running applications can significantly degrade tracing accuracy of FAST and its performance benefit.

Comparison with FAST. FAST is the closest to ours in that its target media is SSDs. In §3.1 we described how disk cache clearing affects tracing accuracy. The most serious drawback of FAST seems to be that the accuracy of its tracing depends greatly on the other applications that are running, because files accessed by these applications through mmap are not traced. Also, metadata used by the applications are not traced. We believe that this issue is frequently occurred in common scenarios. Figure 11 shows the significance of this issue. Conversely, the page fault monitoring and detecting missing metadata used by Paralfetch leads to launch times similar to that of a warm start.

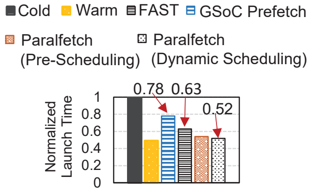


Figure 12: Average launch time for 16 apps on a laptop equipped with a QLC SSD, normalized to cold start times.

Although tracing under a system-cold state favors FAST, the launch times averaged across all 16 applications were 11% less with Paralfetch than with FAST as shown in Figure 12. The relatively poor performance of FAST can be attributed to its reliance on system calls, which limits both the accuracy of tracing and its scheduling options, in particular its use

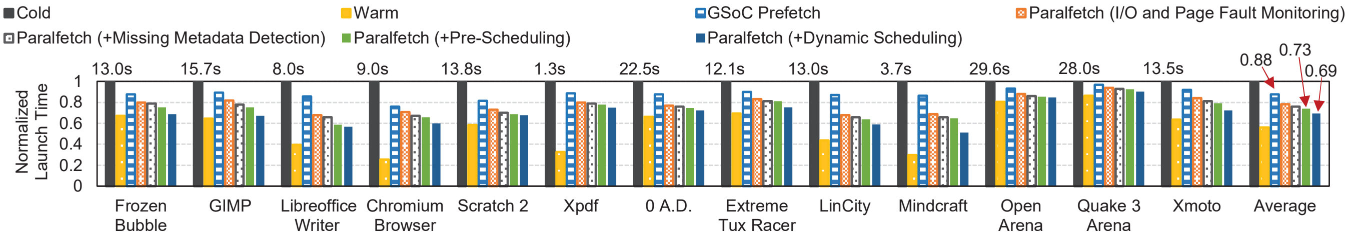


Figure 13: Launch times on a Raspberry Pi 3, normalized to cold start times. Optimizations for `Paralfetch` are incrementally applied.

of synchronous I/O for prefetching metadata blocks makes it difficult to exploit parallelism.

5.3 Raspberry Pi 3

Our second evaluation of `Paralfetch` was conducted on a Raspberry Pi 3 running the Raspbian OS (Linux kernel 4.9.56) with a Samsung 16 GB MicroSD (class 10). This flash storage does not support CQ (although more recent A2-class MicroSD has both CQ and an SLC cache).

We used 13 applications, 8 of which were games: Frozen Bubble, GIMP, LibreOffice Writer, Chromium browser, Scratch 2, Xpdf, 0 A.D., Extreme Tux Racer, LinCity, Mindcraft, Open Arena, Quake 3 Arena, and Xmoto. The launch times in Figure 13 show that frequent flash accesses contribute about 45% of the delay in application launches. This provides a considerable opportunity for I/O scheduling. After four launches with dynamic scheduling, launch times are further reduced by an average of 4.8% compared to `Paralfetch` with pre-scheduling only. We attribute this reduction to: 1) an application launch on a Raspberry Pi 3 board is a disk-bound process, and 2) the throughput of a MicroSD is usually improved by merging I/O operations: for example, the bandwidth of random reads of 128KB on the MicroSD we used is 28.6 MB/sec, which is 6.7× higher than that of 4KB (only 4.3 MB/sec). Chromium Browser and Xpdf application launch times are more heavily influenced by disk performance than by CPU performance. Due to the significant limitations of timely prefetching with prefetch scheduling, it is difficult to achieve warm start launch performance, especially for SSDs without command queuing.

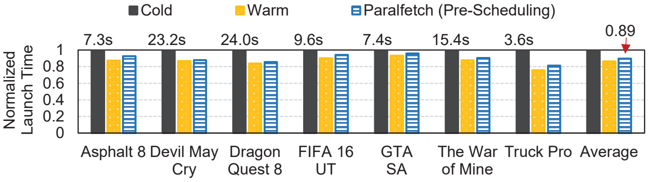


Figure 14: Launch times on an Android smartphone (Google Pixel XL), normalized to cold start times.

5.4 Google Pixel (Android)

`Paralfetch` can be easily ported to Linux variants, such as Android. Android has its own launch mechanism, and hence we needed to modify 180 lines of the Android source code to accommodate `Paralfetch`.

To test `Paralfetch` on Android, we used a new set of seven games: Asphalt 8, Devil May Cry, Dragon Quest 8, FIFA 16 UT, GTA SA, The War of Mine, and Truck Pro. We measured the launch times for these games on a Google Pixel XL smartphone with UFS flash (which supports CQ) running Android 8.0 (Oreo) with the Linux kernel 3.18.52. As shown in Figure 14, the pre-scheduling performed by `Paralfetch` reduced launch times by 11% on average, which equates to as much as 3.5 seconds for Dragon Quest 8. However, dynamic scheduling offers little benefit because 1) application launches are CPU-bound (86% on average in our benchmarks) rather than disk-bound, and 2) launches encounter little dependencies between metadata and data blocks. Another distinct characteristic of an Android app launch is that a number of `write` and `fdatasync` syscalls are issued by SQLite during the launch, making a gap between the times for a warm start and a cold start with `Paralfetch`.

5.5 Overhead

We measure `Paralfetch`'s overheads on a laptop PC from 4 aspects: tracing, pre-scheduling, prefetching and storage.

Tracing overhead. The I/O-based tracing used by `Paralfetch` has a low instrumentation overhead, and in most cases log entries are relatively short (*e.g.*, less than 3000 entries). Android Studio is an exception, as it creates lots of log entries. Nevertheless, the difference in cold start launch time with and without `Paralfetch` was only 136ms. Disk cache invalidation can produce some latency, but this does not affect the working set of pages. Thus, it should not affect the users. In any case, the cache is only invalidated during the learning phase.

Pre-scheduling overhead. In our experiments, the time required by the background jobs which perform pre-scheduling, including missing metadata detection, metadata shift, and range merge, varied between 42ms for VLC Player and 153ms for Android Studio, whereas `FAST` took 21 seconds to generate the prefetch program for Android Studio. When there is an idle CPU core, pre-scheduling delays can be hidden from users because `Paralfetch` creates a dedicated thread for that.

Prefetching overhead. Paralfetch employs threaded prefetching, imposing extra overhead from management perspective. However, we observed that threaded prefetching can reduce CPU usage for an application launch in the cold start. As shown in Figure 2, a synchronous I/O incurs two context switches. On the other hand, the asynchronous I/O requests issued by the prefetch thread significantly reduce the overall number of context switches. In our sampling-based CPU utilization measurement [22], we found that the number of context switches during a launch of Android Studio with Paralfetch was reduced from 9,902 to 1,035, resulting in a 3.2% reduction in CPU usage.

In the warm start where prefetching is unnecessary, Paralfetch still runs the prefetch thread, but this only incurs a delay of hundreds of microseconds if an available CPU core exists. Even if there was no available CPU core, where prefetching overhead could not be hidden, Paralfetch extended Android Studio launch by only 2.8ms for (Eclipse by 3.1ms, which was the worst case).

Storage overhead. Paralfetch used 672 KB of SSD to store the <app_name>.pf files for the 16 applications, whereas FAST required 8.2 MB.

6 Future research direction

Non-intrusive tracing. Paralfetch instruments some kernel functions to trace disk accesses. The (low) instrumentation overhead can be effectively removed by employing dynamic instrumentation tools, such as SystemTap [55] and eBPF [56].

Sophisticated prefetch scheduling. Paralfetch applies metadata shifting and range merging to the entire launch sequence, leaving room for further improvement: by applying prefetch scheduling only to prefetch-bottlenecked regions of the launch sequence, Paralfetch can avoid unnecessary I/O contention between the prefetch thread and the launching application, achieving a better launch performance.

Prefetch scheduling considering internal behaviors of disks. If Paralfetch schedules prefetch entries considering internal behaviors and performance of storage devices, it can schedule them better at the pre-scheduling stage, thus reducing the need for rescheduling with dynamic scheduling.

7 Additional Related Work

Previous application prefetchers are discussed in §2. We now summarize various other approaches to reducing application launch times, which are orthogonal or complementary to Paralfetch.

Predictive disk prefetchers, such as Preload [14] and Windows Superfetch [37], analyze the pattern and frequency of application usage, predict the applications that are likely to be loaded soon, and then preload them. Falcon [42] is a predictive prefetcher that considers mobile context such as location and battery state. Falcon launches an application in advance rather than merely prefetching launch-related blocks. Obviously, the merit of this strategy depends heavily on the accuracy of the prefetcher's predictions [34].

General-purpose disk prefetcher. It has been demonstrated that general-purpose prefetching [11, 28] can also be beneficial in reducing application launch times. However, it can limit the accuracy of tracing launch-related blocks because block-level I/O patterns depend greatly on the contents of disk caches.

A block I/O cache provides another way of reducing latency. Intel Turbo Memory [31], Intel Smart Response Technology [51], and AMD StoreMI [52] store delay-sensitive data in a relatively fast SSD and other data in a larger region of slower storage. A similar behavior is provided by software caching methods, which operate in the device mapping layer [1] and the block layer [2].

I/O scheduling can reduce I/O contention between a launch process and background processes. Several schemes have been proposed: FastTrack [16] prioritizes I/O requests generated by the foreground application, and the BFQ I/O scheduler [10] gives new processes extra I/O bandwidth. Boosting the priority of an I/O request, which is issued asynchronously but results in blocking the issuing process, can also expedite a launch [21].

Memory management can also reduce latency. Re-assigning pages from background apps to foreground apps can improve user experience of mobile operating systems [44]. Similarly, pre-swapping of unused memory can reduce delays by avoiding page reclamation latencies [45]. These schemes can reduce app launch times by timely provision of memory when it is under pressure.

8 Conclusion

We have presented Paralfetch, which achieves launch performance close to the warm start through more accurate tracing, pre-scheduling for fast I/O reads, and prefetch thread overlapping. Paralfetch incurs negligible overhead in terms of CPU, memory, and storage. We have also shown Paralfetch to significantly outperform existing prefetchers on various personal computing/communication devices running Linux.

Acknowledgments

We would like to thank our shepherd Nitin Agrawal and the anonymous reviewers for their valuable feedback and suggestions. This work was partly supported by the IITP grant (No. RS-2022-00155885, Artificial Intelligence Convergence Innovation Human Resources Development (Hanyang University ERICA)) and the National Research Foundation (NRF) of Korea grant funded by the Korean government (MSIT) (No. NRF-2022R1F1A1074505). K. Kang and D. Lee are the corresponding authors.

References

[1] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao. Cloudcache: On-demand flash cache management for cloud computing. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, pages 355–369, 2016.

[2] L. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving virtualized storage performance with Sky. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 112–128, 2017.

[3] J. Axboe. Explicit block device plugging. https://lwn.net/Articles/438256/, 2011.

[4] Y. Takai, M. Fukuchi, R. Kinoshita, C. Matsui, and K. Takeuchi. Analysis on heterogeneous SSD configuration with quadruple-level cell (QLC) NAND flash memory. In *Proceedings of the 11th IEEE International Memory Workshop (IMW)*, pages 169–172, 2019.

[5] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. LipTak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for self-optimizing storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, pages 183–196, 2009.

[6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 188–197, 1995.

[7] F. Chang, and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 1999.

[8] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 266–277, 2011.

[9] L. Colitti. Analyzing and improving GNOME startup time. In *Proceedings of the 5th System Administration and Network Engineering Conference (SANE)*, pages 1–11, 2006.

[10] J. Corbet. The BFQ I/O scheduler. https://lwn.net/Articles/601799/, 2014.

[11] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. Diskseen: Exploiting disk layout and access history to enhance I/O prefetch. In *Proceedings of the 2007 USENIX Annual Technical Conference (ATC)*, pages 261–274, 2007.

[12] C. Dirik, and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 279–289, 2009.

[13] A. Eisenman, I. Abdelrahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*,pages 42:1–13, 2018.

[14] B. Esfahbod. Preload–An adaptive prefetching daemon. Master's thesis, Graduate Department of Computer Science, University of Toronto, Canada, 2006.

[15] W. Fengguang, X. Hongsheng, and X. Chenfeng. On the design of a new Linux readahead framework. *ACM SIGOPS Operating Systems Review*, 42(5):75–84, 2008.

[16] S. S. Hahn, S. Lee, I. Yee, D. Ryu, and J. Kim. FastTrack: Foreground app-aware I/O management for improving user experience of Android smartphones. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 15–28, 2018.

[17] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 71–83, 2011.

[18] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson. Informed mobile prefetching. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MOBISYS)*, pages 155–168, 2012.

[19] W. W. Hsu, A. J. Smith, and H. C. Young. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems*, 23(4):424–473, 2005.

[20] B. Hubert. On faster application startup times: Cache stuffing, seek profiling, adaptive preloading. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 245–248, 2005.

[21] D. Jeong, Y. Lee, and J.-S. Kim. Boosting quasi-asynchronous I/O for better responsiveness in mobile devices. In *Proceedings of The 13th USENIX Conference on File and Storage Technologies (FAST)* pages 191–202, 2015.

[22] Y. Joo, Y. Cho, K. Lee, and N. Chang. Improving application launch times with hybrid disks. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* pages 373–382, 2009.

[23] Y. Joo, J. Ryu, S. Park, H. Shin, and K. G. Shin. Rapid prototyping and evaluation of intelligence functions of active storage devices. *IEEE Transactions on Computers*, 63(9):2356–2368, 2014.

[24] Y. Joo, J. Ryu, S. Park, and K. G. Shin. FAST: Quick application launch on solid-state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pages 259–272, 2011.

[25] Y. Joo, J. Ryu, S. Park, and K. G. Shin. Improving application launch performance on SSDs. *Journal of Computer Science and Technology*, 27(4):727–743, 2012.

[26] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 209–222, 2012.

[27] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk schedulers for solid state drivers. In *Proceedings of the 9th ACM/IEEE International Conference on Embedded software (EMSOFT)*, pages 295–304, 2009.

[28] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)* pages 173–186, 2004.

[29] K. Lichota. Prefetch: Linux solution for prefetching necessary data during application and system startup. http://code.google.com/p/prefetch/, 2007.

[30] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in data-parallel systems. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 383–400, 2016.

[31] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel®Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. *ACM Transactions on Storage*, 4(2):1–24, 2008.

[32] D. T. Nguyen, Improving smartphone responsiveness through I/O optimizations. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication (UBICOMP Adjunct)*, pages 337–342, 2014.

[33] R. Pai, B. Pulavarty, and M. Cao, Linux 2.6 performance improvement through readahead optimization. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 105–116, 2004.

[34] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. M. Marlin. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UBICOMP)*, pages 275–284, 2013.

[35] K. Kim, E. Lee, and T. Kim, HMB-SSD: Framework for efficient exploiting of the host memory buffer in the NVMe SSD. *IEEE Access*, vol. 7, pp. 150403-150411, 2019.

[36] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 79–95, 1995.

[37] M. Russinovich, D. Solomon, and A. Lonescu. Windows Internals, Part 2, 6th ed. Microsoft Press, pages 324-350, 2012.

[38] W. Mauerer. Professional Linux Kernel Architecture. Wrox Press, pages 970–974, 2008.

[39] J. Ryu, Y. Joo, S. Park, H. Shin, and K. G. Shin. Exploiting SSD parallelism to accelerate application launch on SSDs. *IET Electronics Letters*, 47(5):313–315, 2011.

[40] S. Vandebogart, C. Frost, and E. Kohler. Reducing seek overhead with application-directed prefetching. In *Proceedings of the 2009 USENIX Annual Technical Conference (ATC)*, pages 299–312, 2009.

[41] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho. Barrier-enabled IO stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 211–226, 2018.

[42] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fastapp launching for mobile devices using predictive user context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MOBISYS)*, pages 113–126, 2012.

[43] S. S. Hahn, S. Lee, C. Ji, L.-P. Chang, I. Yee, L. Shi, C. J. Xue, and J. Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 759–771, 2017.

[44] N. Lebeck, A. Krishnamurthy, H. M. Levy, and I. Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *Proceedings of*

the 2020 USENIX Annual Technical Conference (ATC), pages 873–887, 2020.

[45] Y. Liang, J. Li, R. Ausavarungnirun, R. Pan, L. Shi, T.-W. Kuo, and C. J. Xue. Acclaim: Adaptive memory reclaim to improve user experience in Android systems. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, pages 897–910, 2020.

[46] BodNara. ADATA Ultimate SU630 960GB. `https://www.bodnara.co.kr/bbs/article.html?num=154114`, 2019.

[47] T. Schiesser. Storage Game Loading Test: PCIe 4.0 SSD vs. PCIe 3.0 vs. SATA vs. HDD. `https://www.techspot.com/review/2116-storage-speed-game-loading`, 2019.

[48] T. Thomas. Samsung's 860 QVO 1-TB SSD reviewed. `https://techreport.com/review/34281/samsungs-860-qvo-1-tb-ssd-reviewed`, 2018.

[49] K. A. Shutemov. mm: map few pages around fault address if they are in page cache. `https://lwn.net/Articles/588802`, 2014.

[50] R. Nelson. The size of Iphone's top apps has increased by 1,000% in four years. `https://sensortower.com/blog/ios-app-size-growth`, 2017.

[51] Intel® Smart Response Technology: Technology Brief. `https://www.intel.com/content/www/us/en/architecture-and-technology/smart-response-technology-brief.html`, 2014.

[52] AMD StoreMI Technology. `https://www.amd.com/en/technologies/store-mi`, 2021.

[53] S. Sivaram, and C. Bergey, Zoned storage for the zettabyte age, `https://www.flashmemorysummit.com/Proceedings2019/08-06-Tuesday/20190806_Keynote2_WesternDigital_Sivaram_Bergey.pdf`, 2019.

[54] James Larus. Spending Moore's dividend. *Communications of the ACM*, 2009.

[55] SystemTap. `https://sourceware.org/systemtap`, 2022.

[56] eBPF. `https://ebpf.io`, 2022.

[57] Install Ubuntu on a Raspberry Pi. `https://ubuntu.com/download/raspberry-pi`, 2022.

Integrated Host-SSD Mapping Table Management for Improving User Experience of Smartphones

Yoona Kim
Seoul National University

Inhyuk Choi
Seoul National University

Juhyung Park
DGIST

Jaeheon Lee
DGIST

Sungjin Lee
DGIST

Jihong Kim
Seoul National University

Abstract

Host Performance Booster (HPB) was proposed to improve the performance of high-capacity mobile flash storage systems by utilizing unused host DRAM memory. In this paper, we investigate how HPB should be managed so that the user experience of smartphones can be enhanced from HPB-enabled high-performance mobile storage systems. From our empirical study on Android environments, we identified two requirements for an efficient HPB management scheme in smartphones. First, HPB should be managed in a foreground app-centric manner so that the user-perceived latency can be greatly reduced. Second, the capacity of the HPB memory should be dynamically adjusted so as not to degrade the user experience of the foreground app. As an efficient HPB management solution that meets the identified requirements, we propose an integrated host-SSD mapping table management scheme, HPBvalve, for smartphones. HPBvalve prioritizes the foreground app in managing mapping table entries in the HPB memory. HPBvalve dynamically resizes the overall capacity of the HPB memory depending on the memory pressure status of the smartphone. Our experimental results using the prototype implementation demonstrate that HPBvalve improves UX-critical app launching time by up to 43% (250 ms) over the existing HPB management scheme, without negatively affecting memory pressure. Meanwhile, the L2P mapping misses are alleviated by up to 78%.

1 Introduction

User experience (UX) design is one of the topmost tasks in designing modern smartphones. In order to create a high-quality UX from a smartphone, it is essential for the smartphone to react promptly to user inputs without a noticeable delay. For example, when an application (app) is launched, if there exists a considerable user-perceived delay, the quality of UX would be significantly degraded. Since user-perceived delays play a key role in realizing high-quality UX, many researchers have investigated various system resource management schemes so that user-perceived delays can be minimized for the user-facing foreground (FG) apps [1–4].

Although existing techniques have explored the most plausible sources that influence user-perceived delays, a storage system has not been actively investigated from the perspective of user-perceived delays. As the capacity of a mobile storage system quickly increases (*e.g.,* a 1-TB Universal Flash Storage (UFS) device [5]), the read latency of the mobile storage system is emerging as a key factor that can negatively affect user-perceived delays [1, 4, 6, 7]. Since the overall quality of UX is determined by how promptly a smartphone responds to a user's input, storage responsiveness has a significant correlation with improved user responsiveness. There are two main reasons why the read latency of the mobile storage system has a high impact on the UX quality. First, the read latency of a mobile storage system accounts for the largest portion of the total latency of a host request in modern smartphones [1]. For example, when an app is launched on an Android smartphone, approximately more than half of the total app launching time is taken by the storage read time [1, 6].

Second, the read latency of a mobile storage system varies significantly because of the limited SRAM capacity in the mobile storage system. Since SRAM in the mobile storage system is used for implementing a logical-to-physical (L2P) mapping table, which is an essential component of a flash-based storage system, the performance of the mobile storage system is highly dependent on the capacity of SRAM. Unfortunately, the capacity of SRAM is quite limited for large-capacity mobile storage systems. Under this design constraint, an L2P mapping table is commonly managed by an on-demand scheme (*e.g.,* DFTL [8]) that only loads a small portion of the entire L2P mapping entries in (fast) SRAM while the complete L2P mapping table is stored in (slow) flash memory. When most host read requests cannot find their L2P mapping entries from SRAM, their read latency can be significantly longer, thus causing a large increase in user-perceived delays. For example, in our exploratory evaluation, we observed that the app launching time increases by up to 50% when the SRAM only contains a portion of the L2P mapping entries as opposed to when it contains all entries.

To overcome the performance problem from the limited

SRAM capacity within a mobile storage system, Host Performance Booster (HPB) [9, 10] was introduced to store L2P mapping entries in the host memory. It was first shipped in production by Google's Pixel 3 in 2018 with Linux kernel v4.9.96 [11] shortly after its introduction. By exploiting the host memory as a (fast) L2P mapping cache in addition to the SRAM of a mobile storage system, HPB can improve I/O performance by reducing costly SRAM L2P cache misses that require slow flash read accesses. Although several researchers have successfully shown that exploiting the host memory is an effective approach for improving I/O performance [9, 12–14], few work has treated the problem of utilizing the host memory for high-performance I/O *from the UX perspective* in a holistic fashion. The main goal of our work is to comprehensively investigate how HPB should be managed so that the UX of smartphones can be enhanced from HPB-enabled high-performance mobile storage systems.

In order to understand how HPB should be managed in a UX-aware fashion, we evaluate how various UX-related performance metrics are affected by different HPB settings on Android environments. To this end, we measure performance metrics that are relevant to UX quality, such as the app launching, switching, and loading times. We measure this systematically using repeatable and reproducible benchmarks to enable accurate and reliable UX-quality evaluation, eliminating the possibility of human errors. From our empirical study, we identify two key requirements for an efficient HPB management scheme in smartphones. First, the existing HPB management scheme [15] is FG app-oblivious in that the app status is not actively considered in managing the HPB memory. For example, HPB only focuses on caching L2P entries with high reference counts without considering its impact on UX. In order to improve UX from a smartphone, HPB memory should be managed in an FG app-centric manner so that the user-perceived delay of a user-facing FG app can be effectively minimized. Second, the capacity of the HPB memory reserved from the host memory should be dynamically adjusted during run time so that no apps suffer extra memory pressure from the HPB-allocated DRAM. For example, when a large amount of memory is statically reserved for HPB, the low memory killer daemon (LMKD) [16] is triggered more frequently to relieve increased memory pressure, significantly degrading the UX. Allocating small-size memory to avoid such cases is not ideal either as it negates the potential performance improvement from deploying HPB.

As an efficient HPB management solution that meets two key requirements, we propose HPBvalve (Hvalve in short), an integrated host-SSD mapping management scheme for HPB-enabled smartphones. Unlike the existing HPB management scheme [15], Hvalve prioritizes FG app in caching entries to HPB by integrating app status (FG or BG) for every submitted I/O. For further UX improvement, Hvalve detects every app launch event which is one of the most important activities of smartphones that highly impacts UX. Then, Hvalve utilizes the profiled launch-time-referenced L2P list ahead of time to reduce user-perceived delays of an app launch. Additionally, Hvalve adjusts the maximum capacity of the reserved HPB memory according to the current memory pressure status of a smartphone. When memory pressure is monitored, Hvalve selectively returns HPB memory to apps. Through dynamic HPB memory size adjustment, Hvalve can utilize the unused host memory efficiently while preventing inadvertent UX regression from using HPB.

In order to validate the effectiveness of the proposed Hvalve, we develop a prototype Hvalve that supports the internal operational logic of Hvalve on a hardware development kit (HDK) based on the Snapdragon 888 SoC [17] (see Section 6.1 for details). Our experimental results show that Hvalve can effectively manage the HPB memory, reducing the user-perceived delays of an FG app by up to 43% over the existing scheme without increasing the overall memory pressure.

The remainder of this paper is organized as follows. We first review how HPB-enabled smartphones work in Section 2 and review related work in Section 3. In Section 4, we present the key design requirements of a UX-aware HPB management scheme based on our empirical observations. In Section 5, we describe the design and implementation of Hvalve. The experimental results are reported in Section 6. Finally, we conclude with a summary in Section 7.

2 Background

In this section, we briefly explain the basics of L2P mapping structures and policies of the conventional and the HPB-enabled storage systems.

2.1 Controller-side L2P Mapping Structure

The latency of I/O requests submitted to a UFS device varies greatly depending on how the underlying L2P mapping scheme works. As the capacity of UFS devices increases, its L2P mapping table size also increases accordingly. Keeping such a large mapping table in a small SRAM inside the UFS device is technically impossible. Therefore, the UFS device employs an on-demand cache scheme that stores the entire L2P mapping table in flash, caching popular mapping entries in SRAM. On a cache hit, the UFS device provides excellent performance. On a cache miss, however, it suffers from long I/O latency because the missing L2P entry must be fetched from the flash first before serving an I/O request.

Fig. 1 illustrates how the UFS deals with I/O requests from the host in detail. When a read request is received (❶), the flash translation layer (FTL), which is responsible for translating a logical page address (LPA - file-system managed address) to a physical page address (PPA - storage device managed address), looks up cached L2P entries in SRAM (❷). If the desired L2P entry is found in cache, the FTL reads the requested data from the flash by consulting the translation information and returns it to the host. To keep track of hot entries, the FTL internally maintains a pseudo-LRU list for

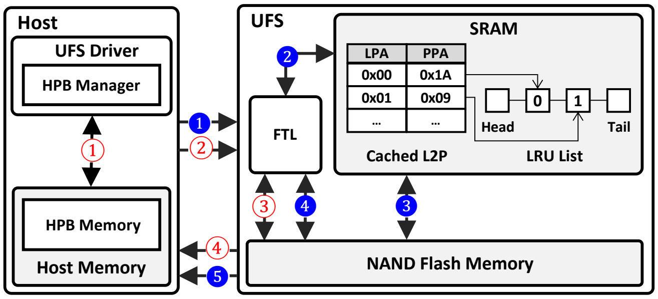


Figure 1: A read operation path.

L2P entries. The hit L2P entry moves to the head of the list. If the FTL fails to find the matched entry, it has to choose and evict a victim entry to make room in the SRAM. The entry at the tail of the list is evicted and the FTL reads in the wanted L2P entry from the flash to the SRAM (❸). Finally, the FTL reads the data from the flash (❹), and delivers the read data to the host (❺).

The on-demand cache scheme performs well when the size of the SRAM cache is large enough to accommodate most of the hot L2P entries. However, the capacity of a UFS device scales much faster than that of SRAM, which makes it difficult to cache sufficient hot entries in the SRAM. For example, the latest UFS device offers up to 1 TB [5] capacity, but its internal SRAM capacity is known to be only several hundred kilobytes [18]. Considering that the mapping table size is estimated as 0.1% of the UFS capacity, only the top 0.0005% of the table entries can be cached in the SRAM when its size is 512 KB, which is too small to keep hot entries.

2.2 Host-side L2P Mapping Structure

The constrained capacity of SRAM results in inconsistent I/O latency, which degrades UX. To overcome this problem, Jeong *et al.* [9] have introduced a Host Performance Booster (HPB) which extends a storage mapping space by exploiting the host memory. The HPB borrows a specific portion of the host memory and then keeps popular L2P entries to improve a mapping cache hit ratio.

Fig. 1 illustrates how the UFS device handles a read request when the HPB is enabled. The HPB manager is implemented in the UFS device driver of the Android kernel and manages the host memory space dedicated to caching L2P entries. Before sending a read request to the UFS, the HPB manager first searches for its L2P mapping entry in the host memory using the logical block address (LBA) of the request. If the desired L2P entry is found, the corresponding PPA is piggybacked on the read request (①), which is then submitted to the UFS (②). Upon the receipt of the request, the FTL in the UFS first verifies the integrity of the given PPA [9] and then directly issues a page read request to fetch the data of the designated PPA (③). It is unnecessary to look up the device-side mapping table. Finally, the FTL delivers the data to the host (④).

The HPB manager is responsible for selecting which L2P entries to fetch from UFS and keep in the HPB-designated host memory, based on its predefined conditions. A single HPB entry is 4 KB in size and stores 512 L2P entries. The HPB manager retrieves 512 L2P entries from UFS through one fetch command. The fetch command involves a normal 4-KB block read request to UFS, so the latency of a single fetch command is comparable to regular read latency. Whenever PPAs of L2P entries are changed due to internal operations such as a garbage collection on the UFS device side, the HPB manager is informed of the invalidated PPAs.

Using HPB, the overall I/O performance can be greatly improved by minimizing L2P misses. However, this benefit comes at the cost of reduced working memory space for apps. When integrating the HPB to the system, the following two technical issues should be carefully considered. The first is to properly decide the size of the HPB-designated memory. If the HPB size is too small, I/O performance gains by the HPB would be marginal. Conversely, if it is overly provisioned, the performance of apps would drop significantly as the HPB steals too much system memory which was to be used for apps. The second is to appropriately choose L2P entries to cache within the limited HPB memory, in a UX-centric manner. While HPB parameters are set vendor-specifically [10], to the best of our knowledge, there are currently no HPB systems in production that actively consider the state of apps [11, 19–28]. The current upstream HPB device driver (included in the Android Common Kernel since v5.10 [15]) employs the *counter-based caching policy* and the *timer-based eviction policy* for efficient HPB memory management. However, we argue that both of these policies fail to improve user-perceived delays which we discuss in Section 4.

3 Related Work

Classifications of FG and BG apps are pivotal in maintaining good UX on both mobile [29–31] and desktop environments [32]. Academia also follows this trend and makes use of FG/BG separation to further improve UX. Marvin [33] and Acclaim [34] modify the memory management subsystem and improve the FG app's performance by de-prioritizing BG apps' memory pages. ASAP [7] categorizes memory pages and prefetches FG app-related pages to improve app switching time. FastTrack [1] accelerates FG I/O requests by resolving I/O priority inversion caused by BG apps.

Despite the great impact of storage performance on UX, little attention has been paid to optimizing L2P caching under mobile device environments. To the best of our knowledge, FOAM [6] is the only work that sophisticates an L2P cache to enhance user-perceived performance. FOAM assigns different priorities to L2P entries, depending on the type of apps (FG or BG) and the type of I/O requests (read or write) that access them. They argue that the FG apps and the read requests precede other counterparts in terms of user-perceived performance. As such, FOAM divides the L2P cache into four partitions, FG-read (FR), FG-write (FW), BG-read (BR), and

BG-write (BW), and it accordingly moves the L2P entries across partitions whenever they are referenced. When choosing a victim, FOAM evicts the partitions from the lowest to the highest priority (*i.e.,* BR, BW, FR, and FW).

While FOAM enhances UX by prioritizing the eviction of L2P entries associated with BG apps, it has two limitations. First, FOAM only assumes an in-device cache, having no consideration of an HPB-enabled system. Thus, its effectiveness is limited to the latest mobile environments where HPB is used due to the increased mapping table size. Second, FOAM does not perform well when the FG and BG apps are switched quickly. This situation happens when the user runs multiple apps simultaneously. In this case, because the effective distinction between the FG and BG apps is not clear, the eviction policy of FOAM might lead to an unintended result.

4 Empirical Study of HPB on Smartphones

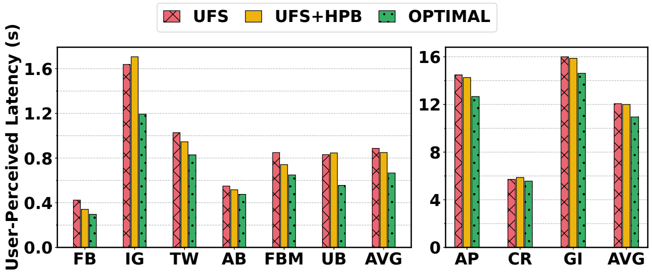
In this section, we empirically investigate how much the performance of FG apps is affected by the storage L2P cache. We first examine how the storage mapping cache affects the quality of UX on smartphones. Then, we assess the effectiveness of the existing HPB cache management policy and how it should be managed to boost UX.

4.1 Evaluation Study Setup

We conduct a set of experiments with a mobile hardware development kit based on the Snapdragon 888 SoC [17]. As for the benchmarks, we use nine popular smartphone apps [1] that are categorized into three types: games, social media, and utilities. We run the nine apps according to a predefined scenario that mimics real-world app-usage patterns of smartphones. In evaluating the UX, there exist various metrics such as app launching [35–41], app switching [7] and app loading [1]. These metrics are directly affected by the I/O performance as numerous libraries and files have to be loaded. In this section, we target app launching and loading times for the key metrics to assess the impact of L2P cache misses on UX, as they are the biggest contributors to the user-perceived latency.

We modify HPB in the Android kernel to implement various HPB cache management policies and to collect various performance-related statistics (*e.g.,* user-perceived latency and mapping cache hit ratios). Unfortunately, it is impossible to modify the firmware of UFS products. As an alternative, we develop a custom-emulated UFS device that mimics the behavior of production UFS devices using an Ultra-Low Latency SSD (ULL-SSD) [42]. The ULL-SSD has very low I/O latency (<20 μs) with extremely low variations, which makes it the perfect vehicle to emulate a slower UFS device.

We attach a 1-TB ULL-SSD as the main storage device for our custom-emulated UFS device. In between the HPB and the ULL-SSD, we run a UFS layer that implements UFS firmware algorithms, including L2P address translation, mapping cache management policies, and garbage collection. To emulate the I/O latency of UFS devices over the ULL-SSD, we also include a UFS I/O latency model [2] on the UFS layer. The UFS layer borrows a part of the host memory space and uses it as an L2P cache. For the UFS layer, we assign 512 KB of memory as an L2P cache space [18]. The HPB-allocated host memory size is set to 256 MB out of the 12 GB of host DRAM. Note that 1 GB of memory is required to cache the entire L2P mapping table. Since we use the same system and benchmark setups used in Section 6, more details of the experimental settings are explained in Section 6.1.



(a) Launching time. (b) Loading time.

Figure 2: Impact of L2P cache misses on user-perceived latency.

4.2 Impact of L2P Cache Misses on UX

In order to understand how much L2P cache misses affect the quality of UX, we quantitatively measure the user-perceived latency when an app is being launched and loaded. We measure the app launching time of apps from social media and utilities, and the app loading time of games while executing the app-usage scenario as described in Section 6.1.

Fig. 2 shows our experimental results. We compare the app launching time of three system setups: `UFS`, `UFS+HPB`, and `OPTIMAL`. `UFS` only uses a small cache (*i.e.,* 512 KB) to keep L2P entries. In addition to the UFS-level cache, `UFS+HPB` expands the capacity of the L2P cache by borrowing the host memory, 256 MB in our setup. `OPTIMAL` represents the optimal case that assumes the underlying UFS has sufficient memory space to keep the entire L2P entries. The `OPTIMAL` setup neither suffers from extra I/Os caused by L2P cache misses nor needs to steal host memory to expand its L2P cache size. Note that the difference in the latency between apps is due to different amounts of data needed for the execution of each app. By monitoring the memory consumption of each app, we observe that the maximum memory consumption gap is approximately 1 GB (between *FBM* and *GI*).

As expected, `OPTIMAL` exhibits the best performance across all apps, outperforming `UFS` and `UFS+HPB` by up to 50% for *UB* and 43% for *IG*, respectively. These results confirm that

[1] Asphalt9 (*AP*), Clash Royale (*CR*), Genshin Impact (*GI*), Facebook (*FB*), Instagram (*IG*), Twitter (*TW*), Airbnb (*AB*), Facebook Messenger (*FBM*), and Uber (*UB*) (see Section 6.1 for app usage scenarios).

[2] We acquired the numbers for the latency model through a discussion with a storage vendor since the official datasheet is not publicly available.

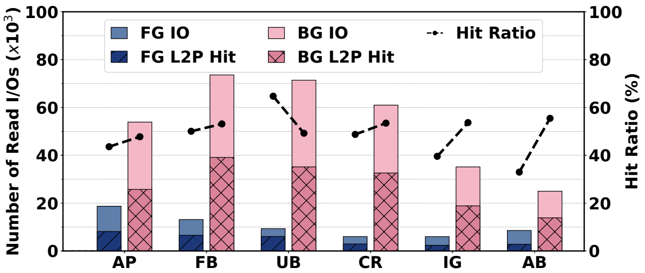


Figure 3: The number of read I/Os and the HPB hit ratios of FG and BG apps.

L2P cache misses greatly impact user-perceived delays. Even worse, absolute launch-time gaps are much wider than our expectations: 220 ms between `OPTIMAL` and `UFS`; 183 ms between `OPTIMAL` and `UFS+HPB`, on average. In order to deliver high-quality UX, reducing every millisecond matters [43, 44]. This is further emphasized by the recent mobile hardware trend of shipping displays with higher refresh rates [45–48]. For example, just 3.5 ms of delay can result in a noticeable stutter with a 144-Hz display [49]. It is important to optimize the user-perceived latency since it is well-known in the industry that a delay of just 100 ms can have significant consequences in online marketplaces [43].

We make two prominent observations from the above results. First, even though `UFS+HPB` borrows relatively a large amount of memory – 256 MB that can cache 25% of the entire L2P entries in its cache – from the host, it shows a marginal improvement in the app launching time. According to our observations (see Section 4.3), the L2P cache management policy fails to cache useful L2P entries that have a high impact on user-perceived latency. Instead, it often caches less important entries associated with BG apps, wasting valuable memory. Second, `UFS+HPB` shows worse performance than `UFS` for some apps – *CR*, *IG*, and *UB*. Our analysis reveals that stealing too much memory from the host incurs severe memory pressure. This leads to the frequent killing of apps, which results in many additional I/Os when the killed apps are launched again (see Section 4.4).

4.3 Impact of HPB Management Policy on UX

To figure out the root causes of why HPB performs poorly with a large mapping cache memory, we compare the hit ratios of FG and BG apps. We observe that FG apps suffer from higher miss ratios than BG apps, regardless of the cache size. Fig. 3 counts the number of I/Os issued by FG and BG apps and also displays how many of them are hit by the HPB cache. Except for *UB*, FG apps experience more L2P cache misses than BG apps.

We analyze detailed behaviors of state-of-the-art HPB management techniques. We find that the low hit ratios of FG apps are mainly due to wrong decisions made by a reference count-based L2P fetch policy and a timer-based eviction policy employed by the HPB manager in the Android kernel [15]. The HPB manager measures reference counts of LBAs and fetches L2P entries from the storage that have a large number of reads. However, as shown in Fig. 3, the number of read I/Os issued by BG apps is relatively larger than those by FG apps. L2P entries associated with BG apps are likely to have larger reference counts than those of FG apps. This results in unintended consequences that the HPB fetches L2P entries for BG apps. Simply fetching LBAs with large reference counts cannot guarantee improved UX.

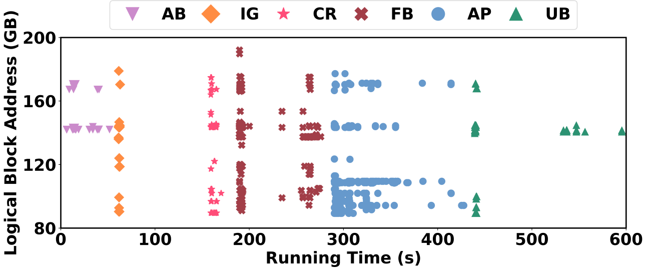


Figure 4: Read I/O access patterns of FG apps.

The timer-based eviction policy is another root cause that makes the HPB inefficient. Even when the HPB cache space is not full, HPB evicts a cached L2P entry that is not referenced for a predefined time (*e.g.,* 100 seconds in the Android Common Kernel v5.10 [15]). This timer-based eviction policy also does not consider the app usage patterns of the user, and thus often evicts L2P entries associated with FG apps. In general, after using an FG app for a while, a user moves to another app and then returns to the former FG app again. If the former FG app has not been used for a relatively long time, the timer-based policy would have evicted its L2P entries. When the user re-launches the former FG app, its L2P entries will no longer exist in the HPB memory, which results in mapping misses and may increase user-perceived delays.

Random I/O patterns that typically occur when an app is launched make it challenging for the HPB to provide high L2P hit ratios. Fig. 4 illustrates partial LBA access patterns of FG apps when they are launched and run for a while. As shown in Fig. 4, we observe that many small random reads, which span a wide range of LBAs, are heavily issued at the beginning of app launches. This randomness results in high L2P cache misses. Fig. 5 illustrates trends of the number of L2P cache misses over time for some selected apps in

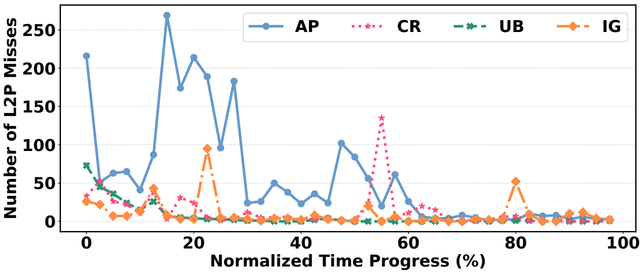


Figure 5: Distributions of the total number of L2P cache misses of FG apps over execution time.

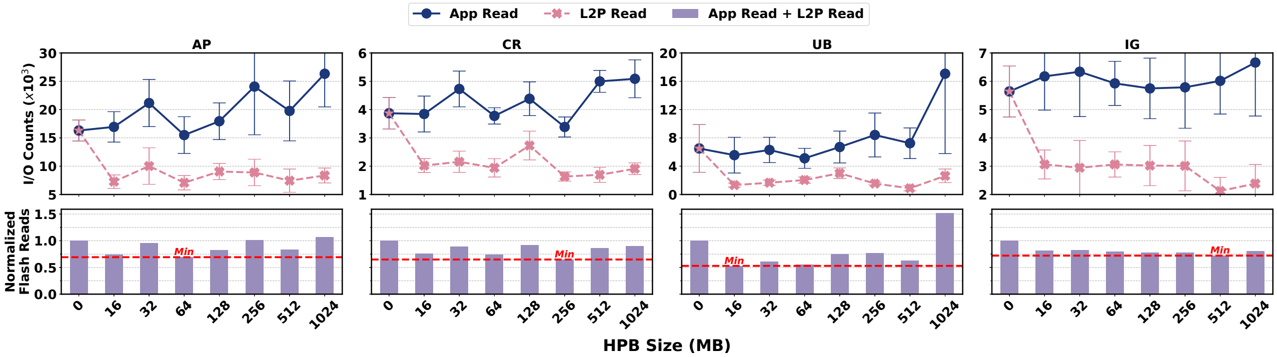


Figure 6: Number of read I/Os issued by FG apps and the corresponding HPB L2P misses with different HPB sizes.

Fig. 4. As expected from I/O patterns in Fig. 4, many L2P cache misses are concentrated in the early stages of an app's execution. For example, *UB* experiences 74% and 90%, and *AP* experiences 20% and 51% of its cache miss in the first tenth and the first fifth of its total execution time respectively. It is worth noting that even if a smartphone user does not manually close the used apps, it is practically infeasible to keep all apps open in BG due to memory constraints [16, 33, 50], even on devices with large amounts of DRAM [51]. Thus, random I/Os are inevitable in the mobile environment, and improving them is a key factor in providing a better UX.

4.4 Impact of HPB Size on UX

HPB shares the same host memory with the Android platform. To provide optimal performance to users, the size of HPB memory should be carefully tuned. Allocating large amounts of host memory to HPB is beneficial in improving L2P hit ratios. On the other hand, as mentioned in Section 4.2, assigning too much memory to HPB might result in UX degradation due to an increase in memory pressure. To prevent HPB from over-consuming memory, the HPB manager employs a timer-based eviction policy. However, as shown in Section 4.3, its FG app-oblivious decisions often cause side effects resulting in evictions of the FG app's cached L2P entries.

To understand how much the HPB memory size affects the user-perceived latency, we observe how the number of read I/Os changes while varying the HPB size usage from 0 to 1 GB. In our evaluation setup, 1 GB of memory is large enough to keep all of the L2P entries in HPB. With recent mobile devices with more and more DRAM (*e.g.,* 18 GB) [52], this amount may sound trivial. However, memory pressure is still often observed in Android systems [51, 53]. Contrary to server or desktop systems, Android tries to maximize memory utilization to maximize its caching capabilities by default [54]. Also, due to the general trend of apps using more resources [55], Android is often susceptible to high memory pressure even with a large capacity of memory. Consequently, relieving memory pressure on the Android system depends on low memory killer by terminating the least important apps. Thus, statically reserving a large amount of memory for improving storage performance is a short-sighted decision with no consideration of its impact on the overall UX.

From the experimental results shown in the Fig. 6, we make two key observations on the impact of the different HPB sizes. First, the optimal HPB size, which results in the minimal number of flash reads (*i.e.,* app reads + L2P reads), is different for each individual app. For example, *IG* shows the minimum number of flash reads with 512 MB whereas *UB* only needs 16 MB. Second, the number of FG apps issued I/Os (app reads) gradually increases as more and more memory is allocated to the HPB. Fig. 6 counts the number of I/Os issued from FG apps and the HPB. As the HPB size increases, thanks to the improved L2P hit ratios, L2P reads from the HPB tend to decrease. While at the same time, since HPB increases the memory pressure of the system, FG apps tend to issue an increased number of read I/Os (*e.g., UB* issues 142% more read I/Os with 1-GB HPB memory when compared to none of the host memory is allocated to the HPB).

Under memory pressure, Android starts killing apps to relieve memory pressure. LMKD uses pressure stall information (PSI) [50] provided by the Linux kernel to detect memory pressure situations, and decides when and how to kill apps. Using PSI, LMKD monitors memory pressure levels and kills the least important app repeatedly until the memory pressure is relieved. If the system consumes more memory, it naturally leads to LMKD killing more apps. As shown in Table 1, killed apps (cold state) not only take much longer to launch, up to 6.2×, but it also incurs much more I/Os, up to 12×, further degrading the UX [7, 34]. Hence, the HPB size is a trade-off regarding the overall UX which should be carefully tuned.

To understand how HPB affects the behavior of LMKD (*e.g.,* how often it kills and how important the victim app is), we analyze the LMKD kill counts for each priority cate-

	Warm state		Cold state	
	Launching time (ms)	**I/O counts**	**Launching time (ms)**	**I/O counts**
AP	352.6	250	2188.6	2164
CR	239	217	668.4	2879
UB	366.4	28	563	60
IG	482.3	349	1245.1	4224

Table 1: App launching time and the corresponding I/O counts of two different launching states.

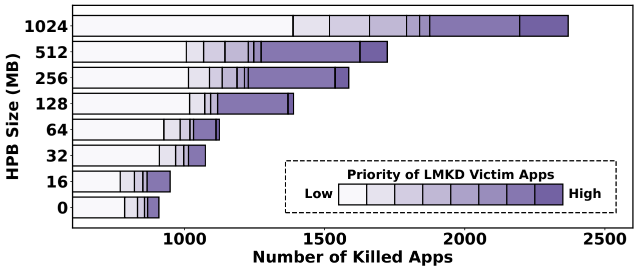


Figure 7: Number of LMKD killed apps and the proportion of the killed apps' priority with different HPB sizes.

gory. Fig. 7 shows the histogram of LMKD kill counts with different HPB sizes. The higher the priority is, the more user-perceptible the app is (*e.g.,* the second-highest apps are the user recently used ones but in the BG) [56]. As expected, an increase in the HPB size results in a greater number of apps and a higher proportion of high-priority apps being killed by LMKD. Even though the existing HPB scheme might provide better L2P hit ratios, UX degradation is inevitable.

Based on our observations, we conclude that the state-of-the-art timer-based HPB size adjustment policy is suboptimal in two aspects. First, while the timer-based HPB size adjustment could lower the HPB's memory usage, it cannot dynamically relieve memory pressure as HPB is unaware of the current memory pressure status. Second, when users run multiple apps simultaneously the system will suffer from severe memory pressure due to the increased memory utilization by both user apps and the HPB. In such a case, the timer-based eviction policy is unable to proactively and selectively evict cached entries as most HPB cached entries are recently referenced. Managing HPB memory with unawareness of the memory pressure status poses a significant risk of degrading the UX. To achieve the best HPB performance, the size of HPB memory should be dynamically adjusted by considering the memory pressure status while not sacrificing the L2P cache performance.

5 Design and Implementation of HPBvalve

Our empirical study presented in Section 4 reveals that the naïve integration of HPB to Android does not guarantee improved UX. Moreover, the existing techniques neither efficiently cache or evict L2P entries in the HPB memory, nor decide a proper size of the HPB memory from the perspective of maximizing the user-perceived performance.

To improve the user-perceived latency of smartphones, we should minimize the L2P cache misses of I/O requests from FG apps. If UX-sensitive I/O requests are always hit by the HPB memory, smartphone users experience the equivalent performance as if the entire L2P entries are cached. At the same time, to prevent user-noticeable and important apps from being killed by LMKD, we should wisely adjust the HPB size according to the status of the system memory pressure.

To accomplish the above goals, the existing HPB layer needs to be improved in two aspects. First, HPB should identify which I/Os are user-latency sensitive or not. Once identified, HPB should appropriately manage associated L2P entries in the L2P cache, particularly in an FG app-centric manner. Second, HPB should be aware of the memory pressure status of the Android system. Then, it should decide whether to increase or decrease the HPB memory size for higher L2P cache hit ratios or for relieving memory pressure.

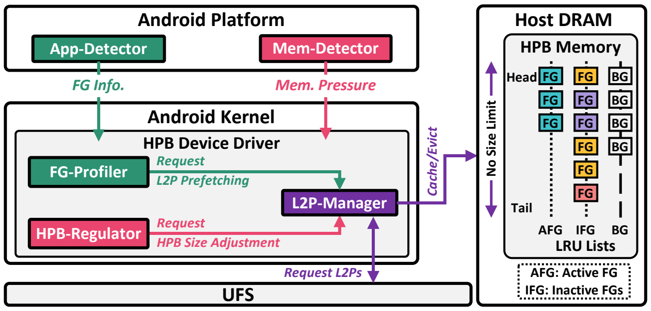


Figure 8: An overall architecture of HPBvalve.

Keeping the above observations in mind, we propose an integrated host-SSD mapping management scheme, called HPBvalve (Hvalve in short), which addresses the limitations of existing techniques. We aim to design Hvalve to be simple yet effective for its wide adoption in real-world devices. To this end, Hvalve leverages information that is already collected by other existing modules in the Android platform, which enables Hvalve to exploit a variety of information in a vertically-integrated manner at a low cost.

5.1 Overall Architecture of HPBvalve

Fig. 8 illustrates an overall architecture of Hvalve that is composed of five key modules – App-Detector, Mem-Detector, FG-Profiler, L2P-Manager, and HPB-Regulator. Hvalve has a cross-layered design that spans across a wide range of system layers from the Android platform to the kernel. Two modules, App-Detector and Mem-Detector, implemented in the Android platform monitor the system status and collect a set of information, including (*i*) the type of apps (*i.e.,* FG or BG) that issue I/Os, (*ii*) app state changes, and (*iii*) the memory pressure status. This information is then delivered to the HPB device driver in the Android kernel. Based on the delivered information, three modules implemented in the kernel, FG-Profiler, L2P-Manager, and HPB-Regulator, manage HPB in a UX-centric manner by (*i*) separately managing and profiling L2P entries of FG apps, (*ii*) prefetching the profiled L2P entries on every app launch, and (*iii*) adjusting the HPB size dynamically depending on the memory pressure status.

5.2 FG App-centric HPB Management

In this section, we explain how Hvalve manages L2P entries using the *FG app-centric caching policy*.

FG/BG classification: In order to identify user-latency-sensitive I/O requests, every submitted I/O has to be distinguished whether it is submitted by an FG or a BG app. To this end, we extend the kernel and Android framework so that the kernel I/O stack becomes aware of the app-level information. With our extension, every I/O request holds its caller UID (a unique number that the Android system assigns to every app). When a regular I/O system call is invoked, a new `struct bio` is allocated under the same process context. Since the same process context is maintained, the caller's process control block (`struct task_struct`) is accessible from the `bio` allocation step. We add a new member field in the `bio` to copy the caller's UID from the PCB. The new UID field can be used in deciding whether a `bio` belongs to an FG or BG app.

In order to distinguish whether the submitted I/O is from an FG app, the UID embedded in the request header has to be compared to the UID of a current FG app. **App-Detector** is designed to detect an FG app in the system. The App-Detector keeps track of every app state change (*e.g.,* a new FG launch-start and launch-end) by referring to Android's activity task manager [57]. Upon every app state change detected, App-Detector delivers a state change message to Hvalve. For example, when a new FG launch is detected, App-Detector delivers a launch-start signal (*e.g.,* a new FG app is launched) to the HPB in the kernel along with its UID. Hvalve makes use of the delivered information for distinguishing every FG app-submitted I/O. If an I/O request passed to the block layer has a different UID from the App-Detector-passed current FG app's UID, it is considered as a BG I/O. This makes it possible for Hvalve to manage HPB memory to assign higher priority to L2P entries of an FG app (*i.e.,* UX-sensitive L2P entries).

L2P management: In order to prioritize L2P entries of FG apps, the **L2P-Manager** manages cached L2P entries in three separate LRU lists depending on their importance –*AFG* (an active FG app), *IFG* (inactive FG apps), and *BG* (BG apps) lists, as illustrated in the Fig. 8. The reason for Hvalve managing cached entries with three different LRU lists is to differentiate the priority upon eviction. With these three separate LRU lists, Hvalve is able to give different priorities to the cached entries that are referenced by a user currently interacting FG app, previously user-interacted FG apps, and BG apps. Whenever an L2P of the current FG app gets cached to the HPB memory, it is first inserted into the AFG list. If a user launches a new FG app, the L2P entries in the AFG list get demoted to the IFG list as they now belong to the previous FG app. On the other hand, if an L2P entry from a BG app gets cached to the HPB memory, it directly goes into the BG list. Moving between lists, inserting or removing entries from each list is trivial as all lists are implemented with hash lists. With these three different LRU lists, Hvalve is able to manage the HPB memory in an FG app-centric manner.

Caching policy of Hvalve: Hvalve respects the HPB's counter-based caching policy, thus it also caches frequently referenced L2P entries to the HPB memory. In addition to this, Hvalve cache L2P entries that satisfy the following conditions. First, for every HPB cache miss that originated from the current FG app, the L2P-Manager immediately caches the corresponding L2P entry to the HPB memory to prevent further cache misses of the said L2P. Second, a set of L2P entries of the launched FG app are directly cached to the HPB memory with the help of the **FG-Profiler** upon every app launch signal. FG-Profiler is designed to collect information on the current FG app to help HPB prioritize FG L2P entries to improve the launching time as well as the total user-perceived latency.

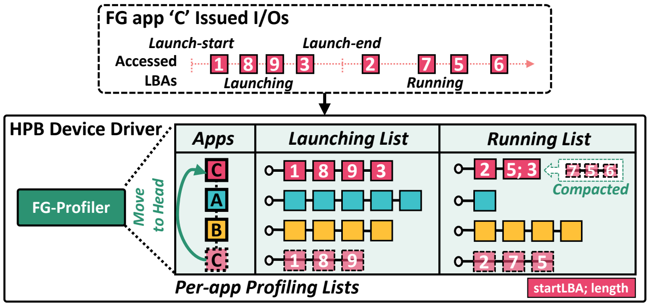


Figure 9: An example of per-app profiling lists being updated.

FG app profiling and prefetching: As discussed in Section 4.3, every app has its unique I/O patterns. To better manage the L2P cache based on apps' unique I/O characteristics, FG-Profiler maintains an LRU list of recently used FG apps, each containing two separate L2P profiling lists: *app launch list* and *app running list*. The app launch list contains a list of LBAs that are referenced during an app launch (*i.e.,* from a launch-start signal to a launch-end signal, delivered by App-Detector). The app running list holds a list of LBAs that are accessed during the execution of the app (*i.e.,* after a launch-end signal).

Fig. 9 illustrates an example of how the FG-Profiler maintains the L2P profiling lists of FG apps. Once it receives a launch-start message, it adds or moves the new FG app to the head of the app-LRU list. Until a launch-end signal arrives, it profiles every L2P entry needed by the launched app.

Fig. 10 shows an overview of how the L2P-Manager prefetches L2P entries for every app launch. When the App-Detector is notified of a launch-start of a new FG app it sends a launch-start signal to HPB (❶). Upon every launch-start signal, the Hvalve tracked UID of the current FG app gets updated to the UID of the new FG app. Then, the FG-Profiler searches for the previously created profiling list of the new FG app (❷). If the profiled list is found, the FG-Profiler requests the L2P-Manager, (❸), to prefetch the L2P entries from the list. This hides mapping miss penalties of the new FG app during an app launching, as well as running.

Since every L2P prefetch request results in a new flash page read operation, the FG-Profiler first prioritizes prefetching L2P entries from the launch list. Only after prefetching the L2P entries profiled in the launch list, the entries in the running list are requested to be prefetched. The L2P-Manager preferentially fetches the FG-Profiler-requested L2P entries from the UFS (❹). Those prefetched FG L2P entries are then inserted to the tail of the FG LRU list, AFG, rather than the head of the

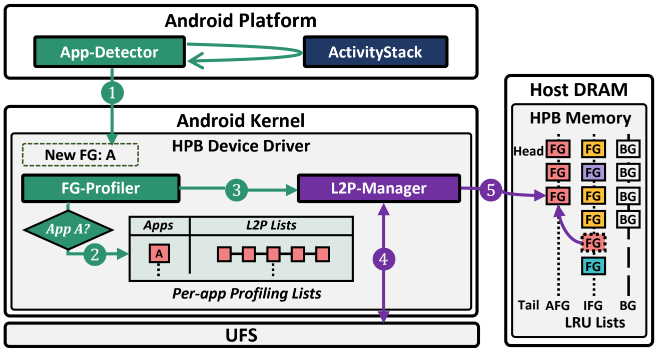


Figure 10: L2P prefetching mechanism of a new FG app.

list (❺). This is to differentiate priority upon victim selection between the actually referenced entries by the FG app and the Hvalve-prefetched entries. Even if the prefetched entries are inserted into the tail of the AFG list, Hvalve does not attempt to evict them until they are demoted to the IFG or BG lists.

Managing L2P lists per app is not only efficient for better L2P hit ratios of FG apps, but also space-efficient in utilizing the host memory. The profiling list is maintained as a pair of the start chunk of the accessed LBA and the length of the neighboring referenced chunks. Each of the profiling lists only requires a few tens of kilobytes for most apps, 7 KB on average. Moreover, to prevent the per-app L2P profiling lists from excessively consuming the host memory, we statically limit the maximum size of the total profiling lists to 1 MB. Hvalve does not provide special handling for deduplicating entries between app lists. Since the size of an entry of the profiled per-app list is only 4 bytes, it is unlikely that significant memory gain can be achieved by removing duplicate entries between lists. If the number of profiled apps exceeds the predefined limit, the FG-Profiler frees the least recently used app from the list to allow the newly launched FG app to be profiled. Our current implementation sets this number to 20, which corresponds to the average smartphone usage [34]. This keeps the entire per-app L2P lists in less than 1 MB of memory – a negligible space overhead. It can also be easily extended to dynamically adjust if needed (see Section 6.3 for more details).

5.3 Dynamic HPB Size Adjustment

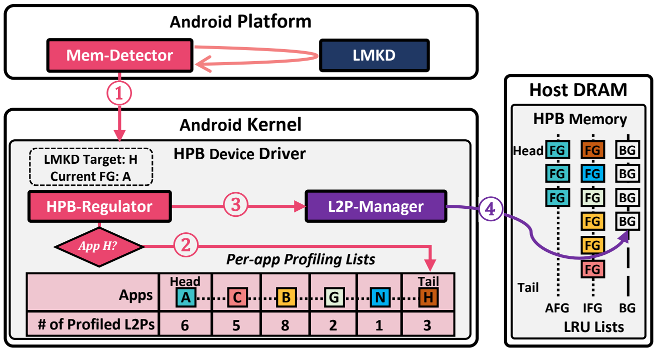
To the best of our knowledge, there exist no techniques that optimally decide the HPB size to provide the best performance by considering its impact on the overall UX quality. Hvalve neither insists on statically allocating small HPB size to minimize its impact on memory pressure nor large HPB size to boost L2P hit ratios. Instead, Hvalve proposes a *dynamic HPB size adjustment scheme* that adjusts the HPB size based on the monitored memory pressure status. Hvalve adaptively controls the HPB memory size for higher I/O performance while no user-interacting apps are mistakenly killed by LMKD. Hvalve is unique in that I/O performance improvements are achieved without negatively affecting UX-critical factors.

With Hvalve, if a non-memory-intensive app runs and the system has enough free memory, the HPB size can increase to cover the entire L2P mapping table. This enables us to maximally exploit the full benefits of HPB. However, whenever the system starts to experience memory pressure, Hvalve immediately adjusts the HPB size accordingly, returning memory for apps to use. As a result, the degradation of UX by excessively assigning host memory to HPB does not occur.

Fig. 11 illustrates how Hvalve dynamically adjusts the allocated HPB memory. As discussed in Section 4.4, Android employs LMKD which selectively kills running apps to relieve memory pressure. Once LMKD decides on a victim app to kill, the **Mem-Detector** notifies the **HPB-Regulator** with the target's UID before LMKD starts killing the victim app (①). The HPB-Regulator decides how important the victim app is by checking the per-app L2P profiling list (②). If the victim app is not found in the per-app L2P profiling list (*i.e.,* not a user recently used app), the signal is ignored and leaves LMKD to continue killing the victim app. On the other hand, if the victim app exists in the per-app L2P profiling list, it is treated as an important app (*i.e.,* a user recently interacted app). Then to prevent the important app from being killed by LMKD, Hvalve preferentially reclaims the HPB memory by aggressively evicting low-priority cached L2P entries.

To decide how much memory to free from HPB, the properties of the LMKD victim app are taken into consideration. As LMKD calculates the expected amount of memory to be freed upon its victim selection, Hvalve attempts to free as much as the LMKD desired amount. When the HPB-Regulator tells L2P-Manager how much HPB memory to free (③), it delivers how many cached entries in the HPB memory should be evicted to prevent the LMKD killing its victim app.

The L2P-Manager considers the priority of cached L2P entries when choosing which entries to evict. Hvalve marks every cached L2P entry with its last-referenced UID, and stores to an LRU list (*e.g.,* AFG, IFG or BG) depending on the type of the app it belongs to. With the three separate LRU lists, Hvalve can make a fine-grained decision on which entry to evict first as Hvalve is aware of which entries belong to the



	Head					Tail
Apps	A	C	B	G	N	H
# of Profiled L2Ps	6	5	8	2	1	3

Figure 11: HPB size adjustment mechanism of Hvalve.

BG apps (*BG list*), a user recently used FG apps (*inactive FG list*), and a currently user-facing FG app (*active FG list*). The L2P-Manager starts to evict entries from the tail of the *BG list*, (④). If there are no more entries from the *BG list* to evict, the ones from the *inactive FG list* are tried next. L2P-Manager never evicts entries from the *active FG list* to avoid UX degradation. The entries in the *inactive FG list* will naturally get evicted by the L2P-Manager if it is left unused for a long time. On the other hand, the entries in the *BG list* can be promoted to the *FG list* if an FG app references entries in the *BG list*.

The proposed dynamic HPB size adjustment policy of Hvalve may result in an increase in the execution time of the LMKD's app-killing process. This occurs when there is an insufficient number of HPB cached entries that can be evicted to alleviate the memory pressure. In such a case, LMKD has to resume the paused app-killing process to reserve free memory space. The increased amount of execution time, however, is marginal compared to the relatively longer procedure of LMKD. In addition, from a long-term perspective, the dynamic HPB size adjustment is much more beneficial to the overall UX quality as it prevents user apps from being killed. A more detailed analysis is described in Section 6.3.

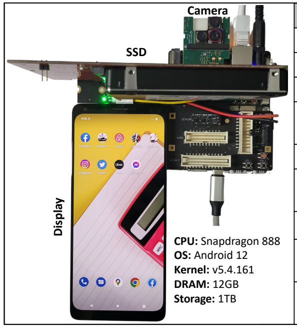
6 Experimental Results

In this section, we evaluate the overall quality of UX when Hvalve is applied to an HPB-enabled system.

6.1 Experimental Setup

To evaluate the effectiveness of the proposed techniques, we implement the App-Detector, Mem-Detector, FG-Profiler, L2P-Manager, and HPB-Regulator of HPBvalve on a Snapdragon 888 Mobile HDK [17], which is illustrated in Fig. 12. Our evaluation platform uses the same board support package that is used on other production smartphones using the same SoC. This HDK has 12 GB of DRAM (effectively 8 GB) and PCIe 3.0 x2 connectivity, which we use to connect PCIe peripherals. We use Android 12 and Linux kernel v5.4.161 to implement Hvalve. As it is practically infeasible to modify the UFS firmware, we use an ultra-low latency NVMe SSD [42] described in Section 4.1 and implement a lightweight FTL in the kernel to mimic UFS storage, which consumes approximately 4 GB of memory to run our test scenarios. We have written about 1,000 LOC to implement Hvalve, which we open-sourced on GitHub[3], including the changes made to the Android platform and the kernel.

We use `am start` command [35] to measure app launching and switching times. For apps with multiple loading stages, the `am start` command is unable to measure the total time taken until the device is ready to take user inputs. For example, *GI* goes through three separate loading stages until the gameplay button appears while the `am start` command only measures the time taken until the first stage. To precisely measure the app loading time, we employ an external high-speed camera, which captures 120 frames per second that match the refresh rate of our evaluation platform's display.



Apps	App Usage Scenario
AP	Play a stage.
CR	Play a stage.
GI	Explore the map.
FB	Browse Feed, Watch, Stories, Notifications, and other pages.
IG	Browse Feed, Reels and write comments.
TW	Browse Timeline, Trending, Notifications , and reply to a tweet.
AB	Browse the globe map and opens multiple listings.
FBM	Browse Chats, Calls, People, Stories, and send messages.
UB	Browse Activity, Account settings, and search the map.

Figure 12: Prototype HPBvalve setup and app usage scenarios.

As for the benchmarks, we evaluate nine popular mobile apps listed in Section 4.1. To automatically run multiple mobile apps under realistic app usage scenarios, we use the Android debug bridge (*adb*) [58]. The predefined app usage scenarios are described in Fig. 12. To avoid cherry-picking sequences that would favor Hvalve, the sequence of the nine apps is randomized and run multiple times to reduce variables. The chosen random sequence is executed for each technique for a fair comparison.

Even though the same randomized sequence was run for each technique, there still exists run-to-run variations due to noises such as network conditions, random advertisement occurrences, and others. In order to minimize run-to-run variations, we fully automated the evaluation process to repeat the same scenario twenty times for each case. We also disabled the checkpoint on the underlying file system, f2fs [59], so that the entire *userdata* partition could be rolled back to the previous state to further minimize variances. After running twenty times, we averaged the results of fifteen runs, excluding the five outliers. We compared Hvalve with the typical HPB system that employs FOAM [6] as its L2P eviction policy, `UFS+HPB`, and an ideal system where all L2P entries are cached in memory, `OPTIMAL`, and a conventional UFS system without HPB, `UFS`. We also compared Hvalve-Only with `HPB-Only` where the underlying UFS SRAM is not considered to evaluate the effectiveness of caching policy of Hvalve.

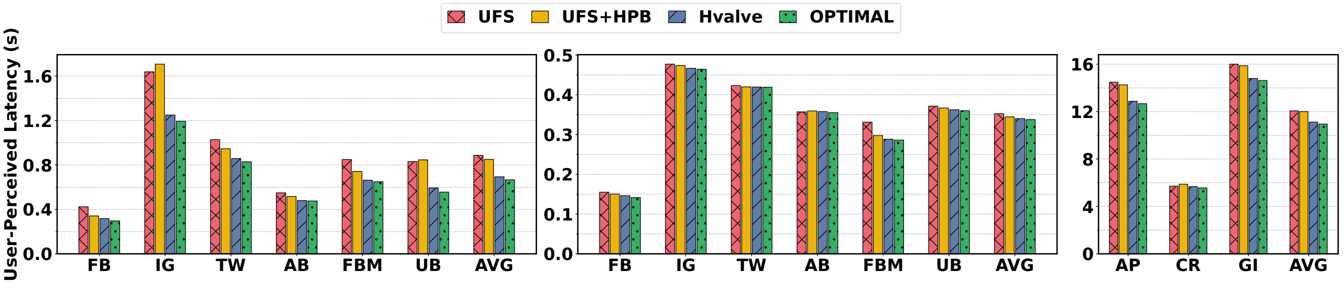
6.2 Performance Evaluation

In order to validate the effectiveness of Hvalve, we assess the impact of FG app-centric HPB management and dynamic HPB size adjustment techniques on the overall UX quality compared against `UFS+HPB`.

6.2.1 FG app-centric HPB management

User-perceived latency: As for the most important performance evaluation metrics in deciding the quality of UX, we assess the app launching, switching, and loading times of various HPB configurations.

[3] The source code is available at https://github.com/cares-davinci/Hvalve.



(a) App launching time of social media and utilities.

(b) App switching time of social media and utilities.

(c) App loading time of games.

Figure 13: Evaluation results of the user-perceived latency.

Fig. 13(a) shows the experimental results of the app launching time. The average app launching time of Hvalve is improved by 28% and 23% when compared to UFS and UFS+HPB, respectively. The maximum and minimum app launching time improvements between Hvalve and UFS+HPB are 43% for *UB* and 7% for *FB* where the corresponding absolute app launching time improvements are 237 ms and 23 ms, respectively. We also compare the app launching time of Hvalve to OPTIMAL. The minimum and the maximum absolute launch-time differences between OPTIMAL and Hvalve are only by 4 ms (+0.85%) for *AB* and 55 ms (+4.7%) for *IG*. On the other hand, UFS+HPB takes 40 ms (+8.5%) for *AB* and 514 ms (+43.1%) for *IG* more when compared to OPTIMAL.

We also measure the app switching time – the latency of when a user switches back to an app that is recently launched. As shown in Fig. 13(b), Hvalve outperforms UFS+HPB for all cases. The maximum app switching time difference between Hvalve and OPTIMAL is only 4.6 ms (+3.15%) with *FB* while it is 13.4 ms (+8.65%) on UFS+HPB in the same scenario. The minimum increase in app switching time of Hvalve compared to OPTIMAL is 0.5 ms (+0.12%) with *TW*, while the gap between UFS+HPB and OPTIMAL of the same case is 4 ms (+0.95%). Since a comparatively small number of I/Os are issued while an app is being switched compared to the app launch process, as described in Table 1, the performance increase with Hvalve for the app switching time is quite marginal compared to the other two metrics.

We also evaluate the app loading time of games by using an external high-speed camera to measure the time from a user launch of an app until the device is ready to take user inputs. As shown in Fig. 13(c), the app loading time of all three games is improved which provides almost the same user-perceived latency as OPTIMAL. The minimum increase in app loading time of Hvalve compared to OPTIMAL is 111.4 ms (+2%) with *CR* while the gap between UFS+HPB and OPTIMAL of the same case is 308.45 ms (+5.5%). The maximum app loading time difference between Hvalve and OPTIMAL is only 205 ms (+1.6%) with *AP* while it is 1589 ms (+12.5%) on UFS+HPB with the same scenario. When comparing Hvalve to OPTIMAL, the increase in user-perceived latency, mainly caused by mapping misses, is very marginal.

As discussed in Section 4.2, every millisecond of responsiveness greatly impacts the UX. According to our evaluation results, the storage mapping miss penalties, resulting in user-perceived delays, are significantly alleviated with Hvalve. These benefits come from the FG app-centric HPB management policy employed in Hvalve, which gives higher priority to L2P entries of FG apps to be managed in the HPB memory. To summarize, Hvalve alleviates the storage mapping miss penalties of the baseline UFS+HPB by 80% for app launching time and by 86% for app loading time on average.

L2P miss patterns: To analyze the impact of our proposed FG app-centric HPB management of Hvalve on FG apps, we first observe how much L2P miss distributions differ from UFS+HPB over an app execution. Due to the page limit, we include four representative apps, *AP*, *CR*, *UB*, and *IG*, two from games and one each from social media and utilities.

As examined in Section 4.2, most of the L2P cache misses occur during the early stages of the total app execution time. Fig. 14 shows the distributions of L2P misses over the execution time of each FG app. Such mapping misses that occur in the early stage are one of the root causes that increase user-perceivable delays. Hvalve significantly reduces the peak number of L2P misses during the early stage as well as the total number of the L2P cache misses compared to UFS+HPB. The maximum L2P miss reduction in the first tenth of the total normalized execution time is 88% with *CR* over UFS+HPB, while the minimum is 45% with *AP*. During the first fifth of the app execution time, 71% and 75% of the mapping misses are alleviated for *UB* and *IG*, respectively.

The peak L2P misses are greatly reduced not only in the early stages, but also throughout the entire execution time. This advantage comes from both the caching and the eviction policy of Hvalve. Hvalve prepares L2P lists by prefetching when an app is being launched, and those of the prefetched or cached entries of the current FG L2P entries never get evicted from HPB memory as long as it remains as an FG. The mapping miss penalties included in the user-perceived latency, which could severely degrade the overall quality of UX, are greatly reduced.

Hit ratios: In addition to the reduced user-perceived latency and L2P misses, we also quantitatively evaluate the

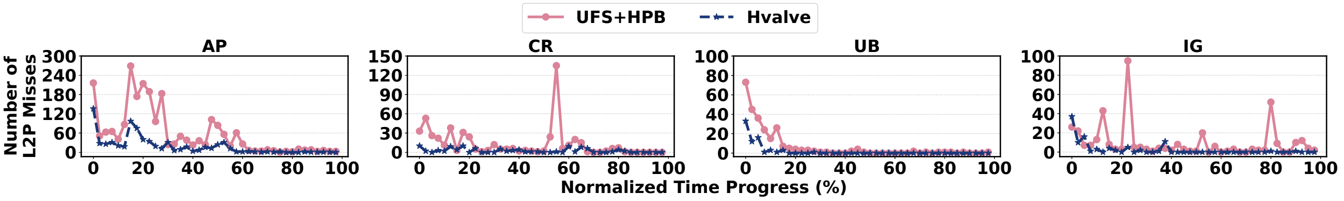


Figure 14: Distributions of L2P misses of FG apps over execution time.

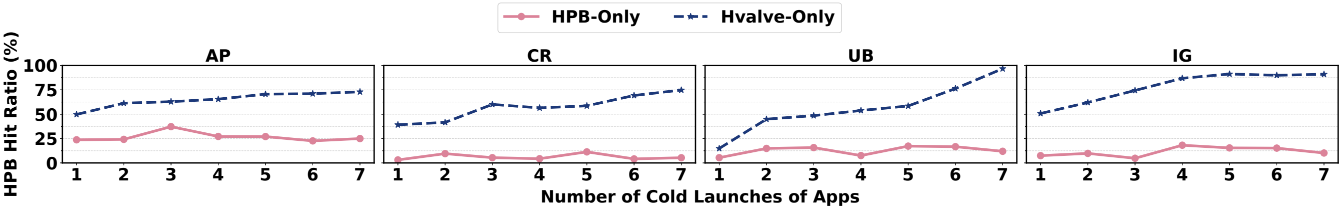


Figure 15: HPB hit ratios of each launch when consecutively launched for seven times.

effectiveness of the FG app-centric L2P management policy of Hvalve, by comparing HPB-Only to Hvalve-Only where the underlying UFS SRAM is not considered. Fig. 15 illustrates the HPB hit ratios of each FG app. The hit ratios of Hvalve-Only outperform HPB-Only for all cases. In order to evaluate the effectiveness of the L2P prefetching scheme of Hvalve, we observe how the hit ratios of each FG app change over a few numbers of consecutive app launches.

As the number of app launch counts increases, the hit ratios of Hvalve keep increasing while the hit ratios of HPB-Only remain consistently low. The largest hit ratio difference between HPB-Only and Hvalve-Only is 84.79% with *UB*. This result proves that the L2P prefetching mechanism successfully improves the performance of FG apps by hiding miss penalties. The above results also confirm that Hvalve is effective in providing a better quality of UX as it actively reflects the app usage patterns of individual smartphone users in managing the HPB memory.

6.2.2 Dynamic HPB size adjustment

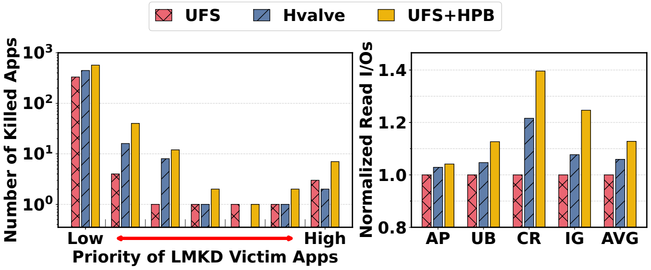
We compare Hvalve with UFS+HPB to evaluate the impact of our proposed dynamic HPB size adjustment scheme on UX. While Hvalve dynamically adjusts the HPB size depending on the monitored memory pressure status, UFS+HPB statically allocates the HPB size and adjusts it with a simple timer-based eviction policy. In this evaluation, we set UFS as a baseline, which does not require extra host memory to load storage mapping entries (*i.e.,* no impact on the memory pressure).

Fig. 16(a) shows a log-scaled histogram of the number of LMKD killed apps with seven different priority categories. As no extra host memory is used for allocating HPB memory (*i.e.,* no extra memory pressure), UFS results in the lowest number of kills on every priority category. On the other hand, the result of UFS+HPB shows much more apps were killed on every priority category when compared to UFS. The number of killed apps in the top three priority categories was increased by 5× with UFS+HPB when compared to the UFS. While providing much higher L2P cache hit ratios, Hvalve reduces the number of apps killed by LMKD in the top three priority categories by 70% compared to UFS+HPB. Therefore, we again prove that simply integrating HPB into the system inevitably increases the memory pressure resulting in high-priority user apps being killed.

To further investigate the consequences of high-priority apps being killed by LMKD, Fig. 16(b) shows the change in the number of read I/Os issued by FG apps, normalized to the number of read I/Os of UFS. The number of read I/Os with UFS+HPB is increased by 13% on average when compared to UFS, while Hvalve is only increased by 5% as it reduces the impact on the FG apps as well.

To demonstrate how each UFS+HPB and Hvalve adjust the allocated HPB memory, Fig. 17 illustrates changes in HPB memory size along with the reported memory pressure signals. Unsurprisingly, UFS+HPB allocates and frees HPB memory in a non-harmonized manner with the overall memory pressure status. Even when the memory pressure is present, UFS+HPB still allocates host memory to HPB (*i.e.,* adding more memory pressure to the system) which increases the possibility of important apps getting killed by LMKD.

On the other hand, Hvalve allocates host memory to the HPB memory and also effectively returns the HPB memory to



(a) Histogram of LMKD killed apps.

(b) Number of app-issued read I/Os.

Figure 16: Impact of HPB on FG apps.

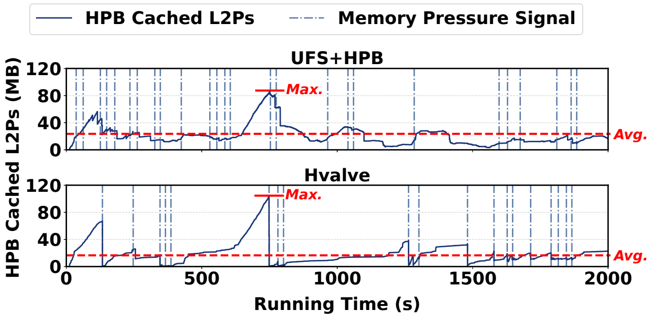


Figure 17: Changes in HPB memory size with the memory pressure signals.

user apps in a harmonized manner with the memory pressure signals. As a consequence, shown in Fig. 16(a), the number of important apps killed by LMKD with Hvalve is much lower than that of UFS+HPB. This is because Hvalve proactively prevents LMKD from killing high-priority apps by promptly returning the HPB-allocated memory to the user apps under the monitored memory pressure signals.

6.3 Overhead Analysis

Space overhead: Unlike the existing HPB technique, Hvalve consumes extra memory space for maintaining the per-app profiling lists to prefetch L2P entries upon every FG app launch. To avoid excessive memory consumption in managing the per-app profiling lists, Hvalve profiles only a moderate number (*e.g.,* 20) of recently used apps (which has a high impact on UX). The total profiling app list size is also regulated to 1 MB. The number of profiled L2P entries of each app differs based on the app access patterns. The total memory consumption for per-app profiling lists of the nine apps we use throughout the evaluations only consumes 99 KB of memory. It can be further minimized by merging neighboring groups as each node can be transformed into a compact profiling list that contains <Start LBA, Length>.

Performance overhead: While Hvalve manages the cached L2Ps with three separate LRU lists, AFG, IFG, and BG, moving cached entries between the LRU lists does not necessitate exhaustive search overhead since all lists are managed with hash lists. Retrieval and relocating a specific cached L2P entry from one list to another list can be done in $O(1)$ time complexity. The process of promoting or demoting an entry from one list to another is also simple as it requires updating only a few associated pointers.

The overhead of dynamic HPB size adjustment is also negligible as the process of returning HPB memory can be executed comparatively faster than that of the LMKD's app-killing process. Hvalve takes about 1.8 ms on average to free HPB memory whereas the app-killing process of LMKD typically takes hundreds of milliseconds. Although unlikely, in the worst case when Hvalve cannot evict sufficient cached entries to free the requested amount of memory, an extra time overhead (a few ms) can incur as LMKD must resume the suspended app-killing procedure. Despite the potential of Hvalve increasing the LMKD's execution time, preferentially freeing the HPB memory under memory pressure is more advantageous to the overall system performance. For example, if an app is killed by LMKD and re-launched after a while, the evicted app-related data has to be reloaded. On the other hand, re-fetching the HPB entries only requires a much smaller number of I/Os.

Energy Consumption: Employing Hvalve does not require extra energy consumption, since the two proposed HPB management schemes of Hvalve do not introduce severe search or space overheads to the system. As the total execution time of apps is reduced by integrating Hvalve, the total energy consumption with Hvalve is even decreased by 3.51% compared to UFS+HPB. When comparing to OPTIMAL, UFS+HPB increases the total energy consumption by 4.02% while Hvalve only increases by 0.56%. Hvalve successfully mitigates the negative impacts of UFS+HPB on resource consumption and the overall quality of UX.

7 Conclusion

In this paper, we present a novel FG app-centric L2P mapping cache management scheme, HPBvalve, for the HPB-enabled system. Hvalve is motivated by the fact that the existing HPB management scheme fails to improve the UX of smartphones due to two main reasons revealed through our empirical investigations. First, the priority of app status (FG or BG) is not considered while managing cached L2P entries in the HPB memory. Second, the memory pressure of the system could get critically high as host memory is allocated to the HPB without considering the memory pressure status. To improve the overall quality of UX upon these shortcomings of the existing HPB, Hvalve prioritizes the FG app in managing the cached L2P entries in HPB memory and dynamically adjusts the size of the HPB-designated host memory by monitoring the current memory pressure status. This allows Hvalve to reduce app kills in the top three priority categories by 70% while achieving significantly higher L2P hit ratios for FG apps. Our experimental results show that Hvalve successfully improves the overall UX quality of smartphones and provides almost equivalent performance as if most entries are cached in the HPB memory.

Acknowledgments

We would like to thank Ali R. Butt, our shepherd, and anonymous reviewers for their valuable suggestions. This work was supported by Samsung Electronics Co., Ltd (IO201207-07809-01) and the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (NRF-2022R1A2C2010529). The ICT at Seoul National University provided research facilities for this study. Sungjin Lee was supported by the National Research Foundation of Korea (NRF-2018R1A5A1060031). (Co-corresponding Authors: Sungjin Lee and Jihong Kim)

References

[1] 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)*, pages 15–28. USENIX Association, 2018.

[2] Google - Android Open Source Project. Performance Management. https://source.android.com/devices/tech/power/performance.

[3] 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)*, pages 345–358. USENIX Association, 2017.

[4] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices. In *13th USENIX Conference on File and Storage Technologies (FAST)*, pages 191–202. USENIX Association, 2015.

[5] Samsung. UFS. https://semiconductor.samsung.com/estorage/ufs.

[6] Chao Wu, Qiao Li, Cheng Ji, Tei-Wei Kuo, and Chun Jason Xue. Boosting User Experience via Foreground-Aware Cache Management in UFS Mobile Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 39(11):3263–3275, 2020.

[7] Sam Son, Seung Yul Lee, Yunho Jin, Jonghyun Bae, Jinkyu Jeong, Tae Jun Ham, Jae W. Lee, and Hongil Yoon. ASAP: Fast Mobile Application Switch via Adaptive Prepaging. In *2021 USENIX Annual Technical Conference (ATC)*, pages 365–380. USENIX Association, 2021.

[8] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. *SIGARCH Comput. Archit. News*, 37(1):229–240, 2009.

[9] Wookhan Jeong, Hyunsoo Cho, Yongmyung Lee, Jaegyu Lee, Songho Yoon, Jooyoung Hwang, and Donggi Lee. Improving Flash Storage Performance by Caching Address Mapping Table in Host Memory. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, page 19. USENIX Association, 2017.

[10] JEDEC. Universal Flash Storage (UFS) host performance booster (HPB) extension, version 2.0. https://www.jedec.org/standards-documents/docs/JESD220-3A.pdf, 2020.

[11] Google - Android Open Source Project. Google Pixel 3 kernel source - drivers/scsi/ufs/ufshpb.c. https://android.googlesource.com/kernel/msm/+/23d68f4b84c3c6a309512f9fef6d80072fb8364a, 2018.

[12] Masafumi Takahashi. UFS Unified Memory Extension. JEDEC Mobile Forum, 2014.

[13] Konosuke Watanabe, Kenichiro Yoshii, Nobuhiro Kondo, Kenichi Maeda, Toshio Fujisawa, Junji Wadatsumi, Daisuke Miyashita, Shouhei Kousai, Yasuo Unekawa, Shinsuke Fujii, Takuma Aoyama, Takayuki Tamura, Atsushi Kunimatsu, and Yukihito Oowaki. 19.3 66.3KIOPS-random-read 690MB/s-sequential-read universal Flash storage device controller with unified memory extension. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 330–331, 2014.

[14] NVM Express. NVMe specifications 1.2. http://www.nvmexpress.org/specifications, 2014.

[15] Google - Android Open Source Project. Android Common Kernel's HPB. https://android.googlesource.com/kernel/common/+/refs/heads/android12-5.10/drivers/scsi/ufs/ufshpb.c.

[16] Google - Android Open Source Project. Low Memory Killer Daemon. https://source.android.com/docs/core/perf/lmkd.

[17] Qualcomm. Snapdragon 888 HDK. https://developer.qualcomm.com/hardware/snapdragon-888-hdk, 2021.

[18] Jung-Hoon Kim, Sang-Hoon Kim, and Jin-Soo Kim. Utilizing Subpage Programming to Prolong the Lifetime of Embedded NAND Flash-Based Storage. *IEEE Transactions on Consumer Electronics (TCE)*, 64(1):101–109, 2018.

[19] OnePlus - GitHub. OnePlus 9 kernel source - drivers/scsi/ufs/ufshpb.c. https://github.com/OnePlusOSS/android_kernel_oneplus_sm8350/blob/1c052de944b391dd50957b03e1fc92f93f35e125/drivers/scsi/ufs/ufshpb.c, 2022.

[20] OnePlus - GitHub. OnePlus 9 kernel source - drivers/scsi/ufs/ufshpb_skh.c. https://github.com/OnePlusOSS/android_kernel_oneplus_sm8350/blob/1c052de944b391dd50957b03e1fc92f93f35e125/drivers/scsi/ufs/ufshpb_skh.c, 2022.

[21] OnePlus - GitHub. OnePlus 9 kernel source - fs/hpb_supp.c. https://github.com/OnePlusOSS/android_kernel_oneplus_sm8350/blob/1c052de944b391dd50957b03e1fc92f93f35e125/fs/hpb_supp.c, 2022.

[22] Xiaomi - GitHub. Redmi 10X, Redmi 10X Pro, Redmi K30 Ultra kernel source - drivers/scsi/ufs/ufshpb.c. https://github.com/MiCode/Xiaomi_Kernel_OpenSource/blob/9be022c0db171ac622e528752410d613c0e4e64d/drivers/scsi/ufs/ufshpb.c, 2021.

[23] Xiaomi - GitHub. Redmi 10X, Redmi 10X Pro, Redmi K30 Ultra kernel source - drivers/scsi/ufs/ufshpb_skh.c. https://github.com/MiCode/Xiaomi_Kernel_OpenSource/blob/9be022c0db171ac622e528752410d613c0e4e64d/drivers/scsi/ufs/ufshpb_skh.c, 2021.

[24] Samsung - Samsung Open Source. Galaxy S20 kernel source - drivers/scsi/ufs/ufshpb.c, fs/hpb_supp.c. https://opensource.samsung.com/uploadSearch?searchValue=G981NKSU1FUL9, 2021.

[25] Motorola - GitHub. Motorola Edge 30 kernel source - Samsung HPB. https://github.com/MotorolaMobilityLLC/kernel-msm/commit/a6cc1ed04b2a76fd325929a7925c01a99a310e0d, 2022.

[26] Motorola - GitHub. Motorola Edge 30 kernel source - SK Hynix HPB. https://github.com/MotorolaMobilityLLC/kernel-msm/commit/e297cf514d64bf85fdc2a1ee8722da8baf0afd4c, 2022.

[27] Motorola - GitHub. Motorola Edge 30 kernel source - Micron HPB. https://github.com/MotorolaMobilityLLC/kernel-msm/commit/26150eb5bb48d37b60f279b976676192 6ba17de3, 2022.

[28] Motorola - GitHub. Motorola Edge 30 kernel source - Kioxia HPB. https://github.com/MotorolaMobilityLLC/kernel-msm/commit/f4da8e2db63f5d5818abc37ba1677b79e604bacd, 2022.

[29] Google - Android Open Source Project. Identifying Capacity-Related Jank. https://source.android.com/docs/core/debug/jank_capacity.

[30] Google - Android Open Source Project. Cgroup Abstraction Layer. https://source.android.com/docs/core/perf/cgroups.

[31] Google - Android Open Source Project. Cached Apps Freezer. https://source.android.com/docs/core/perf/cached-apps-freezer.

[32] Howard Oakley. How M1 Macs feel faster than Intel models: it's about QoS. https://eclecticlight.co/2021/05/17/how-m1-macs-feel-faster-than-intel-models-its-about-qos, 2021.

[33] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the Senseless Killing: Improving Memory Management for Mobile Operating Systems. In *2020 USENIX Annual Technical Conference (ATC)*, pages 873–887. USENIX Association, 2020.

[34] 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)*, pages 897–910. USENIX Association, 2020.

[35] Google - Android Developers. App startup time. https://developer.android.com/topic/performance/vitals/launch-time.

[36] Google - Android Developers. Josh sees increased customer retention by improving app startup time by 30%. https://developer.android.com/stories/apps/josh.

[37] Google - Android Developers Blog. Improving App Performance with Baseline Profiles. https://android-developers.googleblog.com/2022/01/improving-app-performance-with-baseline.html, 2022.

[38] Google - Android Developers Blog. Improving app performance with ART optimizing profiles in the cloud. https://android-developers.googleblog.com/2019/04/improving-app-performance-with-art.html, 2019.

[39] Google - Android Developers (Medium). Testing App Startup Performance. https://medium.com/androiddevelopers/testing-app-startup-performance-36169c27ee55, 2020.

[40] Google - Android Developers. Increasing app speed by 30%: a key ingredient in Zomato's growth recipe. https://developer.android.com/stories/apps/zomato.

[41] Google - Android Developers (Medium). Improving app startup with I/O prefetching. https://medium.com/androiddevelopers/improving-app-startup-with-i-o-prefetching-62fbdb9c9020, 2020.

[42] Samsung. Z-SSD. https://semiconductor.samsung.com/ssd/z-ssd.

[43] Forbes. Why Brands Are Fighting Over Milliseconds. https://www.forbes.com/sites/steveolenski/2016/11/10/why-brands-are-fighting-over-milliseconds, 2016.

[44] NVIDIA. Analysing Stutter – Mining More from Percentiles. https://developer.nvidia.com/content/analysing-stutter-%E2%80%93-mining-more-percentiles-0, 2014.

[45] Engadget. Razer Phone hands-on. https://www.engadget.com/2017-11-01-razer-phone-hands-on.html, 2017.

[46] Samsung. Galaxy S20 Display Developers on What Makes the 120Hz Display Special. https://news.samsung.com/global/interview-galaxy-s20-display-developers-on-what-makes-the-120hz-display-special, 2020.

[47] Apple. Apple unveils iPhone 13 Pro and iPhone 13 Pro Max — more pro than ever before. https://www.apple.com/newsroom/2021/09/apple-unveils-iphone-13-pro-and-iphone-13-pro-max-more-pro-than-ever-before, 2021.

[48] ASUS. The ROG Phone 3 turns mobile gaming up to 144Hz. https://rog.asus.com/articles/product-news/the-rog-phone-3-turns-mobile-gaming-up-to-144hz, 2020.

[49] Google - Android Open Source Project. Identifying Jitter-Related Jank. https://source.android.com/docs/core/debug/jank_jitter.

[50] Johannes Weiner. PSI - Pressure Stall Information. https://docs.kernel.org/accounting/psi.html, 2018.

[51] Google - Android Open Source Project. Google Pixel 7 Pro device tree - device-panther.mk. https://android.googlesource.com/device/google/pantah/+/c9139250db92931907cd2bba1b5253846c389711, 2022.

[52] ASUS. ROG Phone 6 Pro - Tech Specs. https://rog.asus.com/phones/rog-phone-6-pro-model/spec, 2022.

[53] Yu Zhao. Multigenerational LRU Framework. https://lore.kernel.org/linux-mm/20220208081902.3550911-1-yuzhao@google.com, 2022.

[54] Google - Android Developers. Memory allocation among processes. https://developer.android.com/topic/performance/memory-management.

[55] Google - Google Play Apps & Games (Medium). Shrinking APKs, growing installs. https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2, 2017.

[56] Google - Android Open Source Project. ProcessList. https://android.googlesource.com/platform/frameworks/base/+/refs/tags/android-12.1.0_r27/services/core/java/com/android/server/am/ProcessList.java#188, 2022.

[57] Google - Android Developers. ActivityManager.AppTask. https://developer.android.com/reference/android/app/ActivityManager.AppTask.

[58] Google - Android Developers. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb.

[59] Daniel Rosenberg. f2fs: checkpoint disabling. https://lore.kernel.org/lkml/20180807234843.129387-1-drosen@google.com, 2018.