USENIX Association

Proceedings of the 20th USENIX Conference on File and Storage Technologies

February 22–24, 2022
Santa Clara, CA, USA

© 2022 by The USENIX Association

All Rights Reserved

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

ISBN 978-1-939133-26-7

Conference Organizers

Program Committee
Abutalib Aghayev, *The Pennsylvania State University*
Samer Al-Kiswany, *University of Waterloo*
Deniz Altınbüken, *Google*
George Amvrosiadis, *Carnegie Mellon University*
Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Nathan Beckmann, *Carnegie Mellon University*
John Bent, *Seagate*
Randal Burns, *Johns Hopkins University*
Ali R. Butt, *Virginia Tech*
Rong Chen, *Shanghai Jiao Tong University*
Young-ri Choi, *UNIST (Ulsan National Institute of Science and Technology)*
Alex Conway, *VMware Research*
Sasha Fedorova, *University of British Columbia*
Dan Feng, *Huazhong University of Science and Technology*
Ashvin Goel, *University of Toronto*
Danny Harnik, *IBM Research*
Dean Hildebrand, *Google*
Jooyoung Hwang, *Samsung*
Sudarsun Kannan, *Rutgers University*
Sanidhya Kashyap, *EPFL*
Youngjin Kwon, *Korea Advanced Institute of Science and Technology (KAIST)*
Patrick P. C. Lee, *The Chinese University of Hong Kong*
Xiaosong Ma, *Qatar Computing Research Institute*
Peter Macko, *NetApp*
Changwoo Min, *Virginia Tech*
Dalit Naor, *The Academic College of Tel Aviv–Yaffo*
Peter Pietzuch, *Imperial College London*
Florentina Popovici, *Google*
Don Porter, *The Unversity of North Carolina at Chapel Hill*
Raju Rangaswami, *Florida International University*
Rob Ross, *Argonne National Laboratory*
Brad Settlemyer, *Los Alamos National Laboratory*
Keith A. Smith, *MongoDB*
Amy Tai, *VMware Research*
Vasily Tarasov, *IBM Research*
Chao Tian, *Texas A&M University*
Haris Volos, *University of Cyprus*
Carl Waldspurger, *Carl Waldspurger Consulting*
Brent Welch, *Google*
Youjip Won, *Korea Advanced Institute of Science and Technology (KAIST)*
Amelie Chi Zhou, *Shenzhen University*
Danyang Zhuo, *Duke University*

Work-in-Progress/Posters Co-Chairs
Deniz Altınbüken, *Google*
Alex Conway, *VMware Research*

Test of Time Awards Committee
Jiri Schindler, *Tranquil Data*
Keith A. Smith, *MongoDB*

Mentoring Chair
Vasily Tarasov, *IBM Research*

Steering Committee
Nitin Agrawal, *ThoughtSpot*
Marcos K. Aguilera, *VMware Research*
Casey Henderson, *USENIX Association*
Kimberly Keeton
Geoff Kuenning, *Harvey Mudd College*
Arif Merchant, *Google*
Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology)*
Raju Rangaswami, *Florida International University*
Erik Riedel
Jiri Schindler, *Tranquil Data*
Bianca Schroeder, *University of Toronto*
Keith A. Smith, *MongoDB*
Eno Thereska, *Amazon*
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*

External Reviewers

Michael Bender
Andreas Dilger
Djordje Jevdjic
Prashant Pandey
Erez Zadok

Message from the FAST '22 Program Co-Chairs

Welcome to the 20th USENIX Conference on File and Storage Technologies (FAST '22). 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. As we continue living and working in unprecedented times, this year's conference is the first hybrid FAST (both in-person and online). This is a welcome return for at least some of the storage community to be able to congregate after last year's first fully virtual conference. We are pleased to include activities that allow wider participation than a strictly in-person event, as well as accommodate "mask-to-mask" interaction for those who can make the trip. We have a program with talks and posters on a wide range of topics, including emerging and traditional storage technologies; distributed storage; key-value stores and graph analytics; deduplication; performance analysis; and, as always, new file system designs.

To commemorate the 20th FAST conference, we will have several small activities throughout the conference, including trivia and memories from early FAST conferences and an opportunity to make predictions for the future of storage. We will share some predictions and memories throughout the conference, and put them in a time capsule for a future FAST conference where the capsule will be opened to see our community's ability to predict the future. We hope these activities will be fun, informative, and a good chance to think about where we want our community to grow and evolve over the next two decades.

FAST '22 received 130 submissions from authors in academia, industry, government labs, and the open-source communities. Of these, we accepted 28 papers, for an acceptance rate of 21%. 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, 72 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 second year that FAST has included an author response period. After the author response period and online discussion, the PC discussed 51 papers to select the final program. The two-day hybrid PC meeting was held on December 6-7, 2021 and had six PC members attend in-person in Chapel Hill, NC, with the rest joining virtually from global locations that spanned 10 time-zones. We used Eddie Kohler's excellent HotCRP service to manage all 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 three deployed-systems submissions and accepted one.

This year we introduced a new mentoring program, which was spearheaded by Vasily Tarasov. Joining a new community can be daunting. The goal of the mentorship program is to match FAST community newcomers with more seasoned participants to nurture a sense of belonging and remove barriers by introducing them to others, answering questions, offering advice on how to get the most out of the experience, and even just meeting up for a chat. We hope the FAST community benefited from the program and would like to thank all mentors, mentees, and particularly Vasily for leading this valuable program.

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 '22. We would also like to thank the attendees of FAST '22 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 actually happen. We would like to thank the Work-in-Progress Session Chairs, Alex Conway and Deniz Altinbüken and our delegates to the *;login:* editorial board, Sasha Fedorova and Xiaosong Ma. Our thanks go also to the members of the FAST Steering Committee who provided invaluable advice and feedback, and to our Steering Committee Liaison, Keith Smith, for his guidance and encouragement on many issues, large and small, 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. In total, the PC wrote 529 thoughtful reviews and 1282 online comments. HotCRP recorded approximately 344,209 words in reviews (excluding HotCRP boilerplate language)—the same rough length as one book in George R.R. Martin's *Song of Ice and Fire* series (a.k.a. *Game of Thrones*). 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!

Dean Hildebrand, *Google*
Don Porter, *University of North Carolina*
FAST '22 Program Co-Chairs

20th USENIX Conference on File and Storage Technologies (FAST '22)

February 22–24, 2022

Santa Clara, CA, USA

Tuesday, February 22

Persistent Memory: Making it Stick

A Series of Merges

Solidifying the State of SSDs

Wednesday, February 23

Distant Memories of Efficient Transactions

The Five Ws of Deduplication

Meet the 2022 File System Model-Year Lineup

Thursday, February 24

Keys to the Graph Kingdom

Keeping the Fast in FAST

NyxCache: Flexible and Efficient Multi-tenant Persistent Memory Caching

Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen[†], Kwanghyun Park[†],
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin–Madison [†]Microsoft

Abstract. We present NyxCache (Nyx), an access regulation framework for multi-tenant persistent memory (PM) caching that supports light-weight access regulation, per-cache resource usage estimation and inter-cache interference analysis. With these mechanisms and existing admission control and capacity allocation logic, we build important sharing policies such as resource-limiting, QoS-awareness, fair slowdown, and proportional sharing: Nyx resource-limiting can accurately limit PM usage of each cache, providing up to 5× better performance isolation than a bandwidth-limiting method. Nyx QoS can provide QoS guarantees to latency-critical caches while providing higher throughput (up to 6× vs. previous DRAM-based approaches) to best-effort caches that are not interfering. Finally, we show that Nyx is useful for realistic workloads, isolating write spikes, and ensuring that important caches are not slowed down by increased best-effort traffic.

1 Introduction

Memory-based look-aside key-value caches (e.g., memcached [14]) are a critical component of many systems and applications [3, 5, 23, 74]. To improve utilization and simplify management, multiple cache instances are often consolidated onto a single multi-tenant server. For example, Facebook [54] and Twitter [74] each maintain hundreds of dedicated cache servers that host thousands of cache instances. However, multi-tenant servers have the added challenge of ensuring that each client cache meets its performance goals; a range of production and research in-memory multi-tenant caches currently provide different sharing policies, such as enforcing a limit on the used memory capacity and bandwidth [7], guaranteeing a level of quality-of-service (QoS) [18], and allocating resources proportionately [60].

Persistent memory (PM), such as that provided by Intel's Optane DC PMM [10], is emerging as an appealing building block for these caches, due to PM's large capacity, low cost per byte, and comparable performance to DRAM. However, PM performance differs from DRAM and Flash in a number of ways that reduce the effectiveness of current multi-tenant caches for other devices [34, 62]. In particular, unlike DRAM, Optane DC PMM exhibits highly asymmetric read vs. write performance (for a single DC PMM, max read bandwidth is 6.6GB/s whereas max write bandwidth is 2.3GB/s) [45], severe and unfair interference between reads and writes (writing at 1GB/s can cause the same throughput and P99 latency slowdown to a co-running read workload as reading at 8GB/s) [55], and especially efficient access for multiples of 256B [73].

Unfortunately, existing multi-tenant DRAM and storage caching techniques do not readily translate to PM. Some approaches focus exclusively on capacity allocation across clients [34, 60, 62]; capacity allocation is necessary but not sufficient for PM sharing because the rate of requests to PM must also be regulated. Host-level request regulation has been explored extensively for Flash devices using block-layer I/O scheduling [58, 61], but these software overheads are prohibitive given 100ns PM accesses [24]. Device-level request scheduling assumes special hardware that PM lacks [53, 65, 78, 79]. Finally, coarse-grain request throttling underpins the vast majority of DRAM bandwidth allocation techniques; however, these approaches assume both hardware counters and performance characteristics that do not hold for PM (e.g., bandwidth is an accurate estimate of utilization).

In this paper, we introduce NyxCache (Nyx), a standalone lightweight and flexible PM access regulation framework for multi-tenant key-value caches that is optimized for today's PM without special hardware support. Given a PM server and a sharing policy (e.g., QoS), cache instances are admitted and assigned space using existing load admission [36, 37, 52] and capacity allocation [34, 60, 62] techniques. At runtime, Nyx monitors information (e.g., PM resource utilization) of caches, regulates the rate at which each cache is allowed to access PM, and thus enforces the sharing policy's performance goals. Nyx works with any in-memory key-value store that adheres to the memcached interface [14]; the current implementation includes a PM-optimized version of Twitter's Pelikan [17] that can improve single-cache performance by more than 50% for get-heavy workloads and 3× for write-heavy workloads. Nyx's central contribution is a set of software mechanisms designed for PM to extract the information required to flexibly enforce popular sharing policies.

Nyx provides new mechanisms to efficiently i) regulate PM accesses, ii) obtain a client's PM resource usage, iii) analyze inter-client interferences, and has two particularly useful and novel mechanisms for PM. First, Nyx efficiently estimates not only the total PM DIMM utilization (building on pioneering work in this space [55]), but also the PM utilization caused by each cache instance, as is needed for sharing policies; estimating PM utilization is challenging because the number of transferred bytes is not an accurate proxy of PM utilization, unlike on DRAM. Second, Nyx can determine which cache instance most interferes with another cache instance; in PM-based systems, these interactions are difficult to identify because a harmed client may be impacted more by a low-bandwidth client than a high-bandwidth client, unlike DRAM. Both of these mechanisms accurately account for the

	Resource Limit	Quality of Service	Fair Slowdown	Proportional Resource Allocation
Request Regulation	✓	✓	✓	✓
Resource Usage	✓			✓
Interference		✓		*
Application Slowdown			✓	✓

Table 1: **Control and Information Needed.** ✓*indicates control or information is required by the policy. * indicates optional.*

CPU cache prefetching that is essential for high performance on PM. These new mechanisms enable Nyx to easily and efficiently support sharing policies such as resource limiting, QoS, fair slowdown, and proportional sharing.

The sharing policies provided by Nyx are powerful. Nyx can accurately limit the PM utilization of each cache (similar to Google Cloud's memcache [7]), whereas an approach that measures only bandwidth cannot. Nyx can provide QoS guarantees to latency-critical caches while providing higher throughput (up to 6×) to best-effort caches that are not interfering. Nyx can provide proportional resource allocation while redistributing idle PM utilization to clients that will not inadvertently slowdown others. Finally, as shown for real large-scale cache traces from Twitter, Nyx can isolate clients from write spikes and ensure that important caches are not slowed down by increased best-effort traffic.

In the rest of this paper, we evaluate previous multi-tenant caches and their limits for PM (§2); discuss the Nyx design (§3); evaluate overheads of Nyx's mechanisms and the effectiveness of its policies (§4); discuss potential extensions (§5); compare to related work (§6); and conclude (§7).

2 Motivation and Challenges

We provide background on the sharing policies provided by many in-memory multi-tenant key-value caches and the mechanisms needed to implement those policies. We explain why previous approach for providing control and information on DRAM or block I/O do not work well on PM.

2.1 Sharing Policies for Multi-Tenant Caches

In-memory key-value caches such as memcached [14], Redis [66], and Pelikan [17] are an essential part of web infrastructure for many real-time and batch applications [3, 74]. Before accessing data from slow backend-storage or compute nodes, applications first check an in-memory cache server. In production environments, cache servers are usually multi-tenant: many cache instances are consolidated on a single server to improve utilization and simplify management and scaling [54]. In a multi-tenant cache, requests are routed to the cache instance of the corresponding tenant. For example, large companies such as Facebook [54] and Twitter [74] maintain hundreds of large-memory dedicated servers that host thousands of cache instances. Smaller companies use caching-as-a-service providers such as ElastiCache [1], Redis [20] and Memcachier [16]. In this paper, we focus on managing an individual multi-tenant cache server.

Giving competing clients, enforcing performance and sharing goals is critical in multi-tenant caching. Different industrial and research multi-tenant systems have provided different objectives; we focus on the following four.

Resource Limiting: A common objective when clients pay for resources is to guarantee that each client cannot exceed some amount of usage such as bandwidth, ops/sec, or number of resources [2, 7]. For example, Google Cloud memcache limits operations according to a pricing tier, such as "Up to 10k reads or 5k writes (exclusive) per sec per GB" [7]. Multiple resources can be limited simultaneously, e.g., Amazon ElastiCache [2] charges for both memory and vCPUs.

There are two requirements for a multi-tenant cache to enforce per-client resource limits. First, the system must accurately determine the amount of resource each client is using; we refer to this as *resource usage estimation*. Second, the system must reschedule or throttle requests of each client if they exceed this limit, which we call *request regulation*. Below (§2.2), we describe how previous multi-tenant caches have provided request regulation and resource usage estimation, and why these previous approaches are not sufficient for PM.

Quality-of-Service: A multi-tenant system may ensure that each client's performance goals (throughput, latency, or tail latency) are met regardless of other co-located clients, as in Twitter [18] and Microsoft [62]. This objective is useful for latency-critical clients that must meet service-level objectives (SLOs). For example, production caches at Twitter provide a p999 latency of <5 milliseconds [18].

Providing QoS requires knowledge of whether each client is meeting its goals at run-time. When the system observes that one client is not meeting its performance guarantee, interfering clients are identified and limited [31, 39, 51] (e.g., with *request regulation*). Identifying the client causing the most harm is usually straightforward and based on simple bandwidth [39] for DRAM-based caches, but not for PM. A new technique involving *interference estimation* is required on PM to determine how the workloads compose.

In addition to run-time support, guaranteeing QoS requires admission control and space allocation. Admission control must be performed on newly arriving clients to ensure that the system has sufficient resources and that the new client will not interfere with existing clients [36, 37, 52]. Space allocation across cache instances must be performed to provide a specified hit ratio for each client to ensure each can meet its goals. Previous research has focused on this challenge. For example, Microsoft [62] allocates space to meet QoS bandwidth targets, and Robinhood [29] to minimize tail latency. Admission control and space allocation are mostly orthogonal to the new challenges introduced by PM and are not our focus.

Fair Slowdown: Multi-tenant systems in more cooperative environments may ensure that all clients are slowed down by the same amount. Formally, these approaches minimize the ratio of the maximum slowdown to the minimum slow-

down [38, 63]. In web cache settings, application requests may fan out, in which case the cache access with the longest latency determines overall latency [29, 54]; thus, balancing slowdown benefits overall request latency.

Enforcing fair slowdown requires knowledge of each cache's slowdown at runtime. The system must monitor each cache's current performance when sharing the server with others and know its performance if run alone. A technique for *slowdown estimation* is required. Furthermore, to equalize slowdowns of different caches, caches with small slowdown should be further limited and caches with larger slowdowns should be less limited (e.g., with *request regulation*).

Proportional Resource Allocation: Finally, a multi-tenant system may incent clients to share resources by guaranteeing that each of N clients performs within $1/N$-th of its stand-alone performance. This guarantee can be generalized to give each client a different proportional share. Idle resources may be redistributed across clients, such that some obtain more than their guarantee. For example, FairRide [60] ensures proportional cache space allocation.

To guarantee proportional allocation, a multi-tenant cache must meet three requirements. The system must perform *request regulation* and *resource usage estimation* to guarantee that each client does not consume more than its allocation. When assigning idle resources to clients, the system must validate that the additional resource usage does not interfere with others; therefore, the system must track each client's slowdown (i.e., with *slowdown estimation*) and stop idle resource re-allocation before it severely impacts some clients.

In summary, for a multi-tenant cache to provide the above policies, it must control resource usage of each cache instance and obtain information about resources and application performance. Table 1 summarizes the needed control and information for each policy.

2.2 Challenges of PM Cache Sharing

Persistent memory is an appealing building block for key-value caches. After presenting PM background, we describe the challenges of using PM for multi-tenant caching.

2.2.1 Persistent Memory Characteristics

PM is becoming a reality in products and research prototypes. For example, Intel Optane DC PMM [10] is a popularly available device; there are also research prototypes [30,49,70]. In this paper, we use PM to refer to Optane DC PMM. PM performance is similar to DRAM but can deliver extremely large capacity at low cost [10, 11]. PM is significantly faster than NAND Flash and is byte-addressable. PM is directly connected to the memory bus and, when configured in App Direct Mode, can be accessed using loads and stores. Different CPU caching options exist for PM access: loads and stores with CPU caching and prefetching; loads and stores with prefetching disabled (for both PM and DRAM); non-temporal (NT) operations that bypass the CPU cache entirely [73].

Table 2 summarizes the bandwidth and latency of Optane DC PMM for a workload relevant to key-value caches: random 256B and 4KB loads and stores. As shown, for loads, regular loads perform best: CPU cache prefetching is essential for hiding PM latency and increasing throughput. For stores on a random workload, NT-stores that bypass the CPU cache have much better performance. Thus, we use in-PM key-value caches optimized to use regular loads and NT-stores.

	Metric	Load	No-Prefetch	NT-Load	Store	Store+clwb	NT-Store
256B	GB/s	1.59	1.53	0.29	1.12	0.52	3.73
	us	0.49	0.52	0.84	0.38	0.47	0.08
4KB	GB/s	4.08	2.92	2.24	1.03	1.50	3.44
	us	1.22	1.69	1.84	4.14	2.71	1.22

Table 2: **PM Load/Store Performance.** *This table summarizes the throughput/latency of single thread random 256B and 4KB load/store operations (on 2× DC PMMs). No-Prefetch: the CPU's prefetching is turned off (for DRAM/PM); NT: non-temporal operations that bypass the CPU cache.*

PM has unique characteristics that impact multi-tenant caching. For instance, as previously identified, PM exhibits asymmetric read vs. write performance [45], especially efficient access for specific sizes (e.g., 256B) [73], and severe and unfair interference across reads and writes [55]. As we will describe, these characteristics deeply impact the ability to perform request regulation and to estimate resource usage, interference, and application slowdown.

2.2.2 Request Regulation

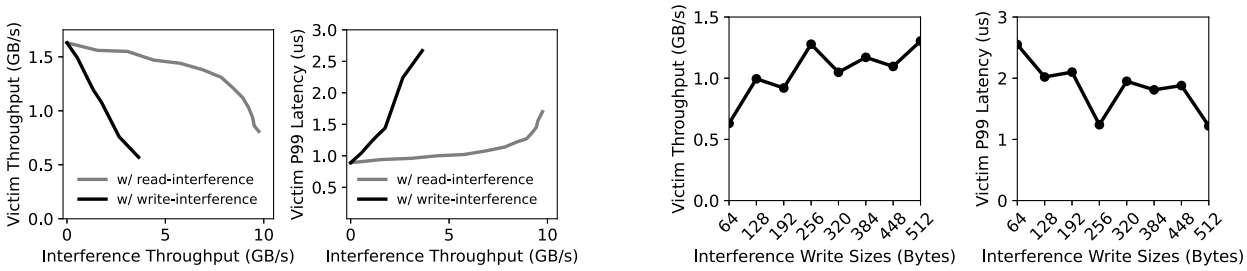
Previous approaches for request regulation have been designed for both DRAM and for block I/O. However, none of these approaches are suitable for PM.

Existing techniques for regulating memory requests have adjusted the number of cores dedicated to an application [39], used clock modulation (DVFS) [57], and Intel Memory Bandwidth Allocation (MBA) [9]. In multi-tenant caching, reducing the number of cores is not suitable because a cache instance is often allotted only a single core [2]. Intel MBA manages last-level cache (LLC) misses from each core to limit memory traffic, but does not distinguish between misses to PM and DRAM [8] and so cannot restrict PM accesses without also slowing down DRAM. Furthermore, Intel MBA does not have access to accurate information about resource usage, interference, and application slowdown, as we will discuss. Likewise, adjusting CPU frequency has an effect on all instructions; Oh et al. [55] demonstrated the ineffectiveness of CPU frequency scaling on regulating PM traffic.

I/O requests have been regulated via software with block-layer I/O scheduling [12], which is not suitable for PM for two reasons. First, the block abstraction would add significant read/write amplification for byte-addressable PM. Second, scheduling requests with merging, reordering, and other synchronization would add unacceptable overhead to otherwise low-latency PM accesses [24].

2.2.3 Resource Usage Estimation

Previous techniques for estimating the memory or I/O usage of clients do not work well for PM. We describe the problems with previous software approaches for tracking I/O



(a) Read vs. Write Interferences

(b) Interferences Related to Access Sizes

Figure 1: **PM Load Performance with Various Interferences.** *We place a victim workload (single thread 256B loads) with various interferences. (a) shows the victim throughput and tail latency when colocated with varying amounts of read and write interferences. (b) shows the victim performance when colocated with 1GB/s store traffic of varying access sizes (range from 64B to 512B with step of 64B).*

usage and with hardware approaches for DRAM.

As discussed above, CPU cache prefetching is required for PM to deliver high bandwidth and low latency. However, when estimating block I/O traffic in software [4, 35, 76], extra PM accesses caused by prefetching are not observed. Running an experiment with 1KB random loads, we found that software-level tracking accounted for only 60% of actual memory traffic, leading to inaccurate resource-usage estimation.

Accounting on DRAM uses hardware counters to track L3 cache line misses to the memory controller per core. While hardware counters accurately measure prefetching, they do not account for the difference between cache line size and PM access granularity, which is needed for PM accounting. Because PM has a 256B minimum access granularity, a 64B load (a single L3 cache line) utilizes the same amount of PM resources as a 256B load (four L3 cache lines). Thus, four cache line accesses can result one to four PMEM accesses. Previous systems for resource estimation have often used bandwidth consumption as a proxy for resource usage [39, 51, 77], but this is not appropriate for PM where operation cost is affected by access size and is different for reads versus writes.

Unfortunately, current hardware counters in PM are also not sufficient; existing PM counters are at the DIMM media-level and do not track per-client or per-core usage [13, 55].

2.2.4 Interference Estimation

In memory-based approaches, interference caused by a particular client was assumed to be related to memory bandwidth. For example, Caladan [39] identifies the client with the highest number of LLC misses, which corresponds directly to the client with the highest memory bandwidth. This simplification does not work for PM, as PM interference depends on both volume and pattern of traffic.

Specifically, on PM, write-intensive clients generate greater interference than read-intensive clients with the same bandwidth, as shown in Figure 1.a. For example, on a read-intensive client, a competing 1GB/s write causes the same throughput and tail latency interference as a competing 8GB/s read. As shown in Figure 1.b, smaller accesses (64B) can cause more interference than larger accesses (256B). Since PM has a minimum granularity of 256B, a 64B access is amplified into 256B on the device; thus, at the same bandwidth, 64B accesses generate significantly more interference than 256B accesses. In short, the bandwidth of a competing client is not a good estimation of interference in PM, unlike DRAM.

2.2.5 Application Slowdown Estimation

Numerous efforts have estimated slowdown for DRAM and Flash-based systems; however, all require specialized device support. For example, FST [38] requires in-DRAM bank conflict counters that are updated with each memory access; MISE [64] and ASM [63] require the DRAM controller to assign priorities to application requests. FLIN [65] changes the Flash controller to track and rearrange each flash transaction. Although application slowdown is not inherently different on PM than DRAM or I/O, previous approaches require special hardware which is not available on PM.

Summary: Multi-tenant PM caching demands new methods for regulating PM accesses and extracting PM resource usage, interference information, and application slowdown.

3 NyxCache Design

Given that existing multi-tenant cache servers cannot handle PM, we introduce NyxCache (Nyx). Nyx provides mechanisms for control (e.g., request throttling) and information estimation on PM (e.g., resource usage, interference, and application slowdown), and supports a range of sharing policies (e.g., resource limiting, quality-of-service, fair slowdown, and proportional resource usage). We describe the overall architecture of Nyx, present our design goals, describe how Nyx provides these mechanisms and policies.

3.1 Architecture

As shown in Figure 2, Nyx provides a multi-tenant in-PM caching framework. Each PM server running Nyx may contain any number of cache instances (e.g., memcached, Pelikan, Redis). Thousands of users may send requests (e.g., set/get) to their associated cache instance. When cache space is exhausted, a cache instance can use any eviction strategy (e.g., FIFO, LRU, and LFU). As in other look-aside caches, users explicitly write desired data into the cache; Nyx does not fetch data from remote storage on a cache miss.

Nyx can be configured with different sharing policies and parameters (e.g., a resource limit, latency target, or propor-

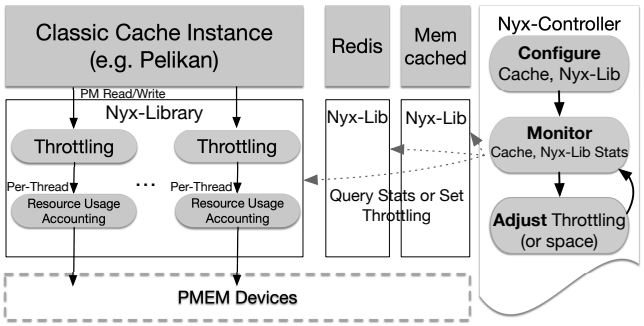


Figure 2: **NyxCache Architecture.** *Nyx implements throttling and resource usage accounting for each cache instance, and enforces sharing policies across cache instances. Nyx contains two major components: 1) a Nyx Library for each instance, and 2) a centralized Nyx Controller.*

tional weight). Administrators can implement new policies using the control and information mechanisms provided by Nyx. At runtime, Nyx enforces the desired sharing policy. Based on information Nyx acquires about per-instance resource usage and performance, the Nyx controller dynamically adjusts the throttling and space allocated to instances.

Nyx has two requirements for cache instances. First, each cache instance must report application-level performance metrics such as throughput and tail latency; most systems have this capability or can be extended [15]. Second, the instances must be integrated with a trusted Nyx-library. When a cache instance reads/writes from/to PM, it must use Nyx library APIs (e.g., read(dest, src), write(dest, src)). For each PM access, the Nyx library throttles access, tracks PM usage, and performs the actual access. The library uses a separate thread to communicate with the Nyx controller. The controller interacts with the library to query statistics and to set configuration, space, and throttling values. Nyx leverages techniques from previous multi-tenant in-memory caches for basic sharing functionality such as admission control and space allocation. As of now, Nyx only manages cache instances on a single NUMA node that share PM (and all PM accesses are local); multiple Nyx can be used to manage multiple NUMA nodes. We leave NUMA-aware management for future work.

3.2 Design goals

Nyx has the following goals. (i) **Lightweight:** Performance is critical for in-PM caching; thus the cost of adding control and acquiring information must be low relative to the cost of accessing PM. (ii) **Flexible Sharing Policies:** Different sharing policies may be required by administrators for different scenarios. Thus, Nyx can be configured with several policies based on a common set of simple mechanisms. (iii) **No Special Hardware:** Previous work has assumed smart resources (e.g. Flash, DRAM) that provide configurable control and information [53, 65, 78, 79]. Nyx handles current devices with existing hardware interfaces. (iv) **Minimal Assumptions:** Storage devices are continuously evolving, with new generations having new performance characteristics. Therefore, Nyx does not assume a particular performance model for all PM devices (e.g., the interference for different operations).

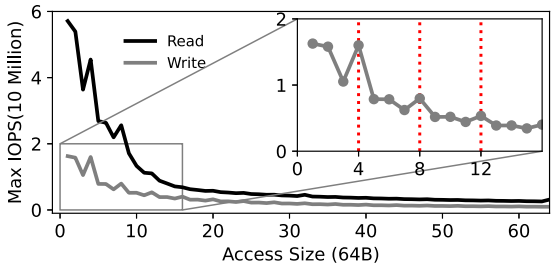


Figure 3: **MaxIOPS Profile.** *MaxIOPS for random reads and writes of different sizes on our 2× Intel Optane DC PMM system.*

3.3 Nyx Mechanisms

Nyx contains low-level mechanisms that enable higher-level sharing policies to be implemented easily. Since request regulation, estimation of resource usage, interference, and application slowdown are changed significantly by PM, we describe these Nyx mechanisms in detail. Access control and space allocation are largely independent of PM and not the focus of this paper; Nyx borrows these techniques from previous systems [29, 34, 52, 60, 62].

PM Access Regulation: To minimize the overhead of regulating requests to PM, Nyx adheres to the basic principle used by previous techniques for DRAM regulation: throttle requests in a coarse-grained manner without reordering or prioritizing. To mimic the behavior of Intel MBA, Nyx implements simple throttling by delaying PM accesses at user-level.

Our current implementation adds delays in units of 10ns with a simple computation-based busy loop. In some cases PM operations may need to be delayed indefinitely (e.g., when a resource limit is reached); in this case, PM operations are stalled until the Nyx controller sets the delay to a finite value.

Resource Usage Estimation: Nyx must determine how much PM resource each cache instance is using. As described in Section 2, for PM the number of transferred bytes is not a good estimate of resource usage; on PM, each operation type (e.g., read or write) and access pattern (e.g., request size) consumes a different amount of the resource and has a different maximum operations per second. Therefore, Nyx determines the utilization of PM as a function of the current IOPS of each operation type relative to the maximum IOPS for that operation type. For example, if the maximum IOPS of pattern A is $MaxIOPS_A$, then the cost of each operation of pattern A is $1/MaxIOPS_A$. If the maximum IOPS of pattern B is $1/N \times MaxIOPS_A$, then each B operation consumes N times more PM than an A operation and has N times the cost. The IOPS cost model accurately captures that writes are more expensive than reads, and the dependency on request size.

Nyx determines the MaxIOPS of each access pattern through profiling, performed once per PM server. The profiler measures IOPS for random read and write operations between 64B and 4KB (in steps of 64B). Because prefetching occurs during profiling, the measured MaxIOPS accurately represents the cost of both the operation itself and any wasted prefetching. Profiling concentrates on random accesses as

multi-tenant key-value caches are mostly random: first, because multiple tenants access PM simultaneously (in different address spaces), their requests are interleaved; second, keys tend to be mapped to arbitrary PM locations based on their time-to-live and size [14, 75]. The profiler stops at request sizes of 4KB which obtain the device's maximum bandwidth.

Figure 3 shows the profiled MaxIOPS for reads and writes as a function of request size. As shown, writes have lower IOPS and thus a higher cost per operation than reads. While larger requests generally have lower IOPS, there is a complex relationship with the minimal PM access size: for example, a 64B random store has a similar maximum IOPS as 256B, the minimum PM access size; accesses that are not aligned to 256B have lower MaxIOPS.

At runtime, Nyx tracks the PM usage of each cache instance. When a cache instance accesses PM, Nyx looks up the MaxIOPS for this operation and size, and increments a cost counter for this cache instance by $\frac{1}{MaxIOPS}$. To reduce synchronization overhead, these counters are maintained per-thread and only lazily combined when needed (e.g., for responding to a resource usage query from Nyx Controller).

While the CPU cache can theoretically introduce errors in PM cost estimation, these errors are negligible for Nyx. First, since CPU prefetching waste depends in part on spatial locality, the profiler mimics the random accesses of cache instances that have little sequentiality. Second, given a cache instance that uses NT-store (as in Nyx-Pelikan), the CPU cache has no effect on stores. Finally, although a PM load could be served in the CPU cache and never access PM, in multi-tenant caches few PM loads hit in the CPU cache: because each instance's working set is typically tens of GBs [28, 74] (and there are many instances), there is little temporal locality in CPU caches of tens of MBs. More intricate cost models for cache instances with spatial (e.g., scan) and temporal locality (e.g., bursty retries) are left for future work.

Interference Analysis: When multiple cache instances are co-located, Nyx determines which instance most impacts another. For example, when an efficient QoS implementation observes that an affected client W is not meeting its guarantee, it will iteratively slow down the one competing client that will produce the greatest benefit for W. In PM-based systems, unlike DRAM, these interactions are difficult to identify because an affected client may be impacted more by a low-bandwidth client than a high-bandwidth client. The amount of interference is due to complex scheduling within the PM device; as future generations of PM devices become available, which clients interfere with which others may change. Therefore, Nyx assumes no prior knowledge of these interactions.

Nyx determines which client is interfering the most with the affected client with a runtime micro-experiment. Given affected client W and several competing clients, Nyx iteratively throttles each competing client by X for some metric of interest while measuring the impact on client W. The throttled client that helps W attain the greatest performance improvement is identified as the client that interferes with W the most. The value of X is configurable, as is the metric (e.g., throughput, average latency, or tail latency). Nyx uses simple pruning techniques to throttle only the clients with the highest resource usage. Optimizations for reducing micro-experiment times (e.g., focus on different client subsets in different trials) are left for future work.

Algorithm 1: Resource Limit The gray area denotes unique functionality used to deal with PM issues

```
EpochLen: ticks in an epoch (e.g. 100), TickLen: (e.g. 10ms)
A.getResCounter(): query A's Nyx-Lib for resource usage
A.setThrottling(t): add t×10ns delay to each access of A
ResAssigned[1..N]: each cache's assigned resource per epoch
while true do
    # Step 1: Begin an epoch and set all cache throttling to 0
    foreach cache A do
        A.setThrottling(0)
        InitResCounter[A] = A.getResCounter()
    # Step 2: Monitor resource utilization and pause clients
      who have used up their allotted resources.
    while Epoch is not completed do
        SleepFor(TickLen)
        foreach cache A do
            ResUsed = A.getResCounter() - InitResCounter[A]
            if ResUsed > ResAssigned[A] then
                A.setThrottling(INFINITE) # Pause
```

SlowDown Estimation: Nyx determines the slowdown that each client experiences at runtime by calculating $\frac{T_{alone}}{T_{share}}$; T_{alone} is the client's performance (for some metric of interest) when it is running alone, and T_{share} is its current performance in the shared environment. As we assume no special hardware, Nyx uses an approach similar to previous work [47].

First, to learn T_{alone}, Nyx briefly pauses all other clients; T_{alone} is updated on a regular basis (e.g., 1s) or whenever a workload change is observed. Second, slowdown is periodically calculated using a runtime measurement of T_{share}. As we will show, at the cost of a small loss of bandwidth and increase in tail latency, this solution adequately approximates slowdown without hardware support. The impact of the pause can be reduced for workloads that do not change frequently.

3.4 Nyx Sharing Policies

Nyx implements four popular sharing policies. We describe how these policies leverage the mechanisms of Nyx for PM.

Resource Limit: Nyx can limit the amount of the PM resource used by each client in multi-tenant caching, isolating the performance of clients from one another. Our policy defines resource limits in terms of standard operations, similar to Google Cloud's memcache [7] (e.g., 1000 1KB random reads per second, or 1MB/s random reads).

As shown in Algorithm 1, Nyx provides resource limits for each client epoch by epoch, extending existing approaches [77]. Each epoch, Nyx monitors the resource utilization of each client; if a client reaches its limit for this epoch, its accesses to PM are delayed until the next epoch. When the epoch ends, the throttling value for each client is reset to zero.

Algorithm 2: QoS The gray area denotes functionality for PM. We omit code to rollback throttling when the action violates any LC task's target.

```
ExperimentStep: a cache's throughput expense pays for an
  interference analysis experiment. (e.g. 500MB/s)
while true do
    # Step 1: Monitor each client's SLO slack
    foreach cache A do
        slack[A] = (A.target - A.latency) / A.target
    S = cache with the smallest slack
    # Step 2: Protect clients violating SLO
    if slack[S] < 0 then
        if S is throttled then
            throttle down S
        else
            # Step 2.1: Pick candidates to throttle
            if there are BE caches then
                candidates = top 3 resource usage BE
            else
                candidates = top 3 res usage LC, slack > 0.2
                if all LCs have little slack then
                    candidates = LC with the most slack
            # Step 2.2: Find the most interfering client
            I = getLargestInterference(S, candidates)
            throttle up I
    else if slack[S] > 0.2 then
        # All caches have slack -> relax throttling
        throttle down every cache

Function getLargestInterference(S, Candidates):
    # Find the tenant who will most improve S at the same
      expense (throughput)
    If there is only one client in Candidates, return the client
    foreach C in Candidates do
        throttle up C by ExperimentStep
        track S latency change after the experiment
        restore all throttle to previous state
    return L who helps S get the largest improvement
```

The implementation allows the administrator to configure the *epoch* and *tick* length to trade-off the overhead of checking counters with reaction time.

Quality-of-Service: Nyx can ensure that latency-critical (LC) tenants meet a service-level-objective while maintaining high PM utilization for best-effort (BE) tenants on the same server. As in earlier work [36, 37], admission control prevents workloads with unachievable QoS targets and space-allocation provides the necessary hit ratio.

As shown in Algorithm 2, Nyx employs an approach similar to Parties [31] and Caladan [39]: for each LC client, the difference between the guaranteed and the current performance is tracked; when the guarantee is violated (i.e., negative slack), a competing tenant is throttled.

Nyx differs in how it identifies the client to be throttled. Caladan always throttles the BE tenant with the maximum bandwidth (LLC misses), whereas Nyx throttles the BE or LC cache that most improves the LC cache, for the same expense across competing tenants. The implementation allows the administrator to configure *ExperimentStep*, allowing a balance between aggressive throttling and faster convergence.

Fair Slow Down: Nyx can achieve fairness in terms of

Algorithm 3: Fair Slow Down

```
A.getSlowDown(): return A's current performance / T_alone
while true do
    if T_alone info is older than P sec then
        foreach cache A do
            refreshTalone(A)
    # Adjust throttling to equalize slowdowns
    foreach cache A do
        SlowDown[A] = A.getSlowDown()
    find cache L and S with the largest and smallest slowdowns
    unfairness = SlowDown[L] / SlowDown[S]
    if unfairness > UnfairnessThreshold then
        throttle down L and throttle up S
        FairIntervals = 0
    else
        # With fair slowdown, try to improve utilization
        FairIntervals ++
        if FairIntervals > FairIntervalThreshold then
            throttle down all caches

Function refreshTalone(A):
    A.setThrottling(0), and pause every other cache
    A.T_alone = measure A throughput
    restore throttle of all caches to previous state
```

equalized slowdown across caches. As in Algorithm 3 [38,63], Nyx minimizes (MaxSlowDown/MinSlowdown) by gradually increasing the throttling of the MinSlowDown cache and decreasing the throttling of the MaxSlowDown cache. The tuning process is terminated when the unfairness metric falls under an UnfairnessThreshold. The implementation periodically (every P seconds) refreshes the estimate of the stand-alone performance (T_{alone}) for each client. Administrators can customize P to balance between lower overhead and faster adjustments for dynamic workloads.

The policy can be generalized to guarantee weighted slowdowns and a hard limit on some cache's slowdown. For the hard limit, Nyx tracks the particular slowdown at runtime and throttles other caches when the hard limit is exceeded.

Proportional Resource Allocation: Nyx implements proportional sharing with actual proportional resource allocation (instead of simple bandwidth allocation) and with interference-aware idle resource redistribution. Nyx ensures that each cache achieves performance equal to or better than accessing PM alone for a given amount of time (time-sharing [67]). For example, if a cache has a weight of 2 out of 3, then it is guaranteed to obtain at least 2/3 of its stand-alone performance.

Nyx first allocates resources (not bandwidth) proportionally to each cache and enforces the resource limit during an epoch (Algorithm 4). We assume cache space has been allocated proportionately. Following an epoch, Nyx forecasts each tenant's desired amount of resources: a tenant that did not use all its given resource may donate idle resources, whereas a tenant that used all assigned resources may consume more (a simple linear model predicts desired resources [77]).

Nyx provides interference-aware resource donation (Option 2 in the Alg.). On PM, idle resource redistribution faces the difficulty that the donated resource may severely interfere with the original donor's performance. For example, as shown

Algorithm 4: Proportional Resource Allocation The slowdown refreshing code is omitted.

```
DonateStep: step to donate idle resources (e.g. 10%)
TotalResource = 1
while true do
    # Step 1: Enforce and track resource usage in an epoch
    Begin a New Epoch
    foreach cache A do
        Enforce A uses resource <= ResourceAssigned[A]
        if A depleted resources, record how long:
          TimeUseUp[A] (e.g. half of the epoch)
        if A left idle resources, record ResourceUsed[A]
    End of the Epoch
    # Step 2: Redistribute Idle resources
    foreach cache A do
        if A has idle resources then
            # Option 1: Donate all extra resources
            DesiredResource[A] = ResourceUsed[A]
            # Option 2: Interference-aware resource donation
            if A.getSlowdown() < TotalWeight / A.weight then
                # Donate a step when within slowdown limit
                DesiredResource[A] =
                Max(ResourceAssigned[A] * (1 - DonateStep),
                ResourceUsed[A])
            else
                # Revoke a step when under slowdown limit
                DesiredResource[A] =
                  Min(ResourceAssigned[A] * (1 +
                  DonateStep), TotalResource * A.weight/
                  TotalWeight)
        if A depleted resources: DesiredResource[A] =
        ResourceAssigned[A] / TimeUseUp[A]
    ResourceAssigned[1..N] = Allocate resources
      proportionally based on weight and desired resource
```

in Section 4.5, if a get-heavy cache A donates idle resources to a write-heavy cache, the new write traffic can dramatically harm A's performance. To prevent this interference, Nyx reallocates resources in increments, stopping when the donating cache's slowdown is near its lower bound; if the slowdown exceeds the lower bound, a portion of the donated resources are returned. Thus, Nyx guarantees the "time-sharing" lower bound while maximizing resource utilization. The implementation allows the administrator to set *DonateStep*, balancing quick idle resource donation and the proportional guarantee.

With Admission Control and Capacity Allocation: In a nutshell, cache instances are 1) admitted, 2) allocated space, and 3) governed by Nyx. A PM free-space check, for example, suffices for resource limiting as admission control for a cache; QoS policy requires logic like [36, 37] to predict SLA compliance given existing caches. The cache size is then determined. For instance, it can be set based on the instance's price tier; to enforce QoS, administrators can profile a client's hit-rate v.s. cache space relationship [62] and allocate enough space to meet SLAs. While running, Nyx assumes the admission logic is correct and is unconcerned about the space allocated.

3.5 Cache Instances: PM-Optimized Pelikan

Nyx has been designed to handle any in-memory key-value store; our current implementation is built upon Pelikan – Twitter's in-memory KV cache [17, 75]. We describe the original Pelikan and optimizations for higher PM performance.

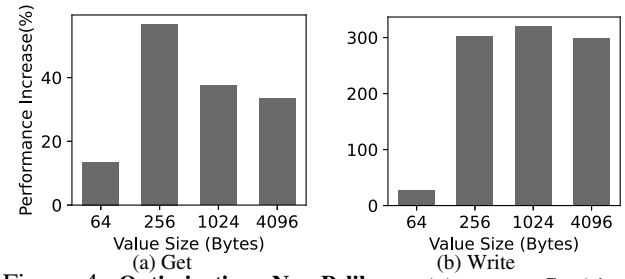


Figure 4: **Optimization: Nyx-Pelikan.** *(a) presents Get (single-thread) throughput improvement due to key-value separation. (b) presents Write (replace, 8 threads) improvement due to changing stores to NT-stores.*

Pelikan (SegCache [75]) maintains a hash table for indexing and segments for storing key-value pairs. Each segment includes items, where each item is a tuple of (key, value, metadata). On a get operation, Pelikan hashes the key to find items. Because of conflicts, multiple keys are likely to be read for a single get. Thus, Pelikan must compare each read item with the key; if the keys match, the value is returned.

When the default version of Pelikan is configured for PM, the hash index is kept in DRAM and the segments in PM. However, this placement is inefficient due to the frequent key accesses in PM: the keys in caching workloads are often much smaller [74] than the granularity of PM access (256B), and small reads perform relatively poorly on PM [73].

Nyx-Pelikan addresses this by separating keys (and metadata) from values into different segments; the keys (and metadata) are placed in DRAM and the values in PM. This design requires DRAM for keys and metadata, which works well because they are typically much smaller than values [73].

As shown previously in Table 2, because non-temporal stores to PM can provide much greater throughput than conventional stores, Nyx-Pelikan uses NT-store. Although non-temporal stores may not benefit from temporal locality in the CPU cache, this loss is negligible on large-scale caching workloads which typically have large working sets. As shown in Figure 4, Nyx-Pelikan improves Pelikan Get performance by up to 55% and set performance by up to 3×.

3.6 Nyx Parameter Values

The values of Nyx's parameters affect its behavior; as previously stated, the appropriate settings depend on the tradeoffs made by administrators. Nyx enables users to configure all of these parameters while also setting defaults.

Nyx follows existing guidelines [38, 63, 77] for policy parameter values' selection. For resource limiting, Nyx uses 10 ms tick and 100 ticks per epoch to limit resource usage offset to 1%. For fair slowdown, Nyx sets the T_{alone} refresh interval to one second to achieve a relatively quick response to workload changes and a within 2% overhead (§4.1).

Nyx provides defaults for newly introduced parameters via sensitivity tests (§4.7). Nyx QoS uses 500MB/s ExperimentStep because it is the smallest step that produces good interference analysis. In interference-aware resource dona-

Trace	Type	Avg.Key/Value Sizes(B)	Operations (Get/Write ratio)
S1	Storage	36/799	0.86/0.13
C1	Computation	67/2439	0.93/0.07
C2	Computation	18/67485	0.52/0.48

Table 3: **Twitter Traces.**

tion, Nyx sets a 10% DonateStep to balance quick donation and steady donator performance. Nyx sets 10ns throttling delay granularity for fine-grained access rate regulation, which is an order of magnitude less than 100ns PM latency. We will discuss potential optimizations like dynamic/adaptive parameters and automatic parameter value selection in §5.

4 Evaluation

We evaluate the overhead of Nyx's mechanisms and how well Nyx provides the sharing policies of resource limit, QoS, fair slowdown, and proportional resource allocation.

Setup: We use a 16-core, single-socket Intel Xeon Gold 5128 CPU @ 2.3GHz server (Ubuntu 18.04), with a 22 MB L3 Cache, 2x16GB DRAM, and 2x128GB Intel Optane DC PMM in app direct mode. We mount an ext4 file system in DAX mode on the PM.

Synthetic Workloads: We begin with synthetic workloads to illustrate key features. Unless specified, the workloads have uniform random accesses to each cache instance, a working set of 10GB per instance, and 4B keys and variable-sized value. To focus on PM accesses, we use get workloads with a high hit ratio (>99 percent). We use in-place replacement for write-heavy workloads; a cache write implies a replace. The cache is warmed to begin.

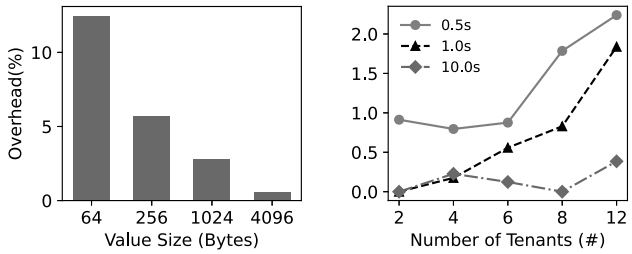
Realistic Workloads: We conclude with three large-scale cache traces from Twitter [74] (Table 3). The traces cover caches with various value sizes (799B to 67845B) and get-percentages (93% to 52%). We pre-load one million operations from the traces and loop through them.

4.1 Mechanisms Overhead

Request Regulation and Resource Usage Estimation: With Nyx, each PM access incurs a call into Nyx-lib, throttling logic, and resource accounting. Figure 5.a shows this can add up to 12% overhead for extremely small value sizes (e.g., a cache line), but less than 6% for access sizes above 256B. Given the benefit of request regulation and resource usage accounting, we believe this overhead is justified.

Interference Analysis: Determining the most interfering client takes longer than simply selecting the client with the greatest bandwidth due to the lag necessary to observe tail latency. In Section 4.3 we will demonstrate the benefit of trading increased analysis time for more precise information.

SlowDown Estimation: The overhead of slowdown estimation is influenced by the time to measure T_{alone} per instance, the frequency of this measurement, and the number of cache instances. We determined that 1ms is a sufficient pause time to accurately determine T_{alone} for a client. Figure 5.b shows that calculating T_{alone} for up to 12 instances adds less than 2.5% overhead, even when performed every 500ms.



(a) Regulation, Accounting Overhead (b) Slowdown Estimation Overhead

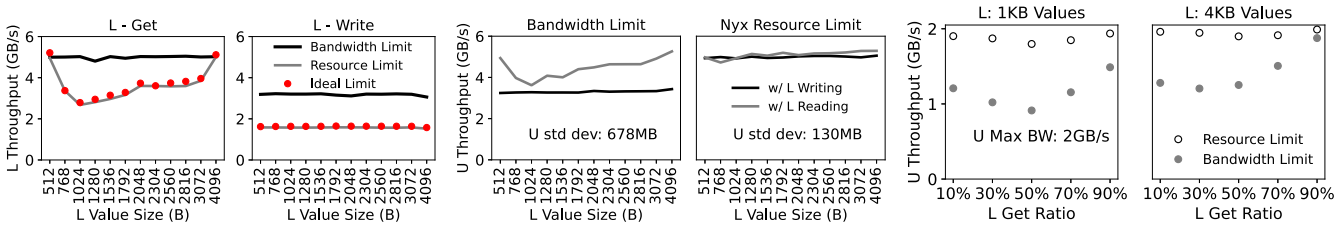
Figure 5: **Mechanisms Overhead.** *(a) shows Nyx request regulation and resource usage accounting overhead (throughput). It is measured with 8-threads get-only caches. A similar percentage of latency overhead was observed. (b) shows Nyx slowdown estimation overhead (throughput). It is measured with 1ms T_{alone} pausing time, different number of clients (x axis) and different frequency (0.5/1/10s) of updating T_{alone} for all caches.*

4.2 Resource Limiting

We demonstrate that Nyx can enforce a true resource limit on PM, in contrast to an approach based only on bandwidth. We begin with a workload containing one unlimited (U) cache and one limited (L) cache. Cache U is a get-heavy cache instance, while Cache L changes: get-only or write-only, with varied value sizes. L has a resource limit of 1.25M 4KB random load OPS, or 42% of the total device resource given that MaxIOPS for 4KB random loads is 3 Million. Defined in terms of bandwidth, this equates to 5GB/s for these 4KB random loads; however, this IOPS limit results in different bandwidths for other workloads.

Figure 6.a shows the bandwidth of L; the target IOPS, in which no more than 42% of the device resource is used, is shown in red. As desired, Nyx always limits L's throughput to the target limit, regardless of L's access pattern (determined by value sizes and read/write). In contrast, a policy based only on bandwidth mistakenly allows L to significantly exceed the target limit, up through the maximum bandwidth of 5GB/s. When L is get-only, this problem is most noticeable when the value size is around 1KB; as previously noted, 1KB accesses result in significant CPU prefetching waste not captured by software-level bandwidth accounting. On the other hand, Nyx's MaxIOPS cost model accurately captures resource usage. Similarly, bandwidth cannot capture PM write cost and fails to properly limit L's throughput.

The impact on the unlimited client (U) is shown in Figure 6.b for the same L workloads. With the bandwidth policy, U's performance depends on L's access pattern. Due to asymmetric read/write cost of PM, whether L performs reads or writes significantly impacts U; similarly, the varied prefetching waste of each access pattern causes up to 45% impact on U. In contrast, Nyx provides U with steady and predictable performance, regardless of L's access pattern: across all of L's workloads, the standard deviation of U's performance is only 130MB/s (bandwidth limit's deviation is 678MB/s). Finally, Figure 6.c shows that when the percentage of gets in L is varied, Nyx provides steady performance for U, whereas a PM-oblivious bandwidth-based approach does not.



(a) L Throughput (b) U Throughput (c) U + Broader L Setups

Figure 6: **Resource Limit: Cache U (unlimited) + Cache L (limited).** *Cache U is get-only. (a) Cache L throughput when resource limit is 5GB/s (1.25M 4KB random load OPS, or 42% of the total device resources). The red dotted line represents L's performance under the "ideal limit", which is calculated as 42% of the current access pattern's MaxIOPS. L is get-only or write-only, and its value sizes varies (x axis). (b) Cache U's performance when colocated with the same L in (a), comparing bandwidth limit and Nyx resource limit. (c) Additional L setups: 1KB/4KB value sizes and 10% - 90% gets. U is a lighter cache than (a) and (b). The label indicates U's max bandwidth when colocated with a 5GB/s cache instance (4KB-value, get-only).*

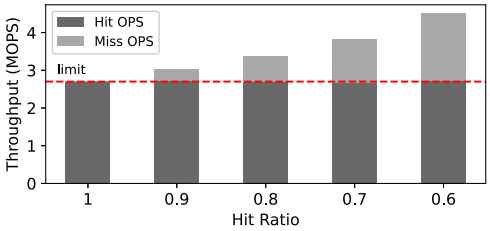


Figure 7: **Resource Limit: Behaviors with a Varying Hit Rate.** *Operations Per Second (OPS) for a Cache with 1KB Get-only workloads when resource limit is 5GB/s. We vary the workloads with different working sets to achieve a different hit rate; note there is no insertion after each miss.*

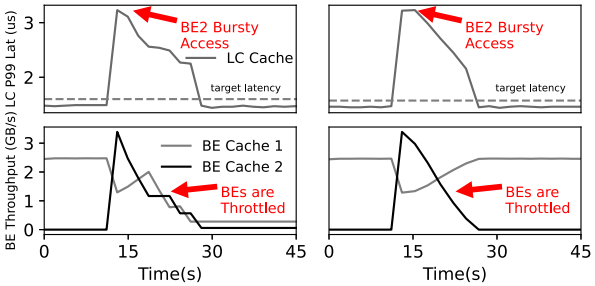


Figure 8: **QoS: Nyx-Caladan vs. Nyx Tuning.** *This figure shows how Nyx and Nyx-Caladan throttle BE caches to ensure LC cache P99 latency. LC cache is colocated with two BE caches; BE1 is get-heavy, B2 is write-heavy (i.e., more interference to LC). BE2 has burst at 12s, breaking LC latency targets. Nyx-Caladan (left) throttles the highest-bw client, whereas Nyx (right) throttles the client with the most interferences to LC. Nyx-Caladan incorrectly throttles BE1, resulting in ×6 less bandwidth for BE1.*

Figure 7 demonstrates Nyx's resource limiting behaviors as the cache hit rate varies. As shown, Nyx restricts PM resource usage from (get) hits. Misses in look-aside caches (e.g., Pelikan) are simply returned after checking the index (in DRAM) and do not use PM resources, so they are not limited.

4.3 QoS-Aware

Nyx can provide QoS guarantees for latency-critical (LC) caches while providing high utilization to best-effort (BE) caches with interference-aware regulation; in contrast, a PM-oblivious approach such as that in Caladan may not be able to deliver the same performance to the BE cache. For comparison, we implemented the Caladan approach in Nyx-Caladan.

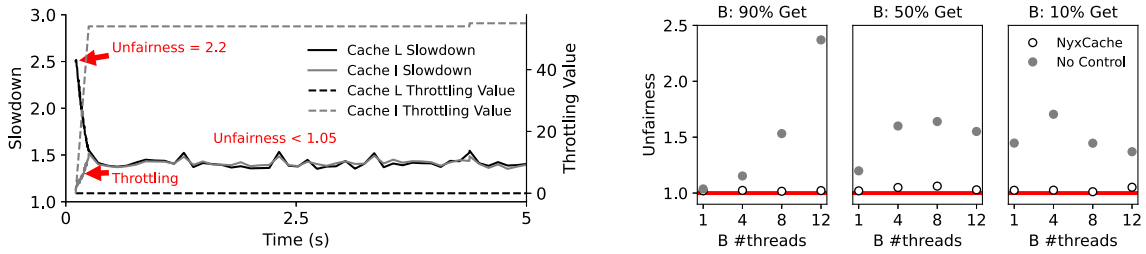
Figure 8 shows an LC cache (P99 latency target of 1.5μs) colocated with two BE caches: BE1 is get-heavy, BE2 is write-heavy. Initially, when BE2 has low throughput and BE1 has moderate throughput of 2.4GB/s, LC meets its P99 objective; however, at 12s, BE2 performs many bursty writes, causing LC's P99 latency to exceed 3μs and violate its target. Both Nyx-Caladan and Nyx resolve the situation by iteratively throttling a BE cache. Nyx-Caladan throttles the cache currently consuming the most bandwidth, shown in the left two subfigures; as a result, Nyx-Caladan throttles both BE1 and BE2, resulting in ×6 less bandwidth for BE1. Nyx, on the other hand, identifies the cache that most interferes with LC as BE2, the write-heavy cache. As a result, Nyx stabilizes to throttling only the correct interference source; after 28 seconds, only BE2 is throttled, and BE1 returns to its original throughput. To summarize, Nyx provides high utilization for multiple caches while guaranteeing each target.

Nyx's convergence time of tens of seconds is similar to prior work such as Parties [31]: the majority of the converging time is spent monitoring tail latencies. As in Parties, Nyx measures tail latency for 500ms because shorter intervals can result in noisy measurements. We leave faster tail latency measurement at network packet queues (as utilized in the original Caladan [39]) for future investigation.

Our experiments reveal that Nyx has an intriguing effect on convergence time: as shown in the Figure, Nyx can bring the LC cache to its target performance in a comparable amount of time to just selecting the cache with the highest bandwidth (which does not require any micro-experiment time). The implication of these results is that, rather than simply acting quickly and throttling any competing instance, Nyx acts correctly and throttles the source of the interference.

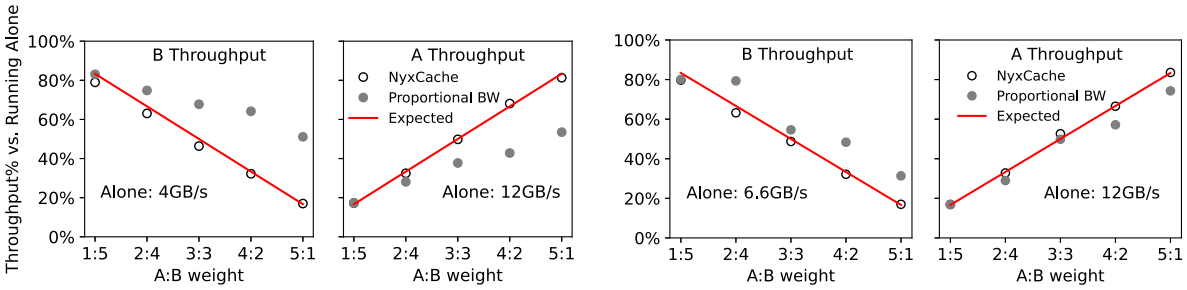
4.4 Fair Slowdown

Nyx implements fair slowdown by iteratively regulating requests according to the measured slowdown of each client (i.e., $\frac{T_{alone}}{T_{share}}$). Figure 9.a shows Nyx's tuning given colocated light and intensive get-heavy caches. Initially, the slowdown of the light cache is 2.2 times higher than that of the intensive cache. Over time, Nyx dynamically increases the throttling of the cache with the minimum slowdown and decreases throttling for the cache with maximum slowdown. Relatively



(a) Tuning Process (Light + Intensive) (b) L + various B

Figure 9: **Fair Slowdown.** *(a) shows how Nyx equalizes slowdown over time for two cache instances (a light one (L) and an intensive one (I)). Both cache instances are get-heavy. (b) shows the unfairness metric when colocating L (a light get-heavy cache) with different B instances (get-heavy -> write-heavy, and light -> intensive). Unfairness = MaxSlowDown / MinSlowDown, the more close to 1, the more fair.*



(a)A: Get-heavy, B: Write-heavy (b) A: Efficient-Get, B: Inefficient-Get

Figure 10: **Proportional Sharing.** *(a) shows A (get-heavy cache) and B (write-heavy cache)'s throughput with different weight configuration. The labels indicate running alone throughput of A and B. With bandwidth allocation, B surpasses its allotted proportional performance. (b) shows A (efficient get-intensive cache, 4KB value sizes) and B (inefficient get-intensive cache, 1KB value sizes).*

quickly, both caches converge to a slowdown near 1.5 and the unfairness metric of $\frac{MaxSlowdown}{MinSlowdown}$ settles near 1.05.

Figure 9.b shows Nyx's fair slowdown policy on a range of caches. Cache L remains a light get-heavy cache; Cache B varies the number of threads and can be get-heavy, 50% mixed, or write-heavy. Without Nyx, L can experience dramatically unfair slowdown (due to PM's complex performance); for example, colocating A with a multi-threaded get-heavy cache B gives unfairness near 2.4. In contrast, Nyx achieves fair slowdown (< 1.05 unfairness) for all 12 cases.

4.5 Proportional Resource Allocation

Nyx achieves proportional resource allocation and guarantees a time-sharing lower bound while performing idle resource re-distribution. We begin with simple scenarios in which two caches that use all their assigned resources are colocated. The scenarios in Figure 10 vary the desired proportional share for A and B along the x-axis; the red line indicates the ideal proportional throughput given their throughput when run alone. Figure 10.a shows that a PM-oblivious bandwidth approach cannot guarantee a proportional share; in particular, the write-intensive B cache obtains up to 3× more throughput than desired and the get-intensive cache A suffers significantly (∼40%). However, by correctly estimating resource usage, Nyx delivers the desired allocation to each cache. Figure 10.b shows a similar effect occurs when efficient-get (value: 4KB) and inefficient-get (value: 1KB) caches are colocated.

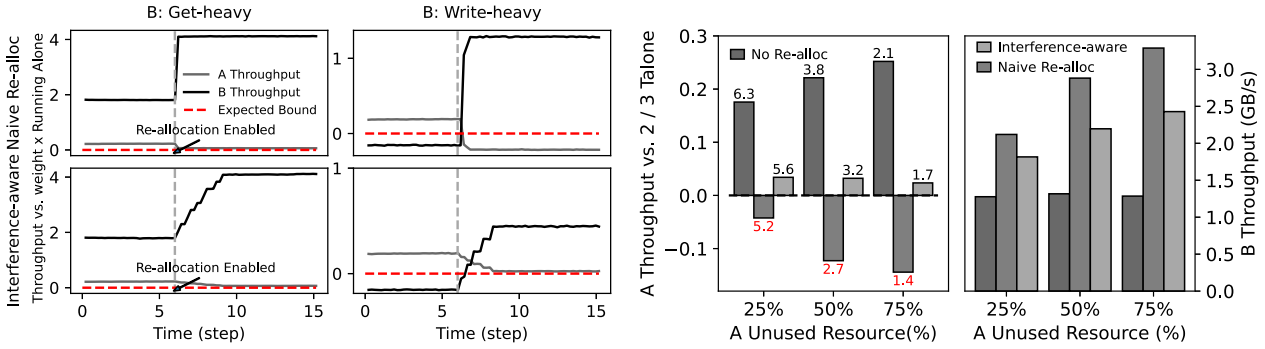
Proportional allocation is more challenging when there are idle resources to be redistributed. Figure 11.a shows two caches A and B, where A uses only 25% of its share. When B is get-heavy (left-top subfigure), A can donate all its idle resources to B; A's performance is slightly degraded, but B receives substantially higher throughput. However, when B is write-heavy (right-top subfigure), if A donates all its idle resources, the higher throughput of B substantially interfere with A, breaching A's time-sharing lower bound (2/3 of A's stand-alone throughput). Therefore, Nyx does not perform naive donation; instead, Nyx donates idle resources in increments while monitoring each cache's slowdown. As shown in the bottom two graphs, Nyx guarantees the time-sharing lower bound for each cache while improving utilization.

We next examine workloads varying the percentage of idle resources in Cache A. When cache B is get-heavy, all of A's idle resources can be safely redistributed to B, and Nyx achieves the same performance for cache B as simple donation (Figure not shown due to space limit). However, when cache B is write-heavy, simple donation of A's idle resources to B violates A's time-sharing bound (Figure 11.b); Nyx accurately protects cache A's performance while still improving the performance of cache B relative to no donation.

4.6 Realistic Traces

Nyx provides isolation for realistic workloads. We demonstrate use cases for resource limiting and slowdown limiting.

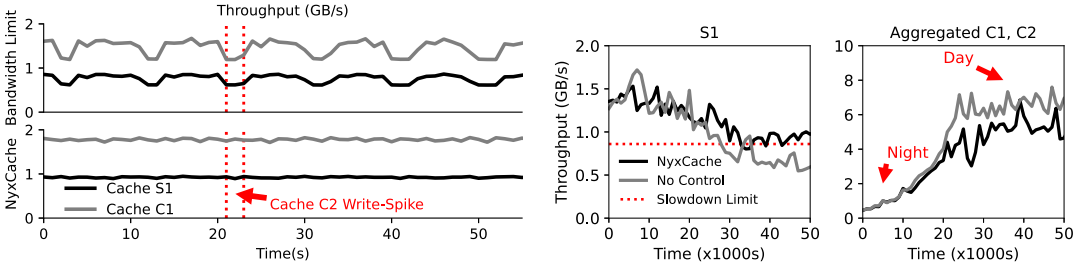
In production workloads, write spikes are common; for example, when a cache is used for ML models, write spikes occur with model parameters are regularly refreshed [74]. Figure 12.a shows how Nyx can isolate caches S1 and C1



(a) Problem of Naive "Donate All" Strategy

(b) Re-distribution between Get and Write-heavy Caches

Figure 11: **Proportional Share: Extra Resource Re-distribution.** *Cache weight A:B is 2:1. (a) shows cache A (light, get-heavy) throughput before and after it donates its extra allocated resource. A has a 75 percent idle resource. Y axis is the normalized difference. When cache B is get-heavy (the top-left figure), A gets nominal performance drop due to donation. However, when cache B is write-heavy (top-right figure), donating cause severe slowdown for A. Unlike naive extra re-distribution, Nyx (two bottom figures) ensures that tenant A's performance is always more than two-thirds of its running alone performance. TimeStep = 2ms. (b) shows A's slowdown (left figure) and B's throughput (right figure) before and after A donating extra resource. A is get-heavy and B is write-heavy. The label indicates absolute throughput number. Naive extra resource allocation can easily break isolation guarantee, while Nyx always ensures it.*



(a) Protecting Caches from Write Spikes

(b) Limiting S1 Slowdown During Day and Night

Figure 12: **Realistic Traces.** *(a) shows the performance of Cache S1 and C1 when colocated with Cache C2. Cache C2 has write spikes. Nyx (bottom figure) can isolate write spikes, whereas bandwidth limits cannot (top figure). (b) shows the performance of Cache S1 (the cache we guarantee its slowdown is always smaller than 1.5×). S1 is colocated with C1 and C2; both C1 and C2 have a strong diurnal pattern (light during the night, and intensive during the day). Without Nyx, S1 performance plummets during the day (because the impact from C1 and C2), discouraging sharing. However, Nyx can always offer reasonable performance (e.g. within x1.5 slowdown vs. running alone). The red line represents S1's performance guarantee.*

from (added) write spikes in cache C2. If resource limiting is based only on the bandwidth of C2, S1 and C1 suffer when C2 experiences write spikes. However, Nyx's resource-limit policy can cap C2's resource usage (at 4GB/s, defined as 1M 4KB random load OPS) to keep S1 and C1 steady.

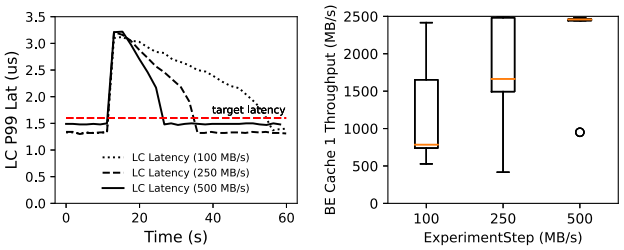
Nyx can also protect the performance of critical caches. To encourage tenants to use multi-tenant PM environments, some caches must be guaranteed performance similar to exclusive use of the PM device. In the experiment shown in Figure 12.b, S1 (the critical cache) is colocated with C1 and C2 which have diurnal patterns [74]. With no control (gray lines), the performance of S1 drops below its target during the day due to the heavy accesses of C1 and C2. However, Nyx can establish a hard limit of slowdown (e.g., 1.5) for S1. As observed, Nyx keeps S1 performance loss within a fair range.

4.7 Parameters Sensitivity Analysis

Here, we present the sensitivity analysis of Nyx behaviors with different ExperimentStep and DonateStep values.

The ExperimentStep affects the Nyx interference analysis's accuracy. As shown in Figure 13.b, using the same configuration as Figure 8, a smaller ExperimentStep is more likely to result in a lower BE 1 final throughput. When ExperimentStep is small, the tail latency change is more likely to be due to measurement noise rather than interference, leading to a less accurate interference analysis. Our experiments suggest an ExperimentStep of at least 500MB/s. ExperimentStep also influences how quickly the Nyx QoS can ensure LC tail latency. As shown in Figure 13.a, a larger ExperimentStep indicates faster convergence. However, it increases the risk of over-throttling BE caches and lowering system utilization. ExperimentStep in Nyx QoS defaults to 500MB/s for good interference analysis and high system utilization while maintaining a reasonable convergence time.

Figure 14 shows how DonateStep affects Nyx proportional resource allocation. A larger DonateStep causes faster idle resource donation, but also potentially large performance fluctuations (e.g., 60% DonateStep, 12s and 18s in the figure). At runtime, cache throughput always varies slightly, causing donation adjustments. These adjustments are subtle with small DonateSteps but significant with large ones. The fluctuation harms donors by slowing them down at times (exceeding the limit). Nyx uses a 10% DonateStep, which balances between quick resource donation and steady donor performance.



(a) Convergence Time (b) BE 1 Final Throughput

Figure 13: **QoS: ExperimentStep Sensitivity Analysis.** *Same as in Figure 8. (a) shows how fast Nyx QoS can ensure LC P99 latency varying ExperimentSteps. (b) shows BE 1 final throughput (boxplot, five runs) varying ExperimentSteps; BE 2 has near zero final throughput in all cases.*

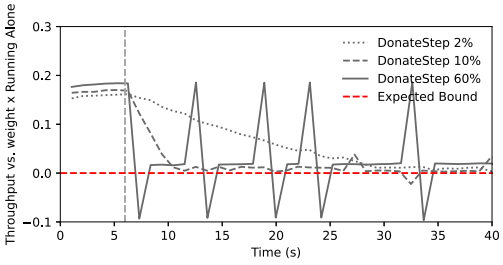


Figure 14: **Proportional Share: DonateStep Sensitivity Analysis.** *We use the same setup as in Figure 11 when B is write-heavy. This figure shows A, the donator's throughput over time with different DonateStep.*

5 Discussion

Beyond Basic Policies: Nyx can be extended to more sophisticated policies for more complex setups. For instance, a proportional sharing policy can be applied across groups of caches. Then, within a group, another sharing policy (e.g., QoS) can be enforced. We leave a full study as a future work.
Multi-tenant Caching Alternatives: Nyx manages caches, each with its own space. There are alternatives to shared caching; for instance, a single large instance can be shared by multiple users [60]. This model can make use of the Nyx resource usage accounting and interference analysis techniques. However, it may create new problems like: how should users be charged for PM writes to commonly cached objects?
Smarter Parameter Value Selection: i) Adaptive parameters can be beneficial, e.g., the DonateStep can be larger when it is far from the threshold (for quick donation) and smaller when it is close (to avoid performance fluctuations). ii) Auto-tuning [46, 68] may ease the load for choosing parameter values. We leave these optimizations as future work.
Security: Nyx policies can be attackable, e.g., in resource limiting, an adversary client may limit its access in the first ticks while putting significant load in the last. A solution would be to use randomized measuring points rather than fixed ones. We leave Nyx security studies as future work.

6 Related Work

Multi-tenant in-mem key-value caching: Our work builds on past research in multi-tenant in-memory key-value cache systems. These efforts include techniques for allocating space across tenants [29, 32, 34, 60, 62] as well as optimization of individual cache instances [25, 27, 28, 33, 42, 54, 75]. Our work instead focuses on the challenges of access regulation and information extraction when many caches share PM.
PM Caching: There have been efforts to integrate PM with individual caching systems. Previous work covers databases [50, 72, 81], file systems [22, 48, 80], in-memory key-value caches [6, 19, 21], and general policies [26, 27]. However, to the best of our knowledge, we are the first to address PM issues in multi-tenant caching settings.
PM Interference: Several efforts have characterized PM devices [45, 69, 71, 73]. However, only a few have investigated the interference effect in PM. To our knowledge, Dicio [55] is the first work in this space. Both Dicio and our work observe the different read-write interference effect in PM. However, the goals of Dicio and Nyx differ. Dicio's purpose is to identify when PM DIMM bandwidth is saturated. Dicio approximates this by using the write pending queue (WPQ) delay as a heuristic. We, on the other hand, aim to provide mechanisms for per-client (not per-DIMM) resource usage accounting, slowdown estimation, and cross-client interference analysis. Dicio protects a single LC task from a single BE task, while our QoS policy applies to multiple clients. Dicio acknowledges that deciding which best-effort task to throttle, with PM media-level statistics, was challenging (and hence not done); we address this issue with a run-time method for interference analysis. Finally, Dicio extends Caladan [39] to use CPU scheduling to regulate PM accesses. This approach is applicable to all applications, including cache, but requires application modifications to use Caladan's unique runtime system (not fully Linux compatible). We leave CPU scheduling approaches for PM regulation to future work.
Sharing Other Resources: Efforts have been made to manage and share other resources such as network, CPU, LLC, storage devices, and locks [31, 39–41, 43, 44, 51, 56, 57, 59, 65]. They are essentially orthogonal to our work; we plan to integrate PM management into these systems in the future.

7 Conclusion

We demonstrated that prior DRAM or storage device-intended approaches for access regulation, resource-usage estimation, and interference analysis fail to work on PM due to its unique properties. We introduced Nyx, which enables these mechanisms in a lightweight manner without hardware support. We showed that Nyx can support a variety of multi-tenant cache sharing policies, meeting performance or sharing goals better than earlier DRAM or storage approaches.
Acknowledgments. We thank Ali R. Butt (our shepherd), the anonymous reviewers, and ADSL members for their valuable input. This material was supported by funding from NSF CNS-1838733, CNS-1763810, Google, VMware, Intel, Seagate, Samsung, and Microsoft. The authors' opinions and findings may not reflect those of NSF or other institutions.

References

[1] Amazon elasticache. https://aws.amazon.com/elasticache/.

[2] Amazon elasticache pricing. https://aws.amazon.com/elasticache/pricing/.

[3] Aws elasticache. https://aws.amazon.com/elasticache/redis/customers/.

[4] Budget fair queueing i/o scheduler. http://algo.ing.unimo.it/people/paolo/disk_sched/.

[5] Caching at reddit. https://redditblog.com/2017/1/17/caching-at-reddit/.

[6] Caching on pmem: an iterative approach. yue yao. https://www.snia.org/educational-library/caching-pmem-iterative-approach-2020.

[7] Google memcache resource limit. https://cloud.google.com/appengine/docs/standard/python/memcache.

[8] Intel mba issue with pm. https://github.com/intel/intel-cmt-cat/issues/170.

[9] Intel memory bandwidth allocation (mba). https://software.intel.com/content/www/cn/zh/develop/articles/introduction-to-memory-bandwidth-allocation.html.

[10] Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[11] Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.

[12] I/o scheduling. https://en.wikipedia.org/wiki/i/o_scheduling.

[13] ipmctl mediareads, mediawrites. https://docs.pmem.io/ipmctl-user-guide/instrumentation/show-device-performance.

[14] Memcached. https://memcached.org/.

[15] Memcached stats. https://docs.oracle.com/cd/E17952_01/mysql-5.6-en/ha-memcached-stats.html.

[16] Memcachier. https://www.memcachier.com/.

[17] Pelikan - twitter. https://twitter.github.io/pelikan/.

[18] Pelikan cache - taming tail latency and achieving predictability. https://twitter.github.io/pelikan/2020/benchmark-adq.html.

[19] Pmem redis. https://github.com/pmem/pmem-redis.

[20] Redis enterprise cloud. https://redis.com/redis-enterprise-cloud/overview/.

[21] The volatile benefit of persistent memory - memcached. https://memcached.org/blog/persistent-memory/.

[22] 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.

[23] 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.

[24] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[25] 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.

[26] Nathan Beckmann, Phillip B Gibbons, Bernhard Haeupler, and Charles McGuffey. Writeback-aware caching. In *Symposium on Algorithmic Principles of Computer Systems*, pages 1–15. SIAM, 2020.

[27] Nathan Beckmann, Phillip B Gibbons, and Charles McGuffey. Block-granularity-aware caching. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 414–416, 2021.

[28] 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.

[29] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, 2018.

[30] E Chen, D Apalkov, Z Diao, A Driskill-Smith, D Druist, D Lottis, V Nikitin, X Tang, S Watts, S Wang, et al. Advances and future prospects of spin-transfer torque random access memory. *IEEE Transactions on Magnetics*, 46(6):1873–1878, 2010.

[31] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[32] 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.

[33] 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.

[34] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, 2017.

[35] Craciunas, Silviu S and Kirsch, Christoph M and Röck, Harald. I/o resource management through system call scheduling. *ACM SIGOPS Operating Systems Review*, 42(5):44–54, 2008.

[36] Christina Delimitrou, Nick Bambos, and Christos Kozyrakis. Qos-aware admission control in heterogeneous datacenters. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 291–296, 2013.

[37] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[38] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ACM Sigplan Notices*, 45(3):335–346, 2010.

[39] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[40] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *FAST*, volume 9, pages 85–98, 2009.

[41] Ajay Gulati, Arif Merchant, and Peter J Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *OSDI*, volume 10, pages 437–450, 2010.

[42] 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.

[43] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, 2017.

[44] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, 2018.

[45] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[46] Yichen Jia and Feng Chen. Kill two birds with one stone: Auto-tuning rocksdb for high bandwidth and low latency. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 652–664. IEEE, 2020.

[47] Ram Srivatsa Kannan, Michael Laurenzano, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Caliper: Interference estimator for multi-tenant environments sharing architectural resources. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(3):1–25, 2019.

[48] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[49] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, 2009.

[50] Gang Liu, Leying Chen, and Shimin Chen. Zen: a high-throughput log-free oltp engine for non-volatile main memory. *Proceedings of the VLDB Endowment*, 14(5):835–848, 2021.

[51] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[52] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[53] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222. IEEE, 2006.

[54] 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 *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 385–398, Lombard, Illinois, April 2013.

[55] Jinyoung Oh and Youngjin Kwon. Persistent Memory Aware Performance Isolation with Dicio. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 97–105, 2021.

[56] Jinsu Park, Seongbeom Park, and Woongki Baek. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[57] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. Hypart: a hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–14, 2018.

[58] Stan Park and Kai Shen. Fios: a fair, efficient flash i/o scheduler. In *FAST*, volume 12, pages 13–13, 2012.

[59] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Michael M Swift. Avoiding scheduler subversion using scheduler-cooperative locks. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[60] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. Fairride: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, 2016.

[61] Kai Shen and Stan Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 67–78, 2013.

[62] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 174–181. ACM, 2015.

[63] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 62–75. IEEE, 2015.

[64] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 639–650. IEEE, 2013.

[65] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. Flin: Enabling fairness and enhancing performance in modern nvme solid state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 397–410. IEEE, 2018.

[66] Team at Redis. Redis. https://redis.io/, 2021.

[67] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. Argon: Performance Insulation for Shared Storage Servers. In *FAST*, volume 7, pages 5–5, 2007.

[68] Benjamin Wagner, André Kohn, and Thomas Neumann. Self-tuning query scheduling for analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1879–1891, 2021.

[69] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508. IEEE, 2020.

[70] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. Metal–oxide rram. *Proceedings of the IEEE*, 100(6):1951–1970, 2012.

[71] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of intel optane ssd. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19). USENIX Association, Renton, WA*, 2019.

[72] 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, 2021.

[73] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. *arXiv preprint arXiv:1908.03583*, 2019.

[74] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020.

[75] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *NSDI*, pages 503–518, 2021.

[76] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T Kaushik, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 474–489, 2015.

[77] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.

[78] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. Flashshare: punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492, 2018.

[79] Jishen Zhao, Onur Mutlu, and Yuan Xie. Firm: Fair and high-performance memory control for persistent memory systems. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 153–165. IEEE, 2014.

[80] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: a tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, 2019.

[81] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2195–2207, 2021.

HTMFS: Strong Consistency Comes for Free with Hardware Transactional Memory in Persistent Memory File Systems

Jifei Yi, Mingkai Dong, Fangnuo Wu, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

File system designs are usually a trade-off between performance and consistency. A common practice is to sacrifice data consistency for better performance, as if high performance and strong consistency cannot be achieved simultaneously. In this paper, we revisit the trade-off and propose HOP, a lightweight hardware-software cooperative mechanism, to present the feasibility of leveraging hardware transactional memory (HTM) to achieve both high performance and strong consistency in persistent memory (PM) file systems. The key idea of HOP is to pick the updates visible to the file system interface and warp them into HTM. HOP adopts an FS-aware Optimistic Concurrency Control (OCC)-like mechanism to overcome the HTM capacity limitation and utilizes cooperative locks as fallbacks to guarantee progress. We apply HOP to build HTMFS, a user-space PM file system with strong consistency. In the evaluation, HTMFS presents up to 8.4× performance improvement compared to state-of-the-art PM file systems, showing that strong consistency can be achieved in high-performance persistent memory.

1 Introduction

File systems are the key cornerstones of many storage services such as key-value stores and databases and applications that persistently store data. In the early days, file systems are designed for performance with loose consistency guarantees. For example, FFS [47] relies on the clean unmount of the file system to avoid consistency issues. In case of crash or power shortages, file system users have to invoke and wait for the lengthy file system consistency checker, i.e., `fsck`, which will detect consistency issues and attempt to recover but with no guarantee [26].

Nowadays, with the speedup of storage devices and their widespread use in applications, performance is not the only feature that applications need. Applications also require strong consistency in order to provide reliable services. For example, key-value stores and databases need strong crash consistency to guarantee that all returned writes are persisted and can be correctly read after a system crash. Upon file systems with no or weak consistency guarantee, these applications have to either compromise on the consistency level or use complicated mechanisms to provide reliable storage. Programming efforts can also be reduced if the file system can provide strong consistency.

The strong consistency of the file system implies per-request sequential consistency, which consists of two aspects. First, for arbitrary file system requests, the modification to the file system states observed by concurrent tasks should be atomic. Most file systems use inode-level locks, which ensures the modification order of different requests to guarantee sequential consistency. Second, whenever the system crashes, after a reboot, all previous file system requests should satisfy the all-or-nothing semantics, i.e., all changes to the file system state by a single file system request should be applied or none of them should be applied. File systems do not necessarily guarantee strong crash consistency. For example, ZoFS [16] do not provide the atomicity of data modification. Suppose a writer crashes halfway through writing; it is possible for a reader to read the partially updated value after the system recovers.

At the same time, modern storage devices have become faster and different. The emerging persistent memory (PM) enforces memory with durability. Consequently, file systems can use load/store instructions to access PM storage with near-DRAM performance. Several PM file systems [11, 13, 16–18, 37, 42, 74, 80] are proposed to exploit the PM characteristics; many of them provide strong consistency.

However, existing PM file systems still require complicated and expensive mechanisms, such as journaling [6, 10] and shadow paging [7,64], for strong crash consistency. Journaling has the double writes problems, while shadow paging needs to propagate the changes to an atomic update, thus it only fits dedicated data structures. The write amplification is related to the data structure it uses and the pattern it writes.

Previous approaches are limited to atomicity unit of CPU writes. Intel's restricted transactional memory (RTM) [32] can provide atomicity of multiple updates. However, file system is incompatible with RTM naturally. Block device based

Table 1: Crash consistency mechanism comparison. The specific write amplification of shadow paging corresponds to the data structure and write locations. The write set size of RTM is evaluated with sequential writes on our platform.

Mechanism	Write Amplification	Write Set	Data Strucute	Crash Consistency
In-place Update	1	Unlimited	Any	No guarantee
Journaling	>2 (double writes)	Unlimited	Any	Strong
Shadow Paging	>1	Unlimited	Dedicated	Strong
Soft Updates	1	Unlimited	Dedicated	Weak
RTM	1	<16k	Any	Strong
HOP	Nearly 1	Unlimited	Any	Strong

file systems access data via IO, which will abort the RTM. Although persistent memory can be accessed directly by CPU load/store instructions, the PM write operations need to be persisted with the help of cache line flush instructions (such as `clflush`, `clflushopt`, and `clwb`) which will abort the RTM.

Recently, Intel proposes its second-generation Optane Persistent Memory products, which in cooperation with the new Xeon platforms enable enhanced asynchronous DRAM refresh (eADR) technique that embraces the CPU cache in the domain of persistence in case of crashes [29]. In particular, the platforms guarantee the persistence of memory writes once they become globally visible, which means that data modification to the persistent memory no longer requires cache line flush for persistence. This gives us the possibility of combining RTM and persistent memory to provide atomicity, concurrency, and persistence at the same time.

Although RTM can be used with PM, several challenges prevent RTM-PM from being used directly in PM file systems. At first, users use file systems to process large data storage and retrieval. However, RTM is limited in both read and write set size, thus can easily abort due to file data copy. Second, there are certain dependencies in the code paths of FS-related system calls. For example, path-related operations (such as `open` and `mkdir`) must be preceded by path lookups, and file indexing must be done before reading and writing a file. The operations can be lengthy and may include memory accesses that do not need to be tracked by RTM. Simply wrapping the entire operation within an RTM not only easily leads to capacity abort, but also increases the probability of conflict aborts.

In this paper, we propose HOP[1], a lightweight hardware-software cooperative mechanism for providing strong consistency in PM file systems. HOP builds on the recent eADR-compliant platforms and leverages Hardware Transactional Memory (HTM) to guarantee the atomic durability of file system updates. To address the capacity limitation of HTM, HOP adopts an OCC[2] [41]-like mechanism to chop a large file system request into smaller pieces, while retaining both concurrent consistency and crash consistency during the execution. To guarantee file system progress, HOP designs cooperative locks as the fallback of HTM. The comparison of HOP and other crash consistency mechanisms is shown in Table 1.

To illustrate HOP, we implement HTMFS, a user-space PM file systems base on ZoFS. Evaluation using FxMark [52], Filebench [72], LevelDB [24], and TPC-C [15] on SQLite [70] shows that HOP outperforms state-of-the-art PM file systems, achieving a similar performance to the weak consistency FS implemention while providing strong consistency. With carefully designed fine-grained concurrency control, HTMFS provides even better performance in competitive cases.

The contributions of the paper include:

- The design of HOP, a lightweight hardware-software cooperative mechanism to provide strong consistency in persistent memory file systems (§3);
- The implemention of HTMFS which provides both strong consistency and performance using HOP (§4);
- Comprehensive evaluation that shows that HTMFS outperforms state-of-the-art persistent memory file systems, proving the effectiveness of HOP (§5).

2 Background and Motivation

In this section, we introduce the background knowledge and motivation of our work.

2.1 File System Consistency and Performance

The original file systems are not built with consistency as the priority, e.g., FFS has no consistency guarantee if a crash happens before a clean unmount [26].

An ancient `fsck` tool simply makes the file system mountable [26], without any guarantees on data consistency or persistency. A `fsck` tool helps recover, repair, and refresh the system.

Without file systems providing consistency, applications need to take responsibility for guaranteeing consistency. For example, to guarantee that data are persisted to file A in atomic, applications need to do the following operations in sequence.

1. Create file B with the same content as file A;
2. Write new data to file B;

[1] HOP is short for Hardware-assisted Optimistic Persistence.
[2] Optimistic Concurrency Control

3. Flush file B to guarantee that the new data is persisted in storage;
4. Rename file B as file A;
5. Sync the directory change;

This obviously is costly for applications. Thus some file systems, such as Ext4 and NOVA [80], provide strong consistency as an optional feature.

However, strong consistency does not come for free. Ext4 uses data journal to provide atomic updates for the data, and therefore has the problem of double writes as shown in Table 1. NOVA can use the CoW (copy-on-write) approach to update data atomically. However, CoW may degrade NOVA's performance by up to more than 60% in our evaluation part.

2.2 Persistent Memory and PM File Systems

Persistent Memory (PM) is an emerging storage technology that enforces byte-addressable memory with persistence. With the same interfaces as volatile memory (i.e., DRAM), data written to PM are guaranteed to retain across power cycling. As a result, the storage hierarchy has changed.

Based on these changes, several PM file systems are proposed to better exploit the PM characteristics for better performance. These file systems revisit existing crash consistency mechanisms in the new scenarios brought by PM, rather than exploring fundamentally different (and more efficient) crash consistency mechanisms of file systems. The only difference would be leveraging atomic instructions to provide small updates up to a single cache line. BPFS [13] organizes the whole file system in a tree structure and provides strong consistency via shadow paging and atomic instructions. PMFS [18] introduces fine-grained journaling and combines atomic instructions and optional shadow paging for data consistency. NOVA [80] is a log-structured file system designed for PM, which combines all the atomic instructions, shadow paging, and journaling for strong consistency. SoupFS [17] is a revisit of the soft update technique on PM and provides no strong consistency guarantee.

Traditional file systems, such as Ext4 and XFS, introduce direct access mode (DAX) to bypass page cache in the data path, optimizing their performance when running on PM. However, this doesn't change the crash consistency level of these file systems.

The byte-addressability and persistence of PM also motivate several user-space file systems, e.g., Aerie [74], Strata [42], SplitFS [37], ZoFS [16], and Libnvmmio [11]. Strata and Libnvmmio use logs to guarantee consistency. SplitFS relies on the underlying Ext4 for metadata processing. ZoFS takes the soft update approach to protect data modification, thus only providing weak consistency. It first updates the data in place and then modifies the size of the file to complete the operation. However, if the system crashes before the file size is changed, partial updates may be read by the next read operation.

In summary, PM brings new opportunities in the design of file systems; while existing new file systems still stick to existing mechanisms for crash consistency, leaving the trade-off between performance and strong consistency a lasting barrier towards fast and reliable file systems.

2.3 Hardware Transactional Memory

Transactional memory provides programmers with an easy (and sometimes efficient) approach to implementing concurrent applications. Hardware transactional memory technologies, such as Intel's Restricted Transactional Memory (RTM) in TSX [1] and ARM's TME [46], provide hardware support of transactional memory. Programmers only need to identify the critical section that wraps the shared memory resources and mark it with `xbegin` and `xend` instructions. The transactional memory mechanism will guarantee that the execution of critical sections can be serialized so that no data race occurs. Executing transactions failing to meet the serializable requirements will be aborted by the hardware, which is detected via the cache coherence protocol. Specifically, data writes of an uncommitted transaction are kept in the private cache of that CPU core and only become globally visible when the transaction successfully commits.

Due to the strong affiliation to the cache implementation, the following limitations will cause HTM to abort, which the users should take care of.

Conflict aborts. HTM uses read/write set to track accesses to memory. Cache lines read in the HTM are added to the read set, and cache lines written are added to the write set. Before HTM is successfully committed, if a cache line in the read set is modified or the write set is accessed by another core, this transaction will be aborted due to conflicts. This type of abort may succeed by retrying the transactions.

Capacity aborts. CPU's private cache size is limited; thus, HTM has limited read and write sets. Any transaction that exceeds the read or write set will inevitably be aborted, no matter how many times the transaction is retried.

Other aborts. Besides conflict and capacity aborts, some other sources could abort a transaction, such as interrupts and HTM-incompatible instructions. It depends on the specific scenario to tell whether a simple retry will make the transaction succeed. For example, if a page fault occurs during the transaction, the interrupt will cause the transaction to abort. In this case, a prefault (trigger the page fault in advance) is necessary before retrying the transaction.

2.4 HTM in PM File Systems

HTM was never an option for file system consistency in the era of block-based storage devices. The emergence of byte-addressable persistent memory gives a chance to use HTM in file systems. However, the volatility of CPU cache forces the use of cache line flush instructions for durability, which intrinsically conflicts with the HTM mechanism that stashes in-flight transaction data within the CPU cache. Until January

2021, Intel's new platform included the CPU cache in the persistence domain, meaning that data that reaches the CPU cache can be guaranteed to be durable even in case of crashes and power shortage, it becomes possible to use HTM upon persistent memory. And it goes beyond that. HTM becomes a good companion to be used with persistent memory. According to Intel [29], only globally visible data will be made durable if a power shortage occurs. In other words, HTM in-flight data modifications will be discarded, making HTM a good alternative approach to enforce atomic updates for crash consistency in file systems. HTM seems promising to be used in file systems to provide both crash consistency and concurrency guarantees at the same time.

3 Design

At first sight, it seems straightforward to equip file systems with HTM: simply wrapping each file system request in a pair of `xbegin` and `xend` can guarantee the ACID of the file system request. However, the reality proves that this is far from enough. Due to the long code path and complicated operations in file system requests, wrapping the entire file system request directly within a hardware transaction will frequently (if not always) result in transaction aborts. Directly adopting HTM in a file system will lead to the following three problems:

1. The long code path may permanently cause capacity aborts;
2. The long code path make the transaction easier to abort due to data conflicts;
3. More works need to be repeated in the retry of the abort transaction.

To resolve the above problems, we designed a lightweight hardware-software cooperative mechanism named HOP. Next, we will first introduce what HOP is and then describe how we use HOP to build an RTM-compliant file system, namely HTMFS.

3.1 HOP

To shorten the code path in the HTM, we split a single file system operation into multiple small pieces. When joined together, they will perform similarly to a single huge transaction. This idea is similar to transaction chopping [68].

All memory accesses in file system operations can be classified into three types:

1. Reads;
2. Invisible writes: updates that cannot be observed via the file system interface (such as memory allocation and updates to the shadow pages);
3. Visible writes: updates that can be observed by the file system interface (like timestamp modification, in-place updates, and the change of file size).

To alleviate the capacity aborts caused by complex file system operations, HOP only wraps visible writes (3) in the transactions. Invisible updates are designed to be able to roll back with minimal overhead, while critical reads are protected by sequence counts.

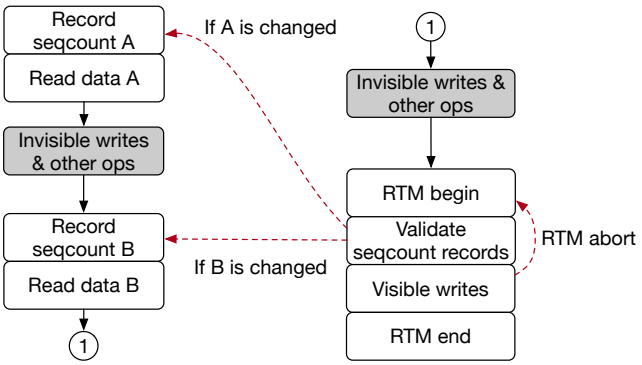


Figure 1: HOP: A transaction wants to read critical data A and B, and write something in atomic. It can read the seqcount and the data in sequence, and validate them in the same RTM with the visible writes to ensure A and B do not change during the whole execution; i.e., the whole process can be considered as an atomic transaction.

In more detail, we first perform all reads and invisible writes outside the RTM. Then we wrap the visible writes to persistent memory using an RTM to complete them atomically. However, not applying any protection to the first part may lead to concurrency errors. HOP ensures concurrent consistency by protecting the fields that may cause concurrency errors by sequence counts. As shown in Figure 1, when we want to access the protected fields outside an RTM, we will first record the corresponding sequence count and then access the persistent memory. These sequence numbers will be validated when entering the RTM-protected region to ensure that the rest remains unchanged throughout the process as long as the RTM commits successfully. If the validation fails (i.e., there is a sequence count that has been changed), HOP will roll back to the first changed point to restart the transaction. For example, if we find that A's seqcount has not changed, but B's seqcount has been modified, we will take the red dotted line "B is changed" to re-record B's seqcount and re-read B.

Besides a modified seqcount, many reasons (introduced in §2.3) may also cause the aborts. If it is an accidental abort caused by an interruption or something else, retrying the RTM transaction again ("RTM abort" in Figure 1) is enough, as going back to the very beginning would cause unnecessary overhead.

Discussion of concurrency correctness. Next, we will discuss all concurrency scenarios (read-read, write-write, write-read, and read-write) in the HOP. Read-read will not bring problems anytime. Since potentially conflicting writes in the HOP are protected by RTM, two conflicting writes will cause each other to conflict abort until one of them succeeds (or keep aborting each other, making it impossible to move forward, which we will avoid by other methods).

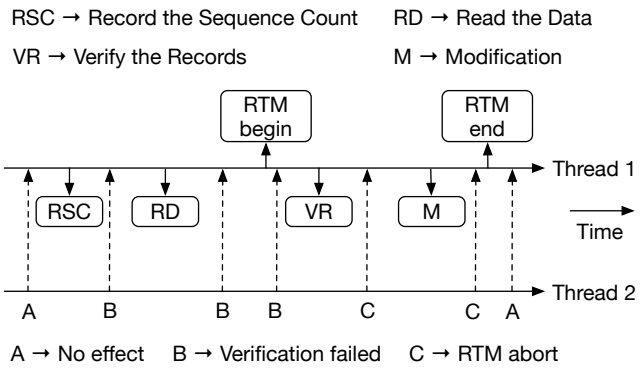


Figure 2: HOP: Thread 2 modifies the sequence count recorded (or to be recorded) by thread 1 at different times, leading to three results. A has no effect on thread 1, while B and C both cause thread 1 to redo, thus guaranteeing that there will be no concurrency errors.

Write-read/read-write, however, will cause an RTM to abort if they conflict, given that potentially confliting write operations are all executed inside the RTM. Any abort will trigger a redo in Figure 1, so a successfully committed transaction guarantees that no concurrency errors exist.

For the read-write scenario, as shown in Figure 2, thread 1 first reads some variables protected by the sequence count, then begins the RTM, validates the sequence numbers, and performs all visible write operations. Thread 2 is simplified to modify the conflict variable at some point in time. The time point modified by thread 2 can be divided into three ranges, resulting in three consequences.

- Result A: If Thread 2 modifies seqcount before Thread 1 reads it or after Thread 1 finishes all operations, it has no effect on the result of Thread 1.
- Result B: If Thread 2 modifies seqcount between Thread 1 reading seqcount and verifying seqcount, it causes Thread 1 to fail validation and thus redo the whole task.
- Result C: If Thread 2 modifies seqcount after Thread 1 verifies successfully (while before the RTM ends), it causes an RTM abort in Thread 1 as it modified Thread 1's read set, thus redoing the whole task.

With HOP, we can break the RTM capacity limit. Then we will introduce how HOP helps to build HTMFS through some specific operations in the file system.

3.2 File Operations

3.2.1 Data Read

For data reads, we use a seqcount-based method to make it atomic. Specifically, the structure of a file is shown in Figure 3, for each page, we first record the persistent pointer (with the sequence count) of the last page, and then read its content. After finishing reading, we verify that the pointers to all records and their sequence counts are unchanged. We will re-read the page that changed and then verify all the sequence

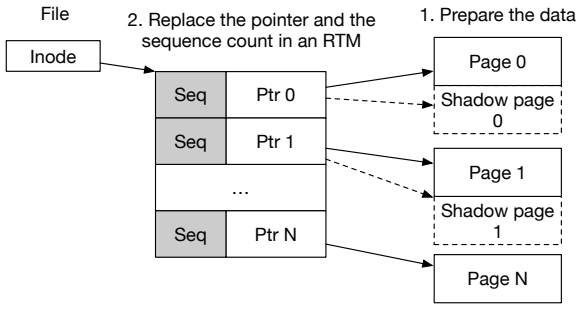


Figure 3: A file is organized in a page table like manner. A single-page update is performed directly wrapped in a transaction. Multi-page updates need to allocate new pages (the shadow pages) for the data, and then copy the new data to the shadow pages. Pointers and the corresponding sequence counts are updated atomically in an RTM.

counts again until all records in this progress stay stable. As only data writes modify the sequence counts or the persistent pointers, we can ensure that there are no changes to the pages we read throughout the entire read operation.

3.2.2 Data Write

Data updates are the foundation of file system operations. One of the major challenges that HTMFS faces is the conflicts between RTM's capacity limitation and the large amount of data involved in file system operations. As a result, directly wrapping the whole file system operation in an RTM transaction will inevitably cause capacity aborts that prevent the operations from being completed.

To address this issue, we propose a hybrid approach that combines the copy-on-write and journalling to convert data updates to metadata updates that can be embedded in the RTM transactions.

Small writes, which fit in a single PM page, are wrapped in an RTM directly. For large data writes, as shown in Figure 3, our strategy first writes data to the persistent memory so that large bulk of data can be represented by pointers, enabling it to be easily embedded in the limited RTM transaction. To explain in detail, we first allocate PM space to store the data. Note that the allocation information is in DRAM, which will not be persisted after a crash. But the data is in PM. Then we start an RTM transaction, in which file system metadata is modified, including the modification of allocation metadata. The persistence point is the RTM commit. Upon a successful commit, the file data and metadata are persistent in an atomic approach. Upon a transaction abort, no changes to the file systems are visible after reboots, with the only exception that the file data are written to the unallocated PM, which is benign most of the time. But the blocks may have leaked after a system crash. Time-consuming scanning of the whole persistent memory can help retrieve the leaked space. To eliminate the recovery process, we design a new allocator based on the free list (as shown in Figure 4) to prevent a memory leak.

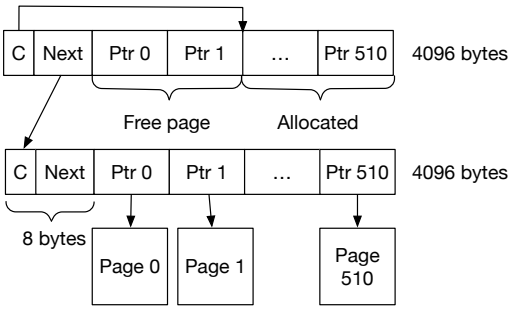


Figure 4: The atomical allocator. `C` stands for `Current`. The pages before `Current` are free pages, while the after is allocated.

3.2.3 Allocation

We split the allocation into two parts: first we move the allocated blocks into a temporal allocating list, which has the same structure as the unallocated space list. Then we simply discard the temporal list inside a transaction to persist the allocation. If a crash happens, we add the temporal list into the unallocated space list to prevent a memory leak.

To ensure that the file system does not reference any unallocated data block, usually the file system modifies (or removes) the reference to the data block before releasing the block. A memory leak may also occur if the file system crashes after a reference to a block of data has been removed (when this block of memory has not yet been freed). When we need to free multiple data blocks, we may also have a crash halfway through the release. An easier way to ensure atomicity is wrapping all these operations in a transaction, but RTM is likely to have a capacity abort. To solve this, we adopt a method similar to the allocation for the free operations. Only operations that must be completed atomically are placed inside RTM, thus avoiding the probability of a capacity abort.

3.3 Directory Operations

3.3.1 Path Walk

File systems usually use a tree structure to maintain the directory hierarchy. Path walking is a quite common scenario in file systems. Many file-system-related system calls require path walking, such as `open`, `mkdir`, `unlink`, etc. These functions will first do a path walk, where the file system will split the full path by slash. It then looks for each level of pathname in turn, starting from the root directory, until it reaches the last level. Then the specified operation (e.g. `open`) is performed on the last file name in the last directory.

All operations that need to walk the path will record the sequence numbers of the directory entries (dentry) it visits, and validate these sequence numbers in the same RTM with the data writes (as shown in Figure 1). Take `touch /a/b` as an example, this operation will first search the dentry `a` in the root directory (`/`). When it finds the matching dentry, it will first record the sequence number of the dentry (`Dseq` of the dentry `a`) and then read the inode number of the directory `a`.

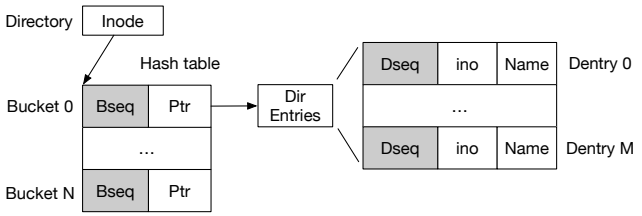


Figure 5: HTMFS uses hash tables to manage the directory entries. `Bseq` is used to serialize changes to directory entries within the same bucket. This prevents the insertion of two files with the same name, etc.

Then it will begin a transaction, validate the sequence number read previously, insert the new dentry `b` into the directory `/a`, and finally commit the transaction.

3.3.2 Directory Updates

We do not use locks on the directory inode to protect updates to the same directory (add/remove a dentry). Instead, we use a separate seqcount in each bucket, and all insert operations need to modify the seqcount of the corresponding bucket in the hash table (Bseq in Figure 5). When inserting multiple different directory entries into a directory simultaneously, the competition will result in only one directory insertion operation succeeding. At the same time, the other will have to redo the whole operation because the sequence number has been modified. In the process of redoing the operation, the operation will find that a directory entry with the same name already exists in the directory and return the error code `EEXIST`.

In file systems, directories can be removed by the system call `rmdir`. However, only empty directories can be removed to avoid deleting useful data accidentally. The utility `rm` can be used to remove a non-empty directory with a parameter `-r`, which will remove directories and their contents recursively. In the implementation, it will remove all the children of the directory first and then delete the empty directory from the file system tree by `rmdir`. This process does not break the restriction that only empty directories can be removed in file systems.

We need to consider the situation that process `A` tries to touch a new file `/a/b/c` into an empty directory `/a/b` while process `B` attempts to delete this empty directory `/a/b`.

As shown in Figure 1, `A` will first walk the path and record the sequence count `Dseq` of `/a/b`'s dentry. Then it will validate the sequence number in the same RTM with the insertion of `/a/b/c`. If the sequence number has been changed before the validation, then `A` will fail to validate it and rollback (lookup the path again). If the sequence number is changed after `A` succeeds in validation, the modification of this sequence number (`B` changes it in another transaction) will cause the transaction of `A` to abort. Then `A` will be rolled back and do again. In the new round of path lookup, it will find that the directory `/a/b` does not exist, which has the same results as if `B`'s entire operation had finished before `A`, will not cause

any problem.

B also needs to validate and modify this sequence number in the same RTM with the operation that deletes this empty directory. Once successfully committed, the insert operation that has not finished the path walking will not be able to find this directory (/a/b), the others will fail to validate the Dseq or be aborted by the modification of the Dseq, thus protecting the correctness of this case. If B is aborted by A because of a conflict, B will find that the directory is not empty when it retries, thus returning ENOTEMPTY as if it is trying to delete a non-empty directory, which is the same as if A operation is finished atomically before B.

This Dseq guarantees that the results of both operations in this case are consistent with a serial execution. So it is no longer necessary to use locks to protect its concurrent correctness.

3.4 Other File Types

Symbolic links. Symbolic links are first expanded to a normal path, and the new path will be returned to the dispatcher, which will re-dispatch the file request. The rest of the operation is just like a normal file.

3.5 The Timestamps

There are several timestamps in file systems to record some information about a file.

- Access timestamp (atime): the last time the file was accessed.
- Modified timestamp (mtime): the last time the file's contents were modified.
- Changed timestamp (ctime): the last time the metadata of the file was changed.

Many file system operations (even read operations such as read, stat, etc.) will modify some of the timestamps. Modifying the timestamp should theoretically happen at the same time as accessing the file, so they need to be done atomically. We need to modify the timestamps in the same transaction as the other operations. Here we observe that placing accesses and modifications to critical variables at the end of a transaction significantly reduces the probability of an abort due to conflicts.

3.6 The Special Case: Rename

Both unlink and rmdir can only remove leaf nodes (files and empty directories) from the file system. Rename, however, has no such limitation and can move a filesystem subtree to another location.

Rename is a special operation that requires atomically removing a directory or a file from the file system tree and adding it to another directory. Usually we will hold locks on both directories to ensure the correctness. However, it may happen that two rename operations both hold a lock and wait to take each other's lock, resulting in a deadlock. This problem can be solved by comparing the two locks and taking the locks in a certain order. However, taking only the locks of the two directories modified cannot prevent the occurrence of a cycle. As shown in Figure 6, there are two path /A/B/C and /X/Y/Z in the directory tree. There are two rename operations; one wants to rename /A/B to /X/Y/Z/B and another wants to rename /X/Y to /A/B/C/Y.

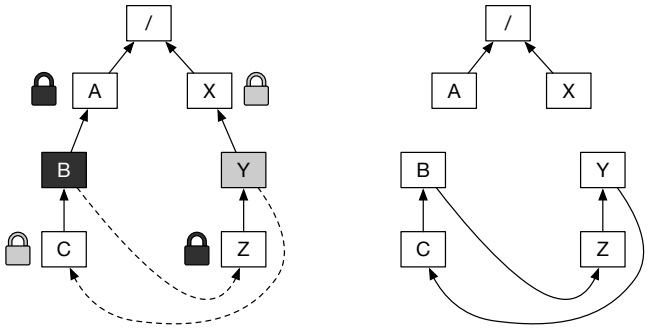


Figure 6: Rename cycle. If rename only locks the parent inode of the source and destination, the rename cycle (outside of the directory tree) may occur.

Take the first operation as an example. 1. First it will walk the path and find the source directory A/B (lock the parent inode A) and the destination /X/Y/Z (lock the inode Z). 2. Then it tries to delete the directory entry B from the directory A and insert a new directory entry B to the directory Z. 3. Finally it will release the two held locks. However, between step 1 and 2, another operation may also finish the path walking and get the two inodes (source Y and destination C). Without other protection, both operations can succeed, thus resulting in a rename cycle. So we need to take extra steps to avoid the cycle, for example, by adding a global rename lock to serialize all rename operations.

We still adopt a lock-free design (HOP) for the rename process. In the path walking (namex), we record the sequence count of all the directories we traversed (as described in § 3.3.1), and finally check if all the sequence counts have changed in one RTM. If there is a change, the namex operation will be executed again from the point of change; if there is no change, the operation of deleting the directory entry and adding it is continued. Since all of the above operations (checking for path changes and modifying directory entries) are done within the same RTM, a successful RTM commit guarantees that the entire rename operation completes atomically. In the preceding example, if both operations complete the path walking and enter the directory modification step (step 2), then when one operation completes, the other operation will abort as its read set is modified, thus re-validating the sequence number and failing because the sequence count has been modified. It then rolls back, redos the path walking and finds the directory tree has changed finally.

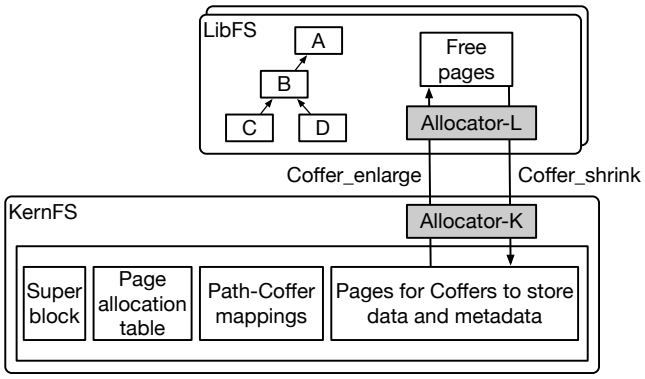


Figure 7: HTMFS consists of a KernFS and a user-space library (LibFS). LibFS calls `Coffer_enlarge` to ask for more PM space from the kernel. When there are too many free pages, LibFS return some to KernFS via `Coffer_shrink`.

4 Implementation

To illustrate the effectiveness of HTMFS, we implement a new file system. After comparing several file systems, we decide to implement HTMFS based on ZoFS [16] because all operations in ZoFS are in user space, thus avoiding the possibility of transaction abort due to system calls.

The overall architecture is shown in Figure 7. ZoFS consists of a kernel-state KernFS and one (or more) user-space file system libraries. HTMFS also consists of two parts, the original KernFS of ZoFS and a new LibFS. In ZoFS, the entire file system tree is divided into multiple zones according to permissions.

4.1 KernFS

KernFS is responsible for maintaining information about all the zones in the entire file system, and the attribution of all persistent memory pages. Each zone has a root page that stores the metadata for that zone. KernFS uses a persistent hash table to store all the zones, where the key is the path prefix of each zone and the value is the relative address of the root page of each zone. When a user-space filesystem library needs to access a path, KernFS uses this hash table to find the root page of that zone and further access that zone.

KernFS manages all PM space globally at page granularity. ZoFS uses a two-level allocation. KernFS allocates PM pages to zones in bulk, and each zone further allocates its pages to store data and metadata. KernFS keeps track of the allocation status of each page, i.e., which zone each page belongs to and which pages are free and can be allocated. In this process, ZoFS uses a global volatile red-black tree to track all free spaces in the allocation table, and another red-black tree [5] to track all allocated spaces and the root page address of the corresponding zone. These volatile data structures can be easily recreated after a system crash.

4.2 LibFS

LibFS is responsible for managing all the metadata and data inside a zone, including mainly files and directories. It contains all the designs in § 3. The file structure, shown in Figure 3, is a three-level structure similar to a page table, supporting files up to 512 GB. Of course, it can be easily extended to support larger files. The directory structure is a hash table as shown in Figure 5.

Since KernFS uses a free list to manage free space, when a zone issues a system call to the kernel to get more free space (`Coffer_enlarge`), KernFS returns a free list. Thus, our LibFS needs to convert the free list to a version recognized by HTMFS (as shown in Figure 4). This prevents modifications to the kernel side.

Fallback path. When RTM fails, we choose to retry or fallback path depending on the return value. We also walk the fallback path when the number of failed retries exceeds the threshold (We choose 60 in the implementation as it gives the best performance when varing the maximum retry number from 10 to 100.). In the fallback path we use inode-level read/write locks for concurrency control and use RTM for crash consistency. When RTM still fails in the fallback path, we use journal as a last resort.

Operations on the normal path will first check if the write lock is held by someone after RTM begins. If the write lock is held by another task, the operation will rollback to the fallback path and try to hold the write lock. If the lock is not held by others during the check, but someone else gets the write lock before the RTM commit, the operation will abort because it's read set has been modified, then retry the RTM operation, and re-check the lock state.

4.3 Prevent RTM abort

There are many causes of RTM abort, starting with RTM capacity abort. The simplest implementation is to wrap the entire file system call in an RTM, and after experimenting we find that most directory and file operations yield capacity abort. After using HOP, HTMFS solve this type of problem.

In our implementation we find that one common cause of RTM abort is page fault, which cannot be predicted. So we prevent page fault failures by first accessing the memory that needs to be accessed and preloading the code to be executed after an RTM abort.

Lastly, the failure is due to conflict, which returns a specific value. In that case HTMFS tries to retry first, which can resolve these conflicts if there is not much competition. If the retries fail a certain number of times, HTMFS fallbacks to the fallback path, i.e., using locks to protect critical code for concurrency control. In fallback path we will first take locks to prevent concurrent accesses and then use HOP to ensure its crash consistency. So it can still meet the strong consistency requirement.

There are some other reasons, such as intermixing AVX and SSE instructions in an RTM, long strings in `REP-MOV*` instructions, etc., which can cause RTM abort [31]. In practice, we found that the `REP-MOV*` instruction used by memcpy will

cause RTM abort in a high probability. So we use cyclic assignments (`SSE2-MOV*`) to replace the memcpy inside RTM.

5 Evaluation

In this section, we evaluate HTMFS against state-of-the-art file systems using different data consistency mechanisms to answer the following questions.

- Can HTMFS's HTM-based hybrid strong crash consistency techniques provide almost as good performance as weak consistency?
- Can HTMFS improve the applications' performance?
- How does HTM improve the performance of file systems?

5.1 Platform Setup

Experiments are conducted on a twenty-eight-core Intel® Xeon® Gold 6330 CPU server. Hyper-threading is disabled, and the CPU frequencies are set to 2.0GHz to get stable results during the evaluation. The server is equipped with 512GB DDR4 DRAM and 1024GB Intel® Optane™ Persistent Memory 200 series.

To evaluate the performance of HTMFS, we compare it against state-of-the-art file systems. ZoFS [16] is evaluated as the baseline. We also evaluate three state-of-art PM-aware file systems (NOVA [80], SplitFS [37], and Libnvmmio [11] on NOVA) to compare. Inode-level locks are used in these file systems. We remove all clflush, clflushopt, clwb, and fence instructions in all of these file systems to improve their performance because these operations are not needed on the eADR platform.

We use FxMark [52], filebench [72], TPC-C [15] on SQLite [70], and LevelDB [24] to evaluate the performance of HTMFS.

5.2 Micro-benchmarks

FxMark includes a set of micro-benchmarks that stress the performance of FS-related system calls. We use FxMark to evaluate the performance and scalability of HTMFS.

Figure 8 shows the performance of file data and metadata operations as the number of threads increases. HTMFS outperforms other file systems in most workloads, including data writes(8(a)(b)) and metadata operations(8(e)(g)(h)). For some workloads, the results of SplitFS and Libnvmmio are not fully displayed, as they get stuck or encounter self-contained errors with an increasing number of threads.

Figure 8a shows the performance for data overwrite operations when different threads overwrite the first 4KB block of different files (DWOL). HTMFS is slower than ZoFS because we replace memcpy (which uses `REP-MOV*` instructions) with `SSE2-MOV*` instructions, which takes more time. If we use `SSE2-MOV*` instructions in ZoFS, the degradation disappears, as ZoFS-SSE2 in the figure shows. Other workloads do not suffer from this degradation because `REP-MOV*`-based memcpy (in ZoFS) only outperforms `SSE2-MOV*`-based memcpy (in HTMFS) when hitting the cache. In DWOL, almost all writes hit the cache, while in other workloads, throughputs are dominated by writes to PM.

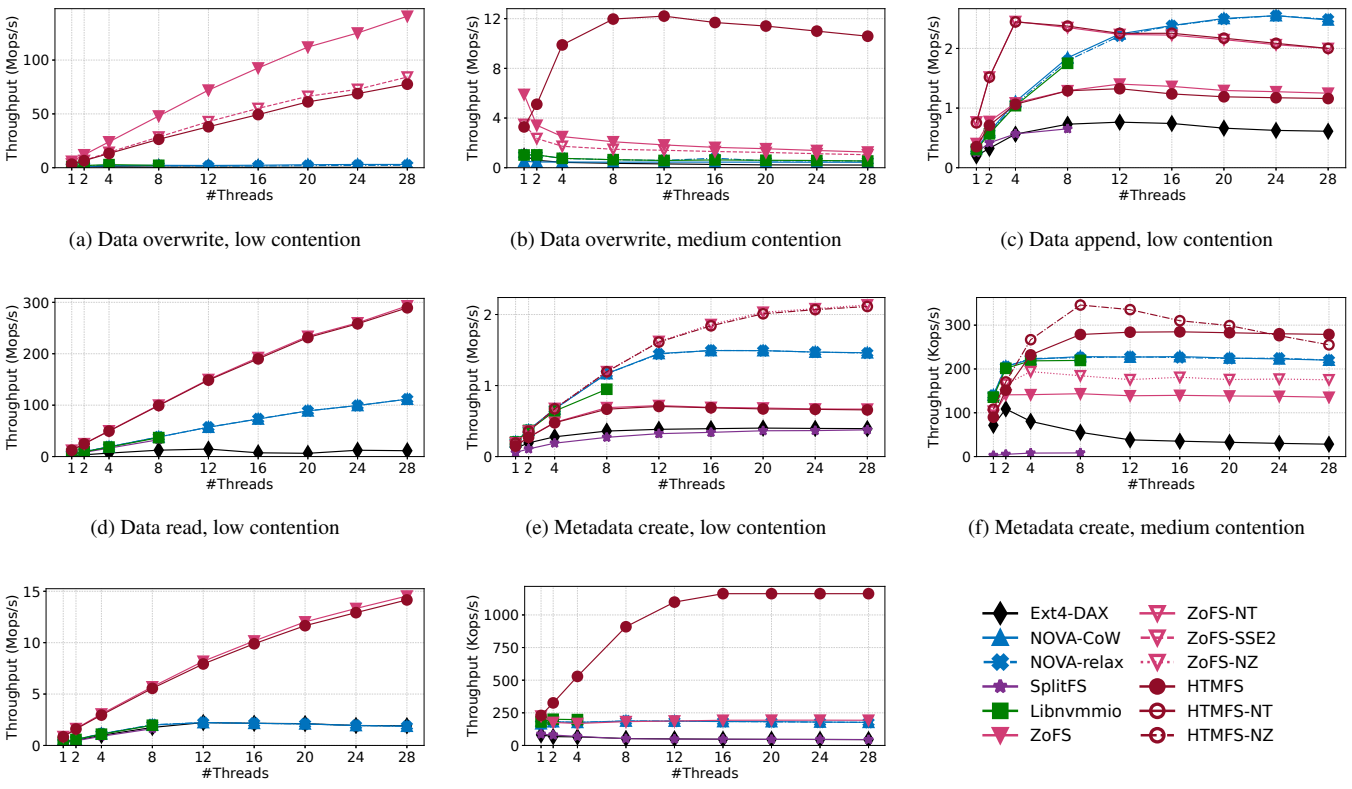
With the medium sharing level, where different threads overwrite different blocks in a shared file (DWOM), HTMFS shows the best scalability. In contrast, the throughputs of other file systems drop as the number of threads increases, as shown in Figure 8b. When there are 28 threads, the throughput of HTMFS is 8.4× of ZoFS. The good scalability of HTMFS mainly comes from the HTMFS's lock-free design. For tests like DWOL and DWOM where the write operations only write to the cache, this part of the difference is magnified to become obvious.

For data append (DWAL, Figure 8c), HTMFS fails to scale after 12 threads. NOVA scales best in this workload, thanks to its per-core allocator. The performance gap between NOVA and HTMFS mainly comes from the different write instructions they use. NOVA uses non-temporal write (NT-write) instructions to store data, which bypass the cache and directly write to PM. It only occupies the write bandwidth of the PM. In contrast, HTMFS uses normal write instructions to store data. In case of a cache miss, HTMFS first reads the data into a cache line, then writes to the cache line, occupying both read and write bandwidth of the PM. However, the reads and writes to the PM interfere with each other, causing a decline in the total bandwidth [81]. Therefore, in DWAL, where most writes miss the cache, HTMFS has lower throughput than NOVA. We replace the write instructions in ZoFS and HTMFS with non-temporal ones and name them ZoFS-NT and HTMFS-NT. They show similar good scalability as NOVA. However, the performance begins to degrade after four threads because ZoFS and HTMFS have reached the upper limit of the PM write bandwidth, which keeps decreasing as the number of threads increases [34, 81].

For data read workloads, when different threads read a block in their respective private file (DRBL, Figure 8d), all file systems scale nearly linearly. Reading a private block in the shared file (DRBM) and reading the same block (DRBH) show similar performance, so these results are not shown here.

For metadata creation workloads, Figure 8e shows the performance when different threads create files in different directories (MWCL). HTMFS and ZoFS stop to scale after eight threads. This is because HTMFS and ZoFS are bounded by the limited PM write bandwidth resource.

However, NOVA performs better than HTMFS with less PM write bandwidth. The reason is that ZoFS uses zero indexes to indicate a non-exist page (as shown in Figure 3). It needs to initialize the file index to zero when creating a file, occupying significant PM bandwidth. ZoFS cannot use file size to indicate whether a page exists in a file because when a crash happens before an update operation completes, the file size will be inconsistent with the file index after reboot. However, the file size and the file index are consistent at any time in HTMFS, which makes it feasible for HTMFS to remove the



(a) Data overwrite, low contention (b) Data overwrite, medium contention (c) Data append, low contention

(d) Data read, low contention (e) Metadata create, low contention (f) Metadata create, medium contention

(g) Metadata rename, low contention (h) Metadata rename, medium contention

Figure 8: Results of FxMark workloads. HTMFS outperformaces ZoFS in DWOM(8b), MWCM(8f), MWRM(8h) and achieves similar performance in most cases. The worse performance of HTMFS (compared with ZoFS, e.g., DWOL(8a)) mainly comes from the gap between a certain instruction we replace.

file index initialization when creating a file. After removing the zero operation from HTMFS and ZoFS, the scalability becomes better than NOVA, as shown by HTMFS-NZ (No Zero) and ZoFS-NZ.

When creating files in a shared directory (MWCM, Figure 8f), HTMFS still scales well while other file systems exhibit poor scalability as the number of threads increases. The good scalability of HTMFS mainly comes from our lock strategy. Instead of locking the parent directory before every create operation, we only need hash table related lock, which avoids a lot competition. HTMFS-NZ performs better than HTMFS because it removes unnecessary memset from code path. It achieves maximum throughput when using eight threads, and degrades as threads increases [34, 81].

For metadata rename workloads, when different threads rename files in different directories (MWRL, Figure 8g), all file systems scale nearly linearly and HTMFS performs best among them. When moving files into a shared directory (MWRM, Figure 8h), like MWCM, HTMFS performs best among them. Thanks to the fine-grained concurrency control provided by HOP, HTMFS outperforms ZoFS by up to 6×.

Abort rate. Since we will retry for up to 60 times, an operation may trigger abort up to 60 times, being counted as 60 aborts. The average abort count is calculated by dividing the number of aborts by the total number of operations completed. After an operation has failed for all the 60 times, it will give up and walk the fallback path. We count the number of times the fallback path is executed and obtain the fallback rate as shown in Table 2. In the several tests both the number of aborts and the number of fallback path executions are negligible.

Table 2: Abort rate. The abort operation accounts for a small proportion of total operations, as well as the fallback path.

Operation	Average Abort Count	Fallback Rate
DWAL-8threads	0.002	0%
DWOL-1thread	0	0%
Varmail-1thread	0.004	0%
Varmail-28threads	0.303	0.17%
TPC-C SQLite	0.001	0%

The latency of the fallback path. For the write operation, we evaluate the operation latency of the normal path (RTM succeeds), the fallback path (RTM fails), and the journal-based path. The results are shown in Table 3. Although the scalability of the normal path is better than the fallback path, their latency is about the same. For writing 4KB files, the latency

Table 3: Operation latency. Each path is independent. e.g., Fallback path latency does not contain that of running a normal path. The journal-based path is slower than others because it requires writing more additional data.

Latency/cycles	Write (4KB)	Mkdir	Rename
Normal path	620.77	12360.67	3378.03
Fallback path	654.47	12486.87	3390.23
Journal-based path	4924.00	13627.00	3548.53
CoW path (NT write)	2293.00	/	/

Table 4: The throughput under high abort rate. We write DWOH that multiple threads write to a shared block in a shared file. HTMFS can fallback to lock rapidly to avoid dramatic performance drops.

Throughput (Kops/s)/#Thread	1	8	28
HTMFS	3732	1309	1111
ZoFS	6148	1241	983
NOVA-CoW	517	416	408
NOVA-relax	1036	1026	992
Libnvmmio	520	416	413

of the journal-based path should be twice the normal path theoretically. However, it is much higher than the theoretical value since the normal path writes are not all written to the PM (reside in the cache). To verify that, we add the latency of the CoW path (using non-temporal write, where the writes fall into the PM directly). The latency of the journal-based path is slightly higher than twice that of the CoW path because the former needs to record some metadata updates.

The latency difference between the different paths is not significant for other metadata operations. The journal-based path requires logging metadata updates, so the latency is slightly higher than the others.

The performance under high abort rate. We design a workload with strong competition for fxmark that multiple threads write to a shared block in a shared file to evaluate how HTMFS's fallback path performs. As shown in Table 4, HTMFS is able to fall back to locks quickly. As the number of threads increases, HTMFS's performance becomes better than ZoFS. The performance of HTMFS is weaker than ZoFS with a single thread, the reason of which is still because we use a slower memcpy to avoid RTM abort.

5.3 Macro-benchmarks

We select two filebench [72] workloads to evaluate the performance of HTMFS. Table 5 summarizes the characteristics of these workloads and the results are shown in Figure 9. We can observe that HTMFS performs well in all chosen workloads.

Webproxy is a read-dominated workload, HTMFS achieves similar performance with ZoFS and shows slightly higher throughput than NOVA and Libnvmmio.

Varmail emulates an email server with a large number of small files and involves both read and write operations. HTMFS is a good fit for this workload as Varmail involves more metadata operations. Besides, NOVA and Libnvmmio also show good scalability.

Table 5: Filebench workload characteristics.

Workload	# Files	Dir Width	File Size	R/W Ratio
Webproxy	10,000	1,000,000	16KB	5:1
Varmail	1,000	1,000,000	16KB	1:1

In both workloads, SplitFS is also tested but not shown here as it fails to scale after 8 threads and not outperforms HTMFS.

5.4 Crash Consistency Correctness

Correctness is difficult to be proven without formal verification. To show the crash consistency correctness of HTMFS, we design a simple experiment to show the difference between HTMFS and ZoFS (a weak crash consistency file system).

We first create a 4KB-file filled with character 'a'. Then we open it and write 8KB 'b' into it with a file system call `write(fd, data, 8192)`. In this process, the file system 1) first allocates a new free page as the second page (4KB–8KB), 2) overwrites the first page (0–4KB) with 'b', 3) fills the allocated page with 'b', 4) then links it to the file data index, 5) and finally updates the file size from 4KB to 8KB and updates both ctime and mtime in the file's metadata to the current time.

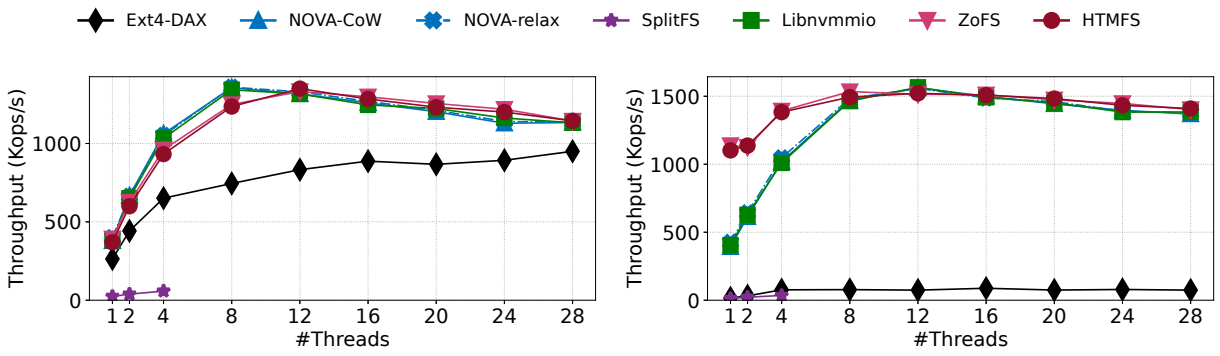
We inject several system crashes during the file system call `write(fd, data, 8192)` and then check some characteristics after rebooting the file system. As all PM writes in the RTM are guaranteed to be persisted atomically [66], rather than injecting crash in the RTM, we insert crash begin/after the RTM. The results are shown in Table 6.

The first two rows show the characteristic of consistent (all-or-nothing) states, respectively. If no change is applied, we should see 4KB "a" in the file, the length of the freelist being 249 (measured in the experiment), the ctime and mtime both unchanged. If the whole operation is finished, the file size, the content, the length of freelist (which should be 248 since a new page will be allocated) and the ctime and mtime should be updated altogether.

Row 3–8 (from ZoFS-1 to HTMFS-3) show the characteristic when ZoFS and HTMFS crash at different points. For every crash point, ZoFS has some difference with both consistent states. For example, at crash point 1, ZoFS is inconsistent for its freelist is reduced by 1, which means there is a persistent memory leak in ZoFS. At the same time, HTMFS is consistent with "nothing" or "all" state, proving HTMFS has stronger crash consistency than ZoFS.

5.5 Application Benchmarks

TPCC on SQLite. SQlite is a widely used lightweight yet full-featured SQL database engine. We drive SQLite with TPC-



(a) Webproxy (b) Varmail

Figure 9: Filebench. HTMFS achieves similar throughput as ZoFS in these workloads.

Table 6: Crash consistency states of ZoFS and HTMFS. We insert three crash points when writing 8KB 'b' into a file of 4KB 'a'. HTMFS is consistent with all-or-nothing states after crash, while ZoFS fail to restore consistent state.

Crash Point	File Size	Content[0]	Len(freelist)	Ctime&Mtime
Nothing	4KB	'a'	249	Not changed
All	8KB	'b'	248	Changed
ZoFS-1	4KB	'a'	248	Not changed
HTMFS-1	4KB	'a'	249	Not changed
ZoFS-2	4KB	'b'	248	Not changed
HTMFS-2	4KB	'a'	249	Not changed
ZoFS-3	8KB	'b'	248	Not changed
HTMFS-3	8KB	'b'	248	Changed

Table 7: TPC-C transaction mix.

Transaction	NEW	PAY	OS	DLY	SL
Ratio	44%	44%	4%	4%	4%

C [15], which is an online transaction processing benchmark that simulates an order processing application.

TPC-C involves five types of transactions: New-Order (NEW), Payment (PAY), Order-Status (OS), Delivery (DLY), and Stock-Level (SL). We use the mixed workload in the experiment and run it with a single thread. Table 7 gives the ratio of different transactions.

Figure 10 summarizes the throughput of different file systems. HTMFS achieves the second highest throughput, which is 2% lower than ZoFS. While NOVA-CoW is 67% slower than NOVA. This demonstrates the low overhead of HTMFS in achieving strong consistency.

LevelDB. LevelDB [24] is a key-value storage library developed by Google. We use LevelDB's `db_bench` benchmarks to prove that we can achieve strong consistency with little overhead. SplitFS and NOVA provide both strong and weak consistent modes. However, we cannot run this benchmark on SplitFS, so we only compare HTMFS with NOVA.

For the read operations, NOVA-CoW and NOVA-relax perform almost the same. For the update operations (fill, overwirte, and delete), NOVA-CoW is obviously slower than NOVA-relax, while HTMFS always performs as well as ZoFS.

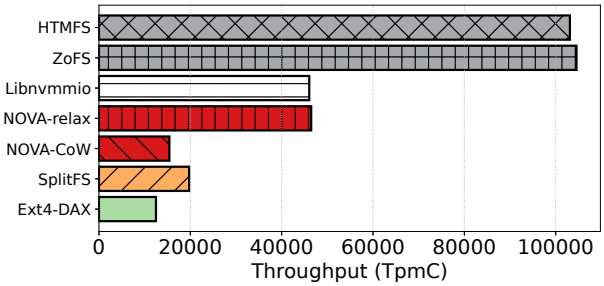


Figure 10: TPC-C SQLite. HTMFS provides stronger consistency with acceptable performance reduction compared to ZoFS, while NOVA sacrifices much more to get the same consistency.

Table 8: Latency of LevelDB. HTMFS and ZoFS perform almost identically, while we can observe a clear latency gap between NOVA-CoW and NOVA-relax. This indicates that HTMFS efficiently achieves data consistency guarantees.

Latency/μs	NOVA-CoW	NOVA-relax	ZoFS	HTMFS
Fill sync.	6.605	5.262	3.190	3.134
Fill seq.	4.605	3.284	2.071	2.039
Fill rand.	31.528	25.142	24.125	24.313
Overwrite.	39.662	31.641	42.128	42.207
Read seq.	1.020	1.004	2.111	2.136
Read rand.	7.357	7.029	11.027	10.600
Read hot.	1.373	1.373	1.289	1.281
Delete rand.	3.169	2.120	1.335	1.281

6 Discussion

6.1 Other File System Features

Previous sections mainly focus on the common file system interfaces, like read/write. We suggest that our design can be further combined with other features.

Compression Some file systems support data compression features to reduce space on storage devices. The compression procedure can be viewed as normal read (read the data and apply the compression algorithm) plus write (write the compressed data), which falls into the scope of our HOP design.

Deduplication Deduplication features the ability to reduce redundancy in stored data to reclaim disk space. It involves scanning all data at intervals to find duplicate blocks and remove them. The scanning part does not introduce any conflicts, and the removing part is no different from common file operations, which can also be handled by the HOP design to ensure consistency.

Checksumming/Encrypting Checksumming and Encrypting features are supported for error-detection and security considerations, which works by checksumming/encrypting the data before write operations and verifying the checksum during read operations. This procedure can be easily wrapped in the original read and write operations protected by HOP.

To summarize, these advanced features are orthogonal to our work and can be implemented in further works.

6.2 HOP in Key-Value Stores

Since key-value stores have a fixed access interface (e.g. `put/get/scan`) like the file system, it is relatively easy to use HOP for key-value stores. Like applying HOP to the file system, when using it for key-value stores, we need to consider how each API needs to be modified to reduce the transaction size.

7 Related Work

To our best knowledge, no prior work has discussed using HTM to improve the performance of strong consistent file systems. We discuss related work in this section.

Persistent Transactions. As PM adds durability to memory, researchers study how to facilitate the PM programming via transaction semantics. Some of these studies [9, 12, 14, 23, 25, 27, 38–40, 44, 45, 48, 49, 56, 61, 62, 75, 82, 83] use software approaches, such as undo and/or redo logs, to guarantee transaction semantics on PM; while the others [3, 4, 8, 21, 33, 35, 36, 44, 54, 63, 71] leverage modified hardware mechanisms. All these existing persistent transaction systems targets on user-space applications or data structures, while our work focuses on using HTM in PM file systems. Compared with the data structures, the file systems put extra challenges due to the FS's inevitable large memory footprint and complex operations.

Before eADR [29] is available, many HTM implementations or modifications are proposed to facilitate PM with HTM [3, 4, 22, 35, 60, 77]. The design of HTMFS is orthogonal to these HTM hardware implementation. Furthermore, HTMFS can be simplified if transaction suspend and resume are supported on the platform, as what is planned in the next-generation Intel's server platform [30].

System Transactions and Transactional FS. Researchers have studied to use transactions in an operating system [58, 59]. TxLinux [65] is the first operating system that leverages MetaTM [60], an interrupt-compatible HTM model, in a cooperative synchronization approach that combines HTM with software locks. TxOS [57] proposes and implements system transactions to provide system-wide transactional support. A transactional Ext3 is implemented in TxOS.

A set of file systems provide transactional APIs to applications so that multiple file operations could be finished in an ACID transaction. Examples include Microsoft TxF [2,50], Inversion [55], OdeFS [20], DBFS [53], TFFS [19], Stasis [67], Amino [78], Valor [69], CFS [51], and TxFS [28]. Unlike these prior studies, our work focuses on leveraging HTM to enforce the performance and strong consistency within each single file system operation. We think it possible to extend HTMFS to implement cross-operation transactions and we leave further exploration as future work.

HTM-assisted OCC. Several prior work has explored to combine HTM with OCC-like mechanisms. DBX [76] first use an OCC-like mechanism to address the limited working set of HTM. Leis et al. [43] proposes to split a database transaction into small pieces, each of which is protected by an HTM transaction. These pieces are then glued together via timestamps to guarantee the atomicity of the whole database transaction. HTCC [79] combines fine-grained locks and HTM-assisted OCC. It uses HTM-assisted OCC only on cold data to reduce the database transaction abort rates and leverages delta-restoration to minimize the overhead of transaction restarts. In contrast to this work that focuses on concurrent consistency of database transactions, our work focuses on providing both concurrent consistency and crash consistency with a combination of HTM and FS-aware OCC-like mechanism.

Page fault in RTM. PfTouch [73] efficiently solves the problem of RTM abort due to page fault by modifying the RTM hardware to recognize page fault and triggering page fault in the abort handler. With RTM hardware support, HTMFS can use these methods to reduce the performance loss due to page fault abort.

8 Conclusion

We provide HTMFS, the first HTM-based PM file system. HTMFS provides strong crash consistency and fine-grained concurrency control with HTM support. Evaluation shows that the performance of HTMFS is as good as file systems that only provide weak crash consistency guarantees while providing strong consistency guarantees. In some competitive scenarios, the performance improvements are significant.

Acknowledgements

We sincerely thank our shepherd Changwoo Min, the anonymous reviewers, Nian Liu, Jingwei Xu, and Qing Wang for their constructive comments and insightful suggestions. This work is supported in part by the National Natural Science Foundation of China (No. 61925206), the High-Tech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), and Huawei. Mingkai Dong (mingkaidong@sjtu.edu.cn) is the corresponding author.

References

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 1, chapter 16. Intel, 2021.

[2] Christian Allred. Understanding windows file system transactions. https://www.snia.org/sites/default/orig/sdc_archives/2009_presentations/tuesday/ChristianAllred_UnderstandingWindowsFileSystemTransactions.pdf, 2009.

[3] Hillel Avni and Trevor Brown. Persistent hybrid transactional memory for databases. *Proc. VLDB Endow.*, 10(4):409–420, November 2016.

[4] Hillel Avni, Eliezer Levy, and Avi Mendelson. Hardware transactions in nonvolatile memory. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363*, DISC 2015, pages 617–630, Berlin, Heidelberg, 2015. Springer-Verlag.

[5] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[6] S. Best. Jfs overview.

[7] Rev C, Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. 10 2000.

[8] Daniel Castro, Paolo Romano, and João Barreto. Hardware transactional memory meets memory persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 368–377, 2018.

[9] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. Association for Computing Machinery.

[10] 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*, SOSP '13, page 228–243, New York, NY, USA, 2013. Association for Computing Machinery.

[11] 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, 2020.

[12] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. Association for Computing Machinery.

[13] 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*, pages 133–146, 2009.

[14] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 271–282, New York, NY, USA, 2018. Association for Computing Machinery.

[15] The Transaction Processing Council. Tpc-c benchmark v5.11. 2021.

[16] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 478–493, 2019.

[17] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 719–731. USENIX Association, 2017.

[18] 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*, pages 1–15, 2014.

[19] Eran Gal and Sivan Toledo. A transactional flash file system for microcontrollers. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 7, USA, 2005. USENIX Association.

[20] Narain H. Gehani, H. V. Jagadish, and William D. Roome. Odefs: A file system interface to an object-oriented database. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 249–260, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[21] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. Crafty: Efficient, htm-compatible persistent transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 59–74, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Ellis Giles, Kshitij Doshi, and Peter Varman. Hardware transactional persistent memory. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, pages 190–205, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Ellis R. Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2015.

[24] Google. Leveldb. https://github.com/google/leveldb, 2021.

[25] 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 Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 913–928, USA, 2019. USENIX Association.

[26] Valerie Henson. The many faces of fsck. https://lwn.net/Articles/248180/, Sep. 2007.

[27] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 703–717, USA, 2017. USENIX Association.

[28] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. Txfs: Leveraging file-system crash consistency to provide acid transactions. *ACM Trans. Storage*, 15(2), May 2019.

[29] Intel. eadr: New opportunities for persistent memory applications. https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html, Jan. 2021.

[30] Intel. Intel® architecture instruction set extensions and future features programming reference. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/architecture-instruction-set-extensions-programming-reference.pdf, May 2021.

[31] Intel. Intel® transactional synchronization extensions (intel® tsx) programming considerations. https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2/intrinsics-for-intel-transactional-synchronization-extensions-intel-tsx/intel-transactional-synchronization-extensions-intel-tsx-programming-considerations.html, June 2021.

[32] Intel. Restricted transactional memory overview. https://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2/intrinsics-for-intel-transactional-synchronization-extensions-intel-tsx/intrinsics-for-restricted-transactional-memory-operations/restricted-transactional-memory-overview.html, 2021.

[33] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 427–442, New York, NY, USA, 2016. Association for Computing Machinery.

[34] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane dc persistent memory module, 2019.

[35] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Dhtm: Durable hardware transactional memory. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 452–465. IEEE Press, 2018.

[36] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, 2017.

[37] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[38] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 399–411, New York, NY, USA, 2016. Association for Computing Machinery.

[39] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 335–349, 2020.

[40] 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 *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 773–787. USENIX Association, July 2021.

[41] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[42] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, 2017.

[43] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting hardware transactional memory in main-memory databases. In *2014 IEEE 30th International Conference on Data Engineering*, pages 580–591, 2014.

[44] 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 Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. Association for Computing Machinery.

[45] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence in transactional persistent memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13, 2015.

[46] Berenice Mann. New technologies for the arm a-profile architecture. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/new-technologies-for-the-arm-a-profile-architecture, April 2019.

[47] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, August 1984.

[48] 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. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. Association for Computing Machinery.

[49] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 789–806, New York, NY, USA, 2020. Association for Computing Machinery.

[50] Microsoft. Transactional ntfs (txf). https://docs.microsoft.com/en-us/windows/win32/fileio/transactional-ntfs-portal, May 2018.

[51] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight Application-Level crash consistency on transactional flash storage. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 221–234, Santa Clara, CA, July 2015. USENIX Association.

[52] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, 2016.

[53] Nicholas Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system. 2002.

[54] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349, 2018.

[55] Michael A. Olson. The design and implementation of the inversion file system. In *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*, San Diego, CA, January 1993. USENIX Association.

[56] pmem.io. Persistent memory development kit. https://pmem.io/pmdk/.

[57] 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, SOSP '09, pages 161–176, New York, NY, USA, 2009. Association for Computing Machinery.

[58] Donald E. Porter and Emmett Witchel. Understanding transactional memory performance. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 97–108, 2010.

[59] Hany E. Ramadan, C. Rossbach, and E. Witchel. The linux kernel: A challenging workload for transactional memory. 2006.

[60] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. Metatm/txlinux: Transactional memory for an operating system. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 92–103, New York, NY, USA, 2007. Association for Computing Machinery.

[61] Pedro Ramalhete, Andreia Correia, and Pascal Felber. Efficient algorithms for persistent transactional memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, pages 1–15, New York, NY, USA, 2021. Association for Computing Machinery.

[62] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163, 2019.

[63] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 672–685, New York, NY, USA, 2015. Association for Computing Machinery.

[64] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, feb 1992.

[65] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: Using and managing hardware transactional memory in an operating system. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 87–102, New York, NY, USA, 2007. Association for Computing Machinery.

[66] Andy Rudoff. Questions about eadr, sfence and intel tsx. https://groups.google.com/g/pmem/c/_DJCFGylfVE/m/L0oyltg8BAAJ, 2020.

[67] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 29–44, USA, 2006. USENIX Association.

[68] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3):325–363, sep 1995.

[69] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 29–42, USA, 2009. USENIX Association.

[70] SQLite. Sqlite transactional sql database engine. https://www.sqlite.org/, 2021.

[71] Long Sun, Youyou Lu, and Jiwu Shu. Dp2: Reducing transaction overhead with differential and dual persistency in persistent memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, New York, NY, USA, 2015. Association for Computing Machinery.

[72] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[73] Rubén Titos-Gil, Ricardo Fernández-Pascual, Alberto Ros, and Manuel E Acacio. Pftouch: Concurrent page-fault handling for intel restricted transactional memory. *Journal of Parallel and Distributed Computing*, 145:111–123, 2020.

[74] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[75] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. Association for Computing Machinery.

[76] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[77] Zhaoguo Wang, Han Yi, Ran Liu, Mingkai Dong, and Haibo Chen. Persistent transactional memory. *IEEE Comput. Archit. Lett.*, 14(1):58–61, January 2015.

[78] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending acid semantics to the file system. *ACM Trans. Storage*, 3(2):4–es, June 2007.

[79] Yingjun Wu and Kian-Lee Tan. Scalable in-memory transaction processing with htm. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 365–377, USA, 2016. USENIX Association.

[80] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[81] 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)*, Santa Clara, CA, February 2022. USENIX Association.

[82] Pantea Zardoshti, Tingzhe Zhou, Yujie Liu, and Michael Spear. Optimizing persistent memory transactions. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–231, 2019.

[83] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 897–911, USA, 2019. USENIX Association.

ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory

Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, Ding Yuan
University of Toronto

Abstract

Persistent byte-addressable memory (PM) is poised to become prevalent in future computer systems. PMs are significantly faster than disk storage, and accesses to PMs are governed by the Memory Management Unit (MMU) just as accesses with volatile RAM. These unique characteristics shift the bottleneck from I/O to operations such as block address lookup – for example, in write workloads, up to 45% of the overhead in ext4-DAX is due to building and searching extent trees to translate file offsets to addresses on persistent memory.

We propose a novel *contiguous* file system, ctFS, that eliminates most of the overhead associated with indexing structures such as extent trees in the file system. ctFS represents each file as a contiguous region of virtual memory, hence a lookup from the file offset to the address is simply an offset operation, which can be efficiently performed by the hardware MMU at a fraction of the cost of software maintained indexes. Evaluating ctFS on real-world workloads such as LevelDB shows it outperforms ext4-DAX and SplitFS by 3.6x and 1.8x, respectively.

1 Introduction

The emergence of byte-addressable persistent memory (PM) fundamentally blurs the boundary between memory and persistent storage. Intel's Optane DC persistent memory is byte-addressable and can be integrated as a memory module. Its performance is orders of magnitude faster than traditional storage devices: the sequential read, random read, and write latencies of Intel Optane DC are 169ns, 305ns, and 94ns, respectively, which are the same order of magnitude as DRAM (86ns) [19].

A number of file system designs have been introduced with the aim of exploiting the characteristics of PM. For example, Linux introduced Direct Access support (DAX) for some of its file systems (ext4, xfs, and ext2) that eliminates the use of the page cache. Other designs bypass the kernel by mapping different file system data structures into user space to reduce the overhead of switching into the kernel [7, 8, 21, 25, 37]. SplitFS, a state-of-the-art PM file system, aggressively uses memory-mapped I/O [21] for significantly improved performance.

All of these systems use conventional tree-based index structures for translating the file offset to the device address. This index structure was first proposed by Unix in the 70s [34] when the speed of memory and persistent storage differed by several orders of magnitude. However, with the emergence of PM, this speed difference has shrunk significantly to the point of being almost negligible. This in turn has shifted the bottleneck from I/O to file indexing overheads.

Indeed, we show in §2 that this indexing overhead can be as high as 45% of the total runtime for write workloads on ext4-DAX (e.g., for Append). While memory-mapped I/O (`mmap()`) can mitigate some of the indexing overhead [11], it does not remove indexing overhead but only shifts its timing to page fault handling or `mmap()` (when pre-fault is used). For example, §2 shows that with SplitFS, file indexing overhead can be as high as 63% of the Append workload runtime. This is 18% higher than that of ext4-DAX, even though the runtime of Append is lower on SplitFS; this is because SplitFS's improved performance further shifts the bottleneck and exacerbates indexing overhead.

An alternative to using file indexing is to use contiguous file allocation. While simple contiguous allocation designs with fix-size or variable-size partitions are known [36], they face two major design challenges: (1) internal fragmentation for fix-size partitions, (2) external fragmentation for variable-size partitions, and (3) file resizing, specifically for expansion which often requires

costly data movement. Therefore, the only use of contiguous file allocation in practice today is on CD-ROMs, where files are read-only [36]. SCMFS [39] proposed the high-level idea of allocating files contiguously in virtual memory. However, it does not address the challenges of contiguous file allocation, namely how files are allocated and how resizing is managed. (Its implementation and evaluation are also only based on simulations).

We propose ctFS, a contiguous file system designed from the ground up for PM. ctFS has the following key designs elements:

- Each file (and directory) is contiguously allocated in the 64-bit *virtual* memory space. We demonstrate the practicality of this idea, given that the 64-bit address space is enormous. Furthermore, the virtual address space is carefully managed by a hierarchical layout, similar to the buddy memory allocation [23], in which each partition is subdivided into 8 equal-size subpartitions. This design speeds up allocation, avoids external fragmentation, and minimizes internal fragmentation (§3.2).
- A file's virtual-to-physical mapping is managed using *persistent page tables* (PPT). PPTs have a similar structure as the regular, volatile page tables in DRAM, except that PPTs are stored persistently on PM. Upon a page fault on an address that is within a ctFS's memory region, the OS looks up the PPT and creates the same mappings in the DRAM-based page tables. Therefore, subsequence accesses are served by hardware MMU from DRAM-based page tables, avoiding the indexing cost.
- Initially, a file is allocated within a partition whose size is just large enough for the file. When a file outgrows its partition, it is moved to a larger partition in virtual memory without copying any physical persistent memory. ctFS does this by remapping the file's physical pages to the new partition using *atomic swap*, or `pswap` (§3.3), a new OS system call that *atomically* swaps the virtual-to-physical mappings. Atomic swap also enables efficient crash consistency on multi-block writes without needing to double-write the data. An atomic write in ctFS simply writes the data to a new space, and then `pswap`s it with the old data (§3.4).

In ctFS, the translation from file offset to the physical address now needs to go through the virtual-to-physical memory mapping, which is no less complex than the conventional file-to-block indexes. The key difference is that page translation can be sped up by existing hardware support. Translations that are cached by TLB will be handled transparently from the software and completed in one cycle. In contrast, a file system's file-to-block translation can only be cached by software. Additionally, ctFS can adopt various optimizations for memory mapping, such as using huge pages, to further speed up a variety of operations.

Our evaluation on Intel Optane DC reveals that ctFS can eliminate most indexing overheads, which results in up to a 7.7x and 3.1x speedup over ext4-DAX and SplitFS [21] on the Append workload. ctFS further improves the throughput of LevelDB running YCSB by up to 3.62x, 1.82x, 3.21x, and 2.45x when compared to ext4-DAX, SplitFS, Nova [40], and PMFS [8], respectively. Finally, ctFS improves RocksDB [35] performance by up to 1.6x when compared to ext4-DAX. The source code of ctFS is available at `https://github.com/robinlee09201/ctFS`.

A limitation of ctFS is that we implement it as a user-space library file system that trades protection for performance. While this squeezes the most performance out by aggressively bypassing the kernel, it sacrifice protection in that it only protect against unintentional bugs instead of intentional attacks. We envision that this is an acceptable, or even desirable, trade-off for data center environments. We discuss other limitations in §5.

2 Understanding File Indexing Overhead

We analyzed the performance overhead of block address translation in Linux's ext4-DAX and in SplitFS [21]. Ext4-DAX is the port of the ext4 extent-based file system to PM. It eliminates the page cache, and directly accesses PM using memory operations (`memcpy()`).

Background on SplitFS. We briefly describe SplitFS for a better understanding. SplitFS splits the file system logic into a user-space library (U-Split) and a kernel space component (K-Split), where K-Split uses ext4-DAX. A file is split into multiple 2MB regions by U-Split, where each region is mapped to one ext4-DAX file. Both U-Split and K-Split participate in indexing: U-Split maps a logical file offset to the corresponding ext4-DAX file, and the ext4-DAX in K-Split further searches its extent index to obtain the actual physical address.

SplitFS also proposed a novel operation called `relink` to improve the performance of file expansion and provide crash consistency on file writes without double-writing data. Under its sync mode, file appends are first made to a staging file, and then relinked to the target file either when `fsync()` gets called or the staging file reaches its size limit; file overwrites are applied in-place. Under its strict mode, every file write, whether it's overwriting or appending data, is applied to a staging file and gets

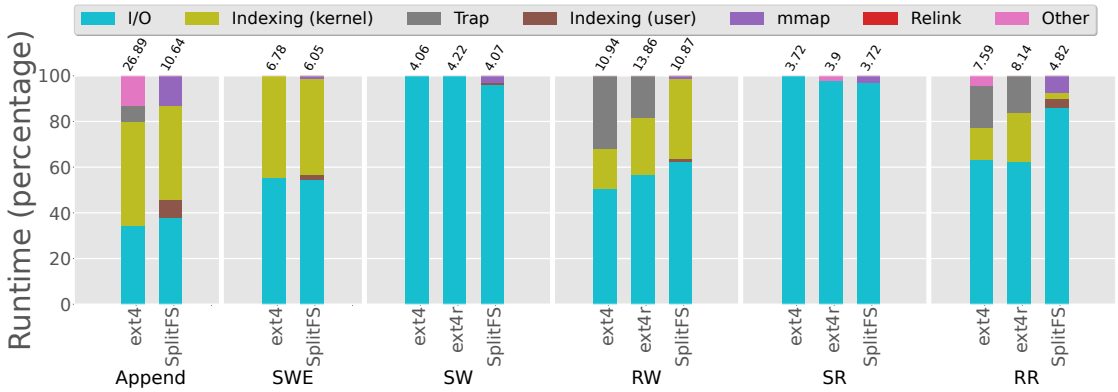


Figure 1: **Performance breakdown** (in percentage) of ext4-DAX and SplitFS on persistent memory. The number above each bar is the total run time in seconds.

relinked at the end of every write. Hence, the indexing time of SplitFS consists of `relink`, mmap, and indexing in both kernel and user components.

Experimental Methodology. Our experiments were conducted on a server with two 128GB Intel Optane DC persistent memory (PM) modules, an 8-core Intel Xeon 4215 CPU running at 2.5 GHz, and 96 GB of DRAM. We used Linux version v5.7.0-rc7+.

We ran a total of 6 tests. The results are shown in Figure 1. Each test either reads or writes a 10GB file. The first test, Append, repeatedly *appends* 4KB of data to a file which is initially empty. The second test, SWE, sequentially writes a total of 10GB of data to an *empty* file with 10 `pwrite()` calls to write 1GB at a time. RR reads 4KB of data from a random (4KB-aligned) offset in a 10GB file, and RW overwrites an existing 10GB file with 4KB of data at a random (4KB-aligned) offset, and they do this 2,621,440 times. Finally, SR/SW we sequentially reads/writes 10GB data, 1GB at a time.[1]

For the SW, RW, RR, and SR tests, we ran the ext4-DAX tests with two types of files: those that were *sequentially* allocated (ext4) and those that were *randomly* allocated (ext4r). Sequentially allocated files were created by SWE, which maximizes ext4-DAX's grouping of blocks into an extent. Randomly allocated files were created by writing to them similarly to the way RW does, except that the file is initially empty (Linux file systems support sparse files); these randomly allocated files limit ext4-DAX's ability to group blocks into extents. The "ext4r" bars in RW, RR, and SR represent tests that operated on such randomly allocated files. Note that ext4-DAX creates 12 extents for a sequentially allocated 10GB file, but creates 256 extents for a randomly allocated file. For SplitFS, all files are sequentially allocated.

Indexing overhead in ext4-DAX. Figure 1 shows the breakdown of the completion time of each test. For ext4-DAX, we observe that indexing overhead is significant in Append and SWE, spending at least 45% of the total runtime on indexing.[2]

For the random access workloads, RR and RW, the proportion of time spent on indexing is lower, but still considerable: 25% and 21% of the total runtime when randomly writing and reading to/from a randomly allocated file (ext4r), and 18% and 15% when the file was sequentially allocated.

Indexing overhead in SplitFS. Figure 1 also shows the breakdown of the completion time of SplitFS's sync mode. [3] Compared to ext4-DAX, SplitFS spends an even higher proportion of the total runtime on indexing in the Append (63%), SWE (45%), and RW workloads (38%), while it spends 14% of the runtime on indexing in RR.

To understand SplitFS's indexing overhead in more detail, consider the Append workload where SplitFS spends a total of 6.62s on indexing. Three components make up this file indexing time: (1) the kernel indexing time as part of page fault handling (4.37s), (2) U-split's file indexing time (0.84s) spent on mapping file offsets to the correct ext4-DAX file, and (3) U-Split's `mmap()` time (1.39s).

3 Design & Implementation of ctFS

This section starts with an overview of ctFS. Then we describe the file system layout (§3.2), and how ctFS interacts with the kernel's memory management system (§3.3). We then explain ctFS's primitive for atomic operations — `pswap()`, and how ctFS handles file updates and en-

[1] We found that the version of SplitFS we tested does not support append operations that write over 128MB under its sync mode. Therefore, in SWE, we write 128MB at a time in SplitFS, instead of 1GB as in ext4-DAX and other the file systems we discuss in §4.

[2] In both cases, the index time includes the time to build the index.

[3] We only show its sync mode result as its semantics is comparable to that of ext4-DAX. SplitFS's strict mode is further evaluated in §4.

Mode	Atomicity		Similar to
	data	metadata	
sync	✗	✓	NOVA-relaxed, PMFS, SplitFS-sync
strict	✓	✓	NOVA-strict, Strata, SplitFS-strict

Table 1: The two modes provided by ctFS.

sures crash consistency (§3.4). Finally we discuss some optimizations (§3.5) and the protection model (§3.6).

3.1 Design Overview

ctFS is a high-performance PM file system that directly accesses and manages both file data and metadata in user space. Each file is stored contiguously in virtual memory, and ctFS offloads traditional file systems' offset to block number indexing to the memory management subsystem. ctFS achieves the following design goals:

- **POSIX compliance:** ctFS currently supports over 30 commonly used functions from the POSIX-compatible file system API.
- **Synchronous writes:** Write operations on ctFS are always synchronous, i.e., writes are persisted on PM before the operation completes. Hence there is no need for `fsync` (which does nothing in ctFS).
- **Crash consistency:** ctFS supports both file data consistency (by using `pswap`) and metadata consistency (by using conventional redo logs).
- **Concurrent operations:** ctFS supports concurrent operations on different files or concurrent reads on the same file; a reader-writer lock is used for each file to synchronize concurrent accesses.

Similar to prior systems, such as NOVA [40] and SplitFS [21], ctFS offers two modes, sync and strict, as shown in Table 1. Both modes ensure atomic metadata operations that include directory operations. Strict mode further ensures file data writes are atomic (by using `pswap`).

ctFS's architecture, shown in Fig. 2, consists of two components: (1) the user space file system library, ctU, that provides the file system abstraction, and (2) the kernel subsystem, ctK, that provides the virtual memory abstraction. ctU implements the file system structure and maps it into the *virtual* memory space. ctK maps virtual addresses to PM's physical addresses using a *persistent page table (PPT)*, which is stored in PM. Any page fault on a virtual address inside ctU's address range is handled

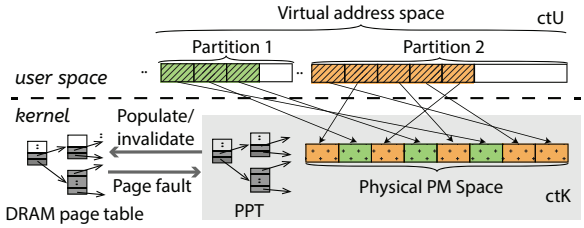


Figure 2: **Architecture of ctFS.** Each box represents a page. Two partitions are shown. The file allocated in partition 1 uses 3 pages (green), and the file in partition 2 uses 5 pages. ctK maintains virtual-to-physical page mappings in the PPT.

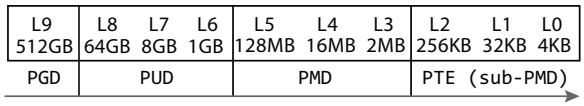


Figure 3: **Size of partitions at levels L0 to L9.** PGD, PUD, PMD, and PTE refer to the four levels of page tables in Linux (from highest to lowest). An L9 partition aligns with PGD, i.e., its starting address has zero in all of the lower level page tables (PUD, PMD, PTE); Similarly, L6-L8 partitions align with PUD, whereas L3-L5 partitions align with PMD.

by ctK. If the PPT does not contain a mapping for the fault address, ctK will allocate a PM page, establish the mapping in the PPT, and then copy the mapping from the PPT to the kernel's DRAM page table, allowing virtual to PM address translations to be carried out by the MMU. When any mapping in the PPT becomes obsolete, ctK will remove the corresponding mapping from the DRAM page table and shoot down the mapping in the TLBs.

With this architecture, there is a clear separation of concerns. ctK is *not* aware of any file system semantics, which is entirely implemented by ctU using memory operations. Next, we discuss the designs that are specific to ctFS. We omit the designs that are similar to existing file systems. For example, we use standard transaction logging to provide crash consistency of *metadata*, including directories, inode, and ctFS data structures such as partition headers, bitmaps, etc.

3.2 File System Structure (ctU)

ctFS's user-space library, *ctU*, organizes the file system's *virtual* memory space into hierarchical partitions to facilitate contiguous allocations. The size of each partition at a particular level is identical, and each level's size is 8x the size of the partitions at the next lower level. Figure 3 shows the sizes of the ten levels that ctFS currently supports. The lowest level, L0, has 4KB partitions, whereas the highest level, L9, has 512 GB partitions. ctFS can be easily extended to support more partition levels, *e.g.* L10

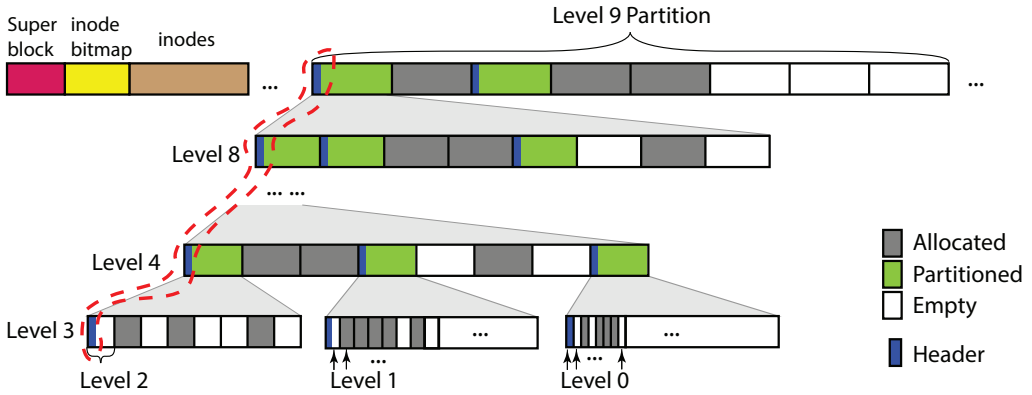


Figure 4: **Layout of ctFS in the *virtual* address space (VAS).** The space of an entire partition is reserved in VAS, whereas the physical PM space is allocated on-demand based on actual usage. Headers circled in the dashed-line reside on the same page.

(4TB), L11 (32TB), *etc*.

A file or directory is always allocated contiguously in one and only one partition, with the size of the partition being the smallest capable of containing the file. For example, a 1KB file is allocated in an L0 partition (4KB); a 2GB file is allocated in an L7 partition (8GB).

We chose each next level to be 8x the size of the previous level because the boundary of the levels should align with the boundary of Linux page table levels (Figure 3). This enables the optimization during `pswap` we describe in §3.3. Therefore, our only options for partition size differences were: 2x (2^1), 8x (2^3), or 512x (2^9). We chose 8x because 2x would be too small and 512x too large.

File System Layout. Figure 4 shows the layout of ctFS. The virtual memory region is partitioned into two L9 partitions. The first L9 partition is a special partition used to store file system metadata: a superblock, a bitmap for inodes, and the inodes themselves. Each inode stores the file's metadata (e.g., owner, group, protection, size, etc.) and *a single field* identifying the virtual memory address of the partition that contains the file's data. The inode bitmap is used to track whether an inode is allocated or not. The second L9 partition is used for data storage. [4]

Each partition can be in one of the three states: Allocated (A), Partitioned (P), or Empty (E). A partition in state A is allocated to a single file; a partition in state P is divided into eight next-level partitions. We call the higher level partition the *parent* of its eight next-level partitions. This parent partition *subsumes* its eight child partitions; i.e., these 8 child partitions are sub-regions within the virtual memory space allocated to the parent. For example, in Figure 4, an L9 partition in state P is divided into 8 L8 partitions. The first L8 partition is also in state P, which means it is divided into 8 L7 partitions, and so on. In this manner, the different levels of partitions form a hierarchy.

This hierarchy of partitions has three properties. (1) For any partition, all of its ancestors must be in state P; and any partition in the A or E state does not have any descendants. (2) Any address in a partition is also an address in the partitions of its ancestors; e.g., any L3 partition in Figure 4 is contained in its ancestor L4-L9 partitions. (3) The starting address of any partition, regardless of its level, is aligned to its partition size; this is the case as long as the top-level L9 partitions are 512 GB aligned.

Partition Headers. ctU needs to maintain book keeping information for each partition, such as its state. To store such metadata, each partition in P-state has a header which contains the state of each of its *child* partitions; ctU stores the header directly on the first page of the partition for fast lookup that does not involve indirections. For example, for each partition in P state at levels L4-L9, the state of its eight children are encoded using 2 bits packed into a `uint16_t`.

To speed up allocation, the header also has an availability level field that identifies the highest level at which a *descendent* partition is available for allocation. For example, the availability level of the left-most L9 partition in Figure 4 is 8 because this L9 partition has at least 1 L8 child partition in E state. With this information, when allocating a level-N partition, if a P partition's availability level is less than N, ctU does not need to drill down further to check its child partitions. This results in constant worst-case time complexity for allocating a parition and is far more efficient than using bitmaps.

Because ctU places the header in the first page of a partition in P state, its first child partition will also contain the same header, and as a result, this first child partition must also be in P state; it cannot be in the Allocated

[4] Note that the 512GB allocated for metadata is virtual memory; The physical pages underneath it are allocated on demand.

state because the first page would need to be used for file content. Therefore, a header page can contain the headers of multiple partitions in the hierarchy. For example, in Figure 4, the headers in the dashed circle are all stored on the same page. This is achieved by partitioning the header page into non-overlapping header spaces for each level from L4-L9.

ctU does not partition L0–L3 further, as the 4KB header space becomes much more wasteful for smaller partition sizes. Instead, each L3 partition (2MB) can only be partitioned as (1) 512 L0 child partitions, (2) 64 L1 child partitions, or (3) 8 L2 child paritions, as shown at the bottom of Figure 4. As a result, there is only one header in each L3 partition that is in state P, and it contains a bitmap to indicate the status of each of its child partitions, which can only be in either state A or E, but not P.

Virtual Memory Allocation. During system initialization, ctU allocates a 1TB, empty (i.e., not backed) virtual memory area (VMA) to accommodate two L9 partitions. It does not restrict the starting address of this VMA, so it can be anywhere in the virtual address space (as long as it is aligned). If the PM size is larger than 512GB, then the next level (L10) would be used, and an 8TB VMA would be allocated. Note that subsequent virtual memory allocations made from the kernel or processes will not clash with ctU's VMA, because the Linux kernel's VMA allocation will only allocate a VMA if it does not conflict with existing VMAs.

TLB usage. ctFS does not use more TLB entries compared to other file systems. In conventional (non-DAX) file systems, the file data will be buffered in memory, either in the file system's buffer cache, or by the process in the case of memory mapped I/O. Such buffering will occupy TLB entries just as ctFS does, and the number of entries used depend on the amount of data a process accesses. Ext4-DAX eliminates the buffer cache by directly accessing the devives using statically mapped virtual kernel addresses. However, this mapping still goes through the page table [14] and hence still occupies TLB entries. Therefore even compared to ext4-DAX, ctFS does not use more TLB entries.

3.3 Kernel Subsystem Structure (ctK)

ctK manages the PPT. PPT is essentially identical to a regular Linux 4-level DRAM page table, except (1) it is persistent, and (2) it uses relative addresses for both virtual and physical addresses. It uses relative addresses because ctFS's memory region may be mapped to different starting virtual addresses in different processes

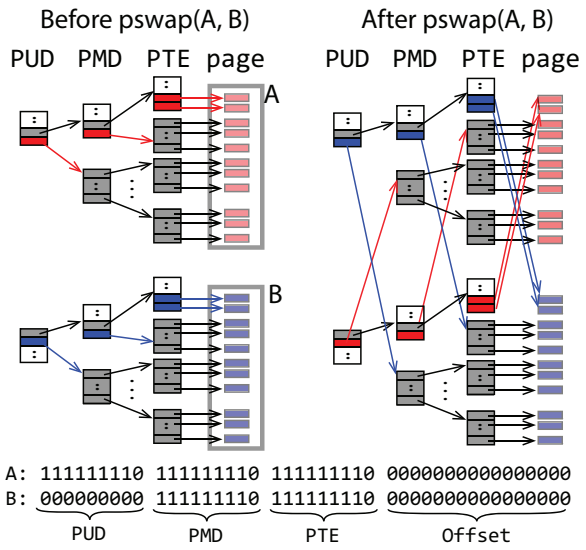


Figure 5: **An example of `pswap`.** The shaded entries in the page tables are the ones used to map the two-page arrays A and B. The red and blue page table entries are the ones that are modified by `pswap`. Before `pswap`, A maps to the red pages and B maps to the blue pages, whereas after `pswap` A maps to blue pages and B maps to red pages. The last 39 bits of A and B's address are shown at the bottom.

due to *Address Space Layout Randomization* [6] [9], and hardware reconfiguration could change PM's starting physical address. We also note that whereas each process has its own DRAM page table, ctK has a single PPT that contains the mapping of all virtual addresses in ctU's memory range (i.e., those inside the partitions). The PPT cannot be accessed by the MMU, so mappings in the PPT are used to populate entries in the DRAM page table on demand as part of page fault handling.

ctK provides a `pswap` system call that atomically swaps the mapping of two same-sized contiguous sequences of virtual pages in the *PPT*. It has the following interface:

```
int pswap(void* A, void* B, unsigned int N,
    int* flag);
```

`A` and `B` are the starting addresses of each page sequence, and `N` is the number of pages in the two sequences. The last parameter `flag` is an output parameter. Regardless of its prior value, `pswap` will set `*flag` to 1 if and only if the mappings are swapped successfully. ctU sets `flag` to point to a variable in the redo log stored on PM and uses it to decide whether it needs to redo the pswap upon crash recovery. `pswap` also invalidates all related DRAM page table mappings (and shoots them down in TLBs), as we found it is more efficient than updating the mappings.

The `pswap()` system call *guarantees crash consis-*

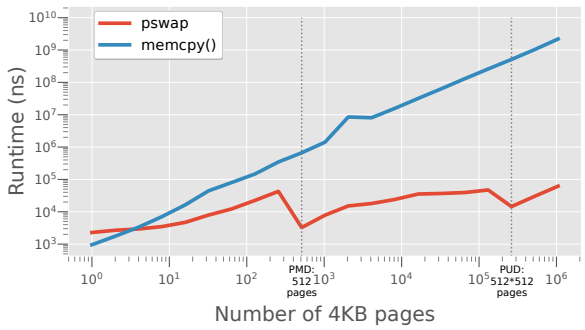


Figure 6: **Comparing the performance of `pswap` and `memcpy`.** Both the X and Y axis are log scale.

tency: it is atomic, and its result is durable as it operates on PPT. Moreover, concurrent `pswap()` operations occur as if they are serialized, which *guarantees isolation* between multiple threads and processes. [5]

`pswap` avoids swapping every target entry in the PTEs (the last level page table) of the PPT whenever possible. Figure 5 shows an example where `pswap` needs to swap two sequences of pages - A and B - each containing 262,658 ($512 \times 512 + 512 + 2$) pages. `pswap` only needs to swap 4 pairs of page table entries or directories (as shown in red and blue colors in Figure 5), as all 262,658 pages are covered by a single PUD entry (covering 512×512 pages), a single PMD entry (covering 512 pages), and two PTE entries (covering 2 pages).

`pswap()` can only perform this optimization if the starting addresses of the two page sequences are *swap-aligned*. We first define the *reach* of a page table *level* to be the size of the memory region that each entry maps — e.g., the reach of PTE, PMD, PUD, and PGD are 4K (bytes), 2M, 1G, and 512G, respectively. Given two contiguous sequences of pages in virtual memory that start at addresses A and B, and given that each sequence spans a memory region of size S, let L be the highest level in the page table such that $reach(L) \leq S$. We then say that the two page sequences A and B are swap-aligned if and only if:

$$A \bmod reach(L) = B \bmod reach(L)$$

In the example of Figure 5, L is PUD, and $reach(L)$ is 1G (2^{30}). $A \bmod reach(L)$ equals $B \bmod reach(L)$ because the last 30 bits of A and B are the same.

Figure 6 shows the performance of `pswap` as a function of the number of pages that are swapped. We compare it with the performance of the same swap implemented with `memcpy` that approximates the use of conventional write ahead or redo logging that requires copying *data* twice. The `pswap` curve shows a wave-like pattern: as the number of pages increases, pswap latency first increases and then drops back as soon as it can swap one entry in a higher-level page table instead of 512 entries in the lower-level table. The two drop points in Figure 6 are when N is 512 (mapped by a single PMD entry) and 262,144 (mapped by a single PUD entry). In comparison, `memcpy`'s latency increases linearly with the number of pages. When N is 1,048,576 (representing 4GB of memory), `memcpy` takes 2.2 seconds, whereas `pswap` takes only 62μs. However, when N is less than 4, `memcpy` is more efficient than `pswap`.

[5] `pswap` uses conventional redo log to ensure crash consistency.

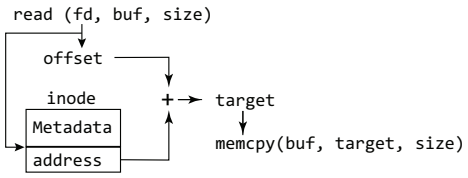


Figure 7: **Implementing `read()` on ctFS.**

Concurrent invocations to `pswap`() will only be serialized if they operate on overlapping memory ranges. We use a binary search tree to store the ranges of concurrent, on-going `pswap`()s.

3.4 File System Operations

Since files are contiguous in virtual memory, read and write operations require special treatment. Other operations that operate on metadata (i.e., directories and metadata in inodes) are similar to those on conventional file systems.

Figure 7 shows how `read()` is implemented in ctFS. Given the file `offset` (from the file descriptor), ctU locates the inode, and further locates the starting address of the file. It adds `offset` to this starting address, which is the virtual address of the data to be read. Then, it uses a single `memcpy`() to copy the data to the user buffer. *All of these operations are performed by the user space ctU.*

ctFS allocates a partition on-demand on the first write to a file. It always allocates the smallest partition that is big enough to store the write. Later when the file size increases beyond the partition size, ctFS will "upgrade" it to the next higher level partition that can accommodate the file. Also recall that ctFS supports two modes, where strict mode offers atomic writes. Consequently there are two special write cases: one that triggers an upgrade and one that requires atomicity. In the normal case where neither applies, write does not differ from read. Both of the special cases use `pswap`, and in both cases ctU *guarantees that the starting addresses are swap-aligned*. Next, we explain each case.

Write with Upgrade. When a write (append) triggers an

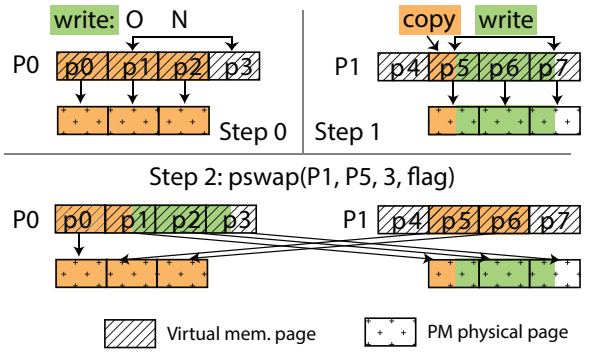


Figure 8: **An example of *atomic write* using `pswap`.** The yellow color indicates the original file content whereas green indicates new data to be written.

upgrade, ctFS will first relocate the file to a new partition before applying the write. It also maintains a redo log to ensure crash consistency of the upgrade. Say a write requires upgrading from a level L partition, P0, to a level M partition, P1 (where $M > L$). ctU first allocates P1 in *virtual memory*. It then calls `pswap (P0, P1, N, flag)`, where N is the number of pages in the level L partition. Note that right after the partition allocation, P1 does not map to any PM pages; therefore, after `pswap()`, P1 points to the PM pages that used to map to P0, and P0 is no longer mapped. Both steps will first be recorded in the redo log, and `flag` is located in the redo log, so if a crash occurs ctU knows whether pswap had completed successfully or not. If a crash happens before `pswap` completes, ctU only needs to free P1. If a crash happens right after `pswap` has completed, then ctU will continue to finish the upgrade by changing the starting address in the file's inode to P1. Partition "downgrades" (e.g., when the file is truncated) are handled in a similar manner.

Atomic Write. In strict mode, ctFS handles an atomic write using a write-and-swap protocol. Assume a write that writes N bytes to offset O of a file in a level L partition, P0. Figure 8 shows an example, where O is *not* page aligned, and N spans three pages where the last page, p3, has not been accessed and is hence not mapped to PM. ctU carries out the following two steps.

Step 1: ctU first allocates a *staging* partition, P1, that is also at level L. It then writes the new data to the *same offset* O in P1. If O is not page-aligned, as is the case in Figure 8, ctU copies the data fragment between the start of the first page and O in P0 to P1, and similarly, it will copy any fragment data at the end if O+N is not page aligned. Note that ctU does *not* need to copy any pages that are not modified.

Step 2: ctU invokes `pswap()` to atomically swap the page sequence in P0 with its corresponding sequence in P1. In Figure 8, it `pswap`s pages p1–p3 in partition P0 with pages p5–p7 in partition P1.

To handle crash consistency, ctU uses the redo log that records the status of each step, and the flag used in `pswap`() is located on this redo log.

3.5 Other Optimizations

Huge Page. ctK allocates huge pages (2MB pages) whenever possible. Because the address of any partition is aligned with the partition size, all files that reside in level L3 or above benefit from huge pages. However, when ctU performs `pswap` with small N, huge pages may have to be broken into base pages, adding extra overhead to `pswap`(). Note that `pswap` can apply its optimization whenever the page sequences are swap-aligned regardless of whether they are huge pages or not. Huge pages are enabled in our evaluation unless otherwise noted. In §4.1.3, we evaluate and explain the impact of huge pages in details.

Optimized append in strict mode. ctFS performs an optimization on append operations by exploring the invariant between a file's metadata and its data [4, 12]. Instead of using the write-and-swap protocol, it directly appends the new data and then changes the file size in the inode. If a crash occurs before the append completes, the file will be consistent, as the file size still has the old value, presenting a view as if the append did not occurr.

Instruction choices in `memcpy()`. We experimented with different memory copy strategies (e.g. repeat instructions, non-temporal instructions, cache flush) and found that AVX512 [1] non-temporal 512-bit load and store (VMOVDQU and MOVNTDQ) performed the best, resulting in a 5%–20% performance gain over what SplitFS and ext4-DAX uses.

3.6 Protection

For protection, ctFS's exloits both Intel Memory Protection Keys (MPK) and regular page table protection. We first explain Intel MPK before discussing ctFS's design.

Background on Intel MPK. MPK allows each memory page to be tagged with one of 16 protection keys, K0, K1, ..., K15. Four unused bits in each page table entry are used to store the key for the page. Each key's protection rights can be changed from user space, *using a special instruction*. For example, key K0's right can be set to no access, K1 can be set to read only, and K2 can be set to read/write. The access rights associated with each key are stored in a register called `PKRU`. Hence the access rights are thread-local (as each core has its own `PKRU` register).

A key advantage of using MPK over the conventional

`mprotect()` system call is performance. While assigning a key to a page still requires a system call, setting/changing the access permission associated with each key is a user-space instruction that only consumes around 20 cycles [33].

Protection in ctFS. ctFS tags each page within ctFS's memory region with a single MPK key, which we refer to as NONE. When a ctFS operation is invoked, ctU sets the access right for the NONE key to be read/write, and it resets the access right back to no access before returning. Therefore, any access to ctFS's memory space from outside of ctFS will be prevented. If multiple processes with different access rights access the same file concurrently, ctFS can protect the same page differently for different processes as the access rights for the same key, NONE, can be set differently on different cores.

This protection strategy protects ctFS against *unintentional* bugs. For example, a dangling pointer in an application won't be able to accidentally corrupt the file system, given that changing the rights associated with the key requires a special instruction. However, this design does not protect against *intentional* attacks. For example, a malicious application could intentionally set the rights for the NONE key to be read/write and modify the file system in an arbitrary manner.

4 Evaluation

In this section, we present the results of evaluating ctFS against other PM file systems (FS) using both real-world applications and microbenchmarks. The server and OS settings used in our evaluation are as described in §2.

4.1 Micro-benchmarks

We evaluate the performance of ctFS and compare it with that of SplitFS, ext4-DAX, PMFS, and NOVA, using the same 6 micro-benchmarks as in §2. We repeat each experiment 10 times and report the average. In all experiments, ctFS uses demand paging and does not prepopulate any pages in order to accentuate the maximum page fault handling overhead in ctFS. SplitFS prefaults staging files at its system bootup time so there are no page faults on those files.

4.1.1 Append

Append is particularly relevant as the append operation is the dominant file system operation of many application [21], and it is the operation on which SplitFS shows the most significant speedup. Figure 9 shows the performance of Append.

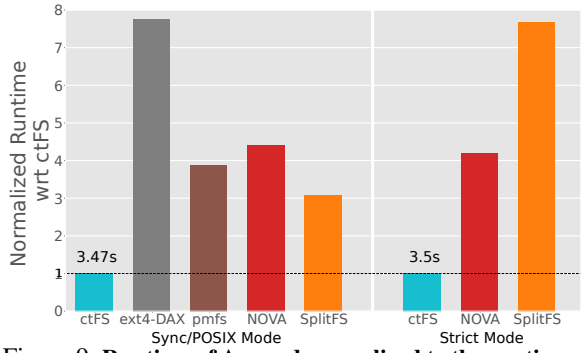


Figure 9: **Runtime of Append normalized to the runtime of ctFS.** The different file system and configuration combinations are grouped by the crash consistency guarantees on file data.

ctFS achieves a 7.7x speedup against ext4-DAX for Append in sync mode. 45% of ext4-DAX's runtime is in building and searching indices as it has to allocate many small extents. Even if we deduct kernel trap overhead (shown in Figure 1) from the runtimes of ext4, ctFS-sync still achieves an 7.0x speedup. This shows the benefit of using contiguous file allocation, regardless of whether it is a user-space or kernel-space implementation.

While SplitFS is able to significantly reduce the indexing time by using memory-mapped I/O, ctFS still achieves 3.1x speedup over SplitFS in sync mode. Specifically, SplitFS takes 7.2s longer than ctFS to run Append, and 92% (6.62s) of that time comes from indexing. The other 8% of the speedup comes from ctFS's improved I/O performance. In contrast, ctFS successfully eliminates most of the overhead of file indexing, primarily by having the MMU perform the indexing in hardware during memory page translation. (See Figure 11 for a breakdown of ctFS's runtime.) It only spends 24ms in page fault handling, compared to SplitFS's 4.4s of page fault handling (§2). Even though ctFS has a similar number of page faults (525,490) as SplitFS (578,260), SplitFS triggers page faults whose handling requires file indexing, whereas all of ctFS's page faults are minor faults.

For the Append workload, whether running in sync or strict mode does not affect ctFS performance because of ctFS's append optimizations (§3.5); ctFS achieves 7.66x speedup over SplitFS in strict mode.

Compared to NOVA's sync mode and pmfs, ctFS-sync achieves 4.4x and 3.87x speedups, respectively.

4.1.2 Other Micro-benchmarks

Figure 10 shows ctFS's performance compared to that of ext4-DAX and SplitFS for the other 5 microbenchmarks. In sync mode, ctFS achieves an average speedup of

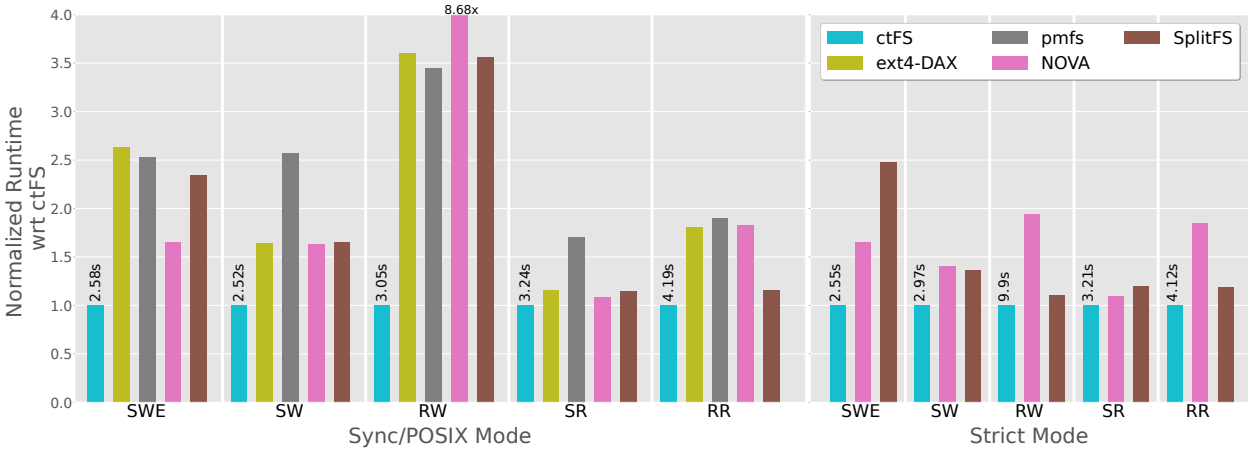


Figure 10: **Performance comparison** of ext4-DAX, Nova, PMFS, SplitFS, and ctFS for five micro-benchmarks.

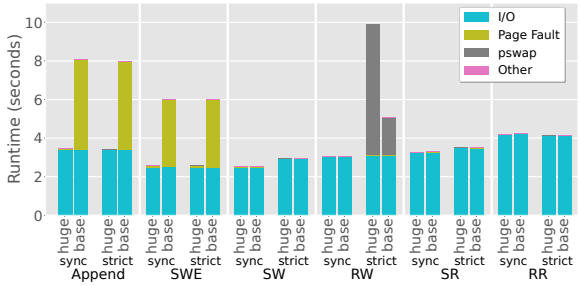


Figure 11: **ctFS overhead breakdown** under four configuration combinations: with huge pages enabled and disabled, while running in sync and strict mode.

2.17x, 1.97x, 2.43X, 2.97X, against ext4-DAX, SplitFS-sync, pmfs, and NOVA, respectively. In strict mode, ctFS achieves an average speedup of 1.46x and 1.59X against SplitFS-strict and NOVA-strict.

4.1.3 ctFS Runtime Breakdowns

Figure 11 shows the breakdown of ctFS's runtime on each test while running in sync and strict mode, and with huge pages enabled and disabled. We first consider the difference between ctFS's sync and strict modes. Recall that ctFS invokes `pswap` at the end of file overwrite operations under strict mode. This affects both RW and SW. In RW, 68% of the run time of ctFS-strict is spent on `pswap`. This test represents the worst-case scenario for ctFS-strict, as each write triggers a `pswap` at the smallest granularity (4KB page): `pswap` cannot perform any optimizations when swapping the entries in the last-level page table, and it is foreced to break up the huge pages into base pages.[6] In comparison, while ctFS also needs to invoke `pswap` in SW when running in strict mode, because `pswap` is only invoked once at the end, it incurs negligible overhead (5.7ms).

The figure also shows the difference between having huge pages enabled and disabled. With huge pages enabled, ctFS indeed eliminates much of the indexing overhead, as all workloads are bottlenecked by I/O, except for the RW workload when ctFS runs in strict mode. With huge pages disabled, both the persistent page table (PPT) and the DRAM page table have 512x more entries, and each TLB entry now only maps 4KB instead of 2MB. For SW, RW, SR, and RR, the overhead after disabling huge pages is negligible in both sync and strict modes (at most 3.4% in SR-strict). This indicates that the overhead of additional TLB misses is negligible. In RR, for example, there are 512x more TLB misses with huge pages disabled, yet this still results in negligible overhead. Note that the number of page faults remains the same even when huge pages are disabled, because ctK copies 512 page table mappings (or the mappings for a 2MB region) from the PPT to the DRAM page tables on each page fault. In comparison, the large overheads in Append and SWE come from allocating physical PM page frames and building the persistent page tables (PPT), because with only base pages, the PPT contains 512x more entries.

Interestingly, in RW, disabling huge pages results in a 2x speedup for ctFS-strict. This is because with huge pages enabled, every write, which is at the granularity of a base page (4KB), will trigger a `pswap` that breaks the huge page and causes TLB shootdowns. In comparison, when huge pages are disabled, there is no need to break up huge pages.

4.2 Real-world Applications

We evaluated ctFS using LevelDB [28] and RocksDB [35], both of which are persistent key-

[6] Even then, ctFS outperforms SplitFS and NOVA in strict mode as shown in Figure 10. SplitFS also uses huge pages, so that it also needs to break up huge pages, which makes up 37.6% of its runtime.

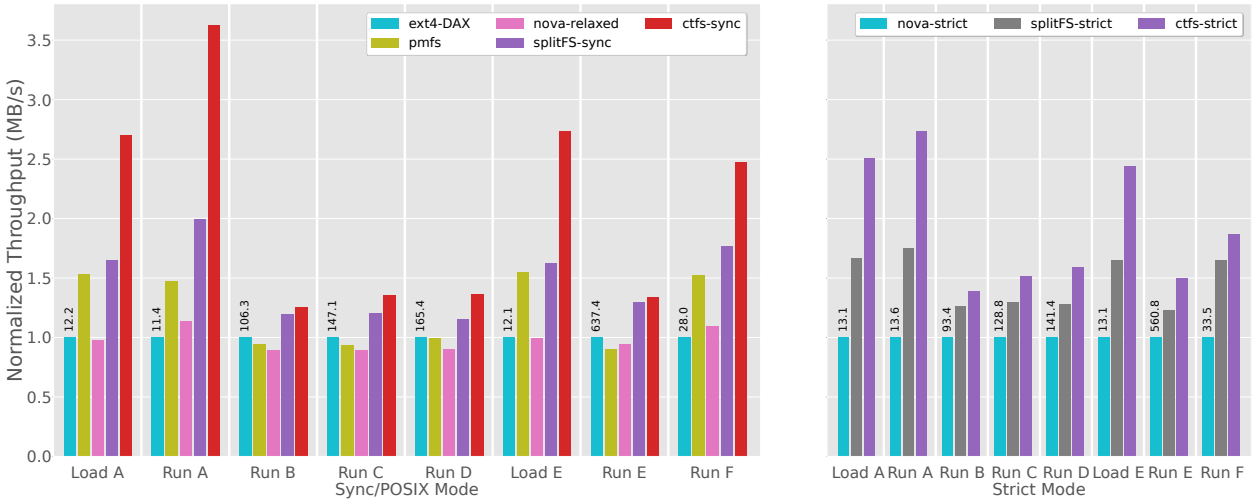


Figure 12: **YCSB on LevelDB.** Results are measured in throughput that is normalized to ext4-DAX in the sync/POSIX group and NOVA-strict in the strict group. The number on top shows the absolute throughput.

A	Update heavy: 50/50 read/write mix
B	Read mostly: 95/5 read/write mix
C	Read only: 100% read
D	Read latest: new records are inserted, and the most recently inserted records are read the most
E	Short ranges: short ranges of records are queried, instead of individual records
F	Read-modify-write: read a record, modify it, and write back the changes

Table 2: YCSB runs and their characteristics.

value stores. We drove LevelDB with the Yahoo! Cloud Serving Benchmark (YCSB) [5]. YCSB includes six different key-value workloads that are described in Table 2. We drove RocksDB using RocksDB's built-in benchmark `db_bench` with three workloads: *random fill*, which creates and adds key-value pairs; *random read*, which returns the values of given keys; and *random update*, which updates the values of given keys. Each of these tests carries out 5 million operations. Both LevelDB and RocksDB use `pwrite` and `pread` instead of memory-mapped I/O.

The LevelDB workloads demonstrate ctFS's performance advantage achieved by removing the indexing overheads in a real world application. The RocksDB workloads show that it is feasible and beneficial to replace write-ahead logs (WALs) with ctFS's atomic writes.

LevelDB. Figure 12 shows the performance of different PM file systems on LevelDB using the YCSB workloads. ctFS outperforms all the other file systems in each of the workloads when run at comparable consistency levels.

ctFS achieves the most significant improvement in throughput under write-heavy workloads: Load A and E and Run A, B, F. [7] Among these write-heavy workloads, ctFS-sync's throughput is 1.64x the throughput of SplitFS-sync on average, with 1.82x the throughput in the best-case (under Load E). In strict mode, ctFS's throughput is 1.30x higher than that of SplitFS on average, with 1.50x higher in the best-case (under Load A). Compared with ext4-DAX, ctFS-sync has 2.88x higher throughput on average and 3.62x higher throughput in the best case (under Run A).

On read-heavy workloads, ctFS's thoughtput is still higher than that of the other file systems, but by a smaller amount. It achieves an average of 1.25x - 1.36x higher throughput over ext4-DAX. As for SplitFS, recall from our microbenchmarks that it spends more time on indexing in random reads than sequential reads. This is why ctFS achieves better throughput than SplitFS in Run B, C, and D, which are dominated by random reads; e.g., ctFS's throughput is 1.18x and 1.25x higher than that of SplitFS under Run D when run in either sync or strict mode. For Run E, which is dominated by sequential reads, cfFS has 1.02x and 1.22x higher throughput.

By studying the breakdowns of ext4-DAX's time consumption, we observe that indexing time takes up 19.6%, 25%, and 24.5% of the total runtime of LoadA, RunA, and LoadE, respectively. Meanwhile, ctFS only spends a maximum of 0.2% on indexing overhead (in handling page faults) across all workloads. Hence, indexing accounts for 39.3%, 49.9%, and 46.4% of ctFS's speedup over ext4-DAX on these three workloads. Another 22.5%, 36.4%, and 33% of ctFS's speedups arise from a more efficient I/O data path over ext4-DAX.

[7] Load A and Load E create the respective key value stores that are used by the six YCSB runs.

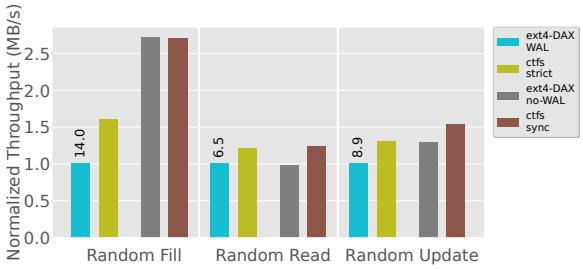


Figure 13: **RocksDB performance.**

	SplitFS		Ext4-DAX	ctFS
	sync	strict		
Bootstrap (µs)	1.4×10^6	1.1×10^6	0	5
open (create) (µs)	40	40	15	2
open (existing) (µs)	4	10	4	2
unlink (µs)	32	43	31	1.6
DRAM usage (MB)	198	572	N/A	0.52
Space available after format (GB)	230.7		230.7	248.1
Space used after YCSB LoadA (MB)	5486		5337	5378

Table 3: **Metadata operation and resource overhead.** There is no difference on between sync and strict modes for ctFS.

RocksDB. We ran our RocksDB experiments two configurations: *strict* and *relaxed*. With *strict*, where data consistency is guaranteed, ext4-DAX is run with WAL enabled, and ctFS is run in strict mode (ctFS-strict) but with WAL disabled. With *relaxed*, where crash consistency is not guaranteed, both ext4-DAX and ctFS-sync are run with WAL disabled.

With *strict*, ctFS-strict has 1.60x, 1.22x and 1.3x the throughput of ext4-DAX for the Random Fill test, the Random Read test and the Random Update test, respectively. This demonstrates the feasibility of replacing WALs in applications with atomic writes in ctFS, as doing so not only improves performance but also simplifies application logic.

With *relaxed*, ctFS-sync is on par with ext4-DAX with the Random Fill test, but has 1.25x and 1.19x the throughput for the Random Read and Random Update test.

4.3 Resource Usage & Other Operations

Table 3 shows the cost of several frequently used file system operations, as well as DRAM overhead after filesystem initialization and space efficiency for ctFS, SplitFS and Ext4-DAX. Notably, SplitFS spends over one second to initialize because it needs to build the U-Split mapping table, create and `mmap` all the staging files. Similarly, because of the mapping table, SplitFS uses 3 orders of magnitude more DRAM comparing with ctFS. The DRAM usage does not change significantly for SplitFS

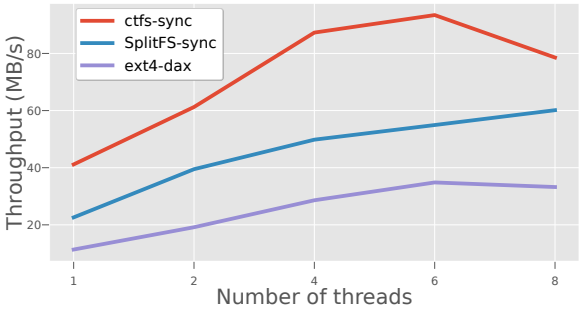


Figure 14: **Scalability of ctFS** versus ext4-DAX and SplitFS on LevelDB running YCSB Run A in terms of throughput.

and ctFS when running workloads as both of them primarily operate on PM.

In terms of space efficiency, ctFS has 7.52% more available space than ext4-DAX and SplitFS when newly formatted. In fact, ctFS only incurs 10MB memory overhead for newly formated 248.06GB space. This is because ctFS allocates inodes and inode bitmaps on demand. After running Load A in the YCSB test on LevelDB, ctFS occupies 0.78% more space than ext4-DAX and 2% less than SplitFS.

4.4 Scalability

The design of the cfFS's concurrency model is the same as that of ext4-DAX. Figure 14 shows ctFS's scalability compared with ext4-DAX, running YCSB Run A on LevelDB. All file systems scale similarly. Performance of ctFS peaks at 6 worker threads in a 8 core machine (as two additional threads are spawned by LevelDB for other purpose).

5 Limitations and Discussion

The design of ctFS presents two unique trade-offs. First, compared with an in-kernel file system, ctFS's user-space file system design trades protection for performance. While ctFS is not suitable as a general purpose file system, it presents a (rather extreme) trade-off point for data center applications to squeeze the most out of the hardware, as in data center environments each machine runs only a small number of applications that often trust each other, and protection against intentional attacks is ensured at the boundary of machines or data centers. Furthermore, ctFS can be used as an application's private file system, i.e., where one or several applications own one instance of ctFS.

Second, ctFS's design is also at the expense of internal fragmentation within each fixed-sized partition in the *virtual* memory address space. We argue this is acceptable

given the size of today's virtual address spaces. Both Intel and Linux now support 57 bits virtual addresses with 5-level paging, enabling a 128PB virtual address space. In comparison, the maximum size of Intel Optane DC that can be supported by a server today is 6TB [27]. Note that ctFS does not waste physical storage space as any unused regions of a partition are not mapped to physical memory. In addition, ctFS's allocation algorithm is similar to the buddy memory allocator, and hence, the internal fragmentation problem is fundamentally inline with that of modern size-segregated memory allocators like jemalloc [10], TCMalloc [13], and Go's runtime [16]; the wide adoption of these allocators further suggests that internal fragmentation is an acceptable trade-off.

6 Related Work

To the best of our knowledge, this paper is the first to propose a complete file system that supports contiguous files with a detailed design and evaluation.

SCMFS. SCMFS [39] proposed the high-level idea of allocating each file contiguously in the virtual address space. However, its design is only at a conceptual level. How files are allocated in the virtual memory space is not clearly described. Specifically, it does not address file resizing and external fragmentation, the two fundamental challenges faced by contiguous files. It is unclear what happens if one file expands into the range of another file. Finally, SCMFS's implementation and evaluation are entirely based on simulation.

File systems for PM. A number of file systems were designed to bypass the kernel. Aerie [37], PMFS [8], Strata [25], SplitFS [21], and ZoFS [7] all allow the user to directly access file data through a user-space component; PMFS, SplitFS, and ZoFS map the metadata and data in application's virtual memory space. In Aerie, metadata updates and locking requests must be sent via IPC to be processed by a trusted system service. Strata logs updates in userspace which are then digested in the kernel. ZoFS strives to provide security by only mapping the metadata to the users who have access permission, and only allows trusted library code to modify the metadata by exploiting MPK memory protection keys. KEVIN [24], a file system for NAND SSD instead of PM, provides an FPGA implementation of the log-structured merge tree, and ports file operations on top.

All of the file systems mentioned above still use a tree-structured index for file indexing. B*e*trFS proposes a B^{ε}-tree that is a write-optimized variant B-tree [20]. HashFS [30] uses a global fixed-sized hash table for indexing. However, it still suffers software indexing overhead, and its performance is no better when compared to SplitFS. KUCO [3] offloads some indexing from the kernel to the userspace through "collaborative indexing", to improve scalability. However, it still uses traditional ext2-style block mapping. In comparison, ctFS uses a contiguous file design that obsoletes file indexing.

Crash consistency on file data. Conventional write-ahead logging/journaling [15, 17, 38] typically requires writing the data *twice*: first to journal before updating the target file. The cost of double-write for data may be large, and several mechanisms that avoid data copying have been proposed [2, 4, 18, 26, 29]. Similar to `pswap`, SplitFS's relink is used to efficiently provide atomic writes without copying the data to the journal. `pswap` differs from `relink` in that the former swaps the virtual-to-physical memory mapping, whereas `relink` changes the mapping within ext4-DAX's extent trees. Failure atomic msync [32] atomically commits changes to a memory mapped file by using ext4's journalling function. SHARE [31] atomically lets pairs of pages share the same physical page in the flash storage. It does not explore the page table hierarchy for optimization.

SubZero [22] proposed a `patch()` function that atomically overwrites the destination region of a mmap file with the content of the source region. `pswap` is different in a few ways. First, `pswap` swaps the mapping whereas `patch` discards the content in the source region. In addition, `pswap` leverages the page table hierarchy to achieve significant speedup. Finally, `pswap` is mainly used for fast cross-partition expansion and shrink, whereas `patch` is only used for atomic writes.

7 Concluding Remarks

This paper proposes ctFS, a persistent memory file system which offloads file system indexing to the memory management hardware by keeping files contiguous in virtual memory. Our evaluation shows ctFS can outperform ext4-DAX and SplitFS by up to 7.7x and 3.1x, and improve YCSB throughputs by up to 3.6x and 1.8x.

Acknowledgements

We thank our shepherd Youngjin Kwon and the anonymous reviewers for their insightful comments. SplitFS authors provided valuable help in obtaining the breakdown of SplitFS's runtime. Rishikesh Devsot and Devin Gibson have ported an early version of ctFS to boot into a Linux shell without hitting Linux's file system. This research was supported by the Canada Research Chair fund, an NSERC discovery grant, and a VMware gift.

References

[1] Intel AVX-512 instructions. https://software.intel.com/content/www/us/en/develop/articles/intel-avx-512-instructions.html.

[2] J. Bonwick and B. Moore. ZFS: The last word in file systems. https://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf.

[3] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu. Scalable persistent memory file system with Kernel-Userspace collaboration. In *Proc. 19th Conf. on File and Storage Technologies*, pages 81–95. USENIX Association, Feb. 2021.

[4] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. 22nd Symp. on Operating Systems Principles*, SOSP'09, pages 133–146, 2009.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st Symp. on Cloud Computing*, SoCC'10, pages 143–154, 2010.

[6] corbet. Address space randomization in 2.6. url-https://lwn.net/Articles/121845/, 2005.

[7] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen. Performance and protection in the ZoFS user-space NVM file system. In *Proc. 27th Symp. on Operating Systems Principles*, SOSP'19, pages 478–493, 2019.

[8] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proc. 9th European Conf. on Computer Systems*, EuroSys'14, New York, NY, USA, 2014. Association for Computing Machinery.

[9] J. Edge. Randomizing the kernel. url-https://lwn.net/Articles/546686/, 2013.

[10] J. Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the BSDCAN Conf., Ottawa, Canada*, 2006.

[11] A. S. Fedorova. Why mmap is faster than system calls. https://sasha-f.medium.com/why-mmap-is-faster-than-system-calls-24718e75ab37, 2019.

[12] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2):127–153, May 2000.

[13] S. Ghemawat and P. Menage. TCMalloc. http://goog-perftools.sourceforge.net/doc/.

[14] M. Gorman. Page table management. https://www.kernel.org/doc/gorman/html/understand/understand006.html.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[16] R. Griesemer, R. Pike, and K. Thompson. Golang. https://golang.org/.

[17] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. 11th ACM Symp. on Operating Systems Principles*, SOSP'87, pages 155–162, 1987.

[18] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. of the USENIX Winter 1994 Technical Conference*. USENIX Association, 1994.

[19] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the Intel Optane DC Persistent Memory. https://arxiv.org/abs/1903.05714v3, 2019.

[20] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. BetrFS: Write-optimization in a kernel file system. *ACM Trans. Storage*, 11(4), Nov. 2015.

[21] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proc. 27th Symp. on Operating Systems Principles*, SOSP'19, pages 494–508, 2019.

[22] J. Kim, Y. J. Soh, J. Izraelevitz, J. Zhao, and S. Swanson. Subzero: Zero-copy io for persistent main memory file systems. In *Proc. 11th Asia-Pacific Workshop on Systems*, APSys'20, pages 1–8, New York, NY, USA, 2020. Association for Computing Machinery.

[23] K. C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, October 1965.

[24] J. Koo, J. Im, J. Song, J. Park, E. Lee, B. S. Kim, and S. Lee. Modernizing file system through in-storage indexing. In *Proc. 15th Symp. on Operating Systems Design and Implementation*, OSDI'21, pages 75–92. USENIX Association, July 2021.

[25] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A cross media file system. In *Proc. 26th Symp. on Operating Systems Principles*, SOSP'17, pages 460–477, 2017.

[26] E. Lee, H. Bahn, and S. H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proc. 11th Conf. on File and Storage Technologies*, FAST'13, pages 73–80, 2013.

[27] Lenovo. Intel Optane Persistent Memory 100 Series Product Guide. https://lenovopress.com/lp1066-intel-optane-persistent-memory-100-series#mixed-mode-requirements.

[28] LevelDB. https://github.com/google/leveldb.

[29] C. Mohan. Repeating history beyond ARIES. In *Proc. 25th Intl. Conf. on Very Large Data Bases*, VLDB'99, pages 1–17. Morgan Kaufmann Publishers Inc., 1999.

[30] I. Neal, G. Zuo, E. Shiple, T. A. Khan, Y. Kwon, S. Peter, and B. Kasikci. Rethinking file mapping for persistent memory. In *Proc. 19th Conf. on File and Storage Technologies*, FAST'21, pages 97–111. USENIX Association, Feb. 2021.

[31] G. Oh, C. Seo, R. Mayuram, Y. Kee, and S. Lee. SHARE interface in flash storage for relational and NoSQL databases. In *Proc. 2016 Intl. Conf. on Management of Data*, SIGMOD'16, pages 343–354, 2016.

[32] S. Park, T. Kelly, and K. Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proc. 8th European Conf. on Computer Systems*, EuroSys'13, pages 225–238, 2013.

[33] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. Libmpk: Software abstraction for Intel memory protection keys (Intel MPK). In *Proc. 2019 Usenix Annual Technical Conf.*, USENIX-ATC'19, pages 241–254, 2019.

[34] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[35] Direct I/O - RocksDB Wiki (accessed 2019-03-05). https://github.com/facebook/rocksdb/wiki/Direct-IO.

[36] A. Tanenbaum and H. T. Boschung. *Modern operating systems*. Pearson, 2018.

[37] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proc. 9th European Conf. on Computer Systems*, EuroSys'14, pages 14:1–14:14, 2014.

[38] D. Woodhouse. JFFS : The journalling flash file system. In *Proc. Ottawa Linux Symposium*. RedHat Inc., 2001.

[39] X. Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proc. 2011 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC'11, pages 1–11, 2011.

[40] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. 14th Conf. on File and Storage Technologies*, FAST'16, pages 323–338, Santa Clara, CA, 2016.

FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory

Ming Zhang, Yu Hua*, Pengfei Zuo, Lurong Liu
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
**Corresponding Author: Yu Hua (csyhua@hust.edu.cn)*

Abstract

Persistent memory (PM) disaggregation improves the resource utilization and failure isolation to build a scalable and cost-effective remote memory pool. However, due to offering limited computing power and overlooking the bandwidth and persistence properties of real PMs, existing distributed transaction schemes, which are designed for legacy DRAM-based monolithic servers, fail to efficiently work in the disaggregated PM architecture. In this paper, we propose FORD, a **F**ast **O**ne-sided **R**DMA-based **D**istributed transaction system. FORD thoroughly leverages one-sided RDMA to handle transactions for bypassing the remote CPU in PM pool. To reduce the round trips, FORD batches the read and lock operations into one request to eliminate extra locking and validations. To accelerate the transaction commit, FORD updates all the remote replicas in a single round trip with parallel undo logging and data visibility control. Moreover, considering the limited PM bandwidth, FORD enables the backup replicas to be read to alleviate the load on the primary replicas, thus improving the throughput. To efficiently guarantee the remote data persistency in the PM pool, FORD selectively flushes data to the backup replicas to mitigate the network overheads. Experimental results demonstrate that FORD improves the transaction throughput by up to 2.3× and reduces the latency by up to 74.3% compared with the state-of-the-art systems.

1 Introduction

Memory disaggregation, which decouples the compute and memory resources from the traditional monolithic servers into independent compute and memory pools, has gained extensive interests in both industry [11, 14, 42] and academia [1, 20, 47, 57, 72]. By efficient resource pooling, the resource utilization, elasticity, failure isolation, and heterogeneity are significantly improved in datacenters [45]. The compute pool runs programs with a small DRAM buffer, and the memory pool stores application data with weak compute units only for memory allocations and interconnections [72]. Fast networks, e.g., RDMA, are generally adopted to connect the compute and memory pools [57]. Recently, the persistent memory (PM) is available on the market [12], which exhibits non-volatility and low latency with high density and low costs [67]. Hence, the efficient use of PM becomes important to build a persistent, large, and cost-effective disaggregated PM pool [54].

To ensure that the data are atomically and consistently accessed in the PM pool, the compute pool is required to leverage distributed transactions (dtxns) to read/write the remote data. However, existing RDMA-based dtxn systems are designed for traditional monolithic servers, in which each server hosts the CPU and DRAM resources. These systems fail to work on the disaggregated PM, since the PM pool does not contain CPUs to frequently handle extensive computation tasks during dtxn processing, e.g., concurrency control in HTM [9, 61], data retrieving [60], locking [19, 29, 44], and busy buffer polling [18]. Moreover, legacy systems do not consider the bandwidth and persistence properties of real PM, leading to low throughputs and inconsistent remote writes. To run dtxns on the disaggregated PM, an intuitive solution is to leverage one-sided RDMA to bypass the CPU in PM pool. However, we observe that using one-sided RDMA in existing dtxn systems incurs substantial round trips and access contentions, which significantly decrease the performance. It is non-trivial to design a high-performance dtxn system for the disaggregated PM due to the following challenges:

1) Long-latency processing. Legacy systems adopt the optimistic concurrency control (OCC) [32] to serialize dtxns, and the primary-backup replication for high availability. OCC is efficient for read-only dtxns due to no locks on read-only data. However, for the read-write dtxns, the data in read-write set consume 3 round trips to be read, locked, and validated before writing remote replicas, thus heavily increasing the latency. Furthermore, to ensure that the dtxn can roll forward once the primary fails, prior designs consume 2 round trips to write remote replicas, i.e., writing redo logs to backups and updating primaries, which however delays the dtxn commit.

2) Limited PM bandwidth on the primary. When using the primary-backup replication, legacy systems only allow the primary to be read, since the newest data in backups are still stored in redo logs after the dtxn commits. Hence, all the

RDMA read/write requests are issued to the primary to be handled. However, the PM DIMM suffers from lower write bandwidth (e.g., 12.9 GB/s of six interleaved 256GB PM DIMMs [67]) than recent RDMA-capable NICs or RNICs (e.g., 25GB/s for a dual-port ConnectX-5 RNIC [52]). The substantial RDMA reads saturate PM bandwidth and further block write requests. As a result, the primary's PM becomes a performance bottleneck, which decreases the throughput.

3) Lack of remote persistency guarantee. Existing DRAM-based systems overlook the persistence property of PM. When issuing RDMA writes to the PM pool, the data are cached in RNIC but not immediately persisted to PM. Hence, the remote persistency [17, 23] is not guaranteed, which possibly causes the remote data to be lost or partially updated once a crash occurs in the PM pool, leading to data inconsistency. Therefore, it is important to ensure the remote persistency in dtxn processing with low network overheads.

Existing studies do not efficiently address these challenges on disaggregated PM. FaSST [29] uses the remote procedure call (RPC) to reduce round trips, but RPC requires the CPU in PM pool to frequently query, lock and update data. DrTM+H [60] employs hybrid RDMA verbs to improve performance, but the two-sided RDMA fails to work in the PM pool due to consuming the remote CPU. NAM-DB [68] decouples compute and storage servers to run dtxns. It adopts snapshot isolation and operation logs without checkpointing to disks. The data are not replicated, thus hurting the availability. After commit, the inputs, descriptions, and timestamps of dtxns are recorded in operation logs. Once the operation logs fill up the memory, the system cannot serve writes. NAM-DB works on DRAM and disks, which is not designed for PM.

To tackle the above challenges, we propose FORD, a **F**ast **O**ne-sided **R**DMA-based **D**istributed transaction system. Unlike prior systems, FORD fully leverages one-sided RDMA to process dtxns for the new disaggregated PM architecture with efficient round trip reductions and PM-conscious designs. Specifically, this paper makes the following contributions:

• ***Hitchhiked Locking*** **and** ***Coalescent Commit*** **to reduce latency.** FORD efficiently attaches the locks with read requests in a hitchhiker manner, to read remote data that belong to the *read-write set* in a single round trip during the dtxn execution phase. Hence, it is unnecessary to consume extra round trips for locking and validations after the execution phase (§ 3.2). Furthermore, FORD leverages a coalescent commit scheme to *in-place update all the primaries and backups* in a single round trip to accelerate commit. To ensure that the dtxn can roll back once the replica crashes, FORD writes undo logs in parallel with the dtxn execution. To prevent the updated data from being partially read, FORD temporarily marks the data to be invisible in the commit round trip. After commit, the data are made visible in the background, which consumes at most 0.5 round trip time (§ 3.3).

• ***Backup-enabled Read*** **to release the PM bandwidth on the primary replicas.** FORD allows the backups to serve the read requests, thus freeing up the PM bandwidth in the primary to serve other requests. Since the backups are in-place updated by using our coalescent commit scheme, the compute pool can easily read the newest data from the backups after the dtxn commits. By balancing the load on the primaries and backups, FORD eliminates the performance bottleneck on the primary to improve the throughput (§ 3.4).

• ***Selective Remote Flush*** **to guarantee remote persistency with low overheads.** FORD leverages one-sided RDMA flush schemes to persist the written data from remote RNIC cache to PM for remote persistency. However, flushing each RDMA `WRITE` to each remote replica incurs substantial round trips. To avoid this, FORD selectively issues the flushes only after the final write and to the backups. Since the $(f+1)$-way primary-backup replication tolerates at most f replica failures, once the updates are persistently stored in the f backups, the remote persistency is guaranteed. Hence, FORD significantly reduces the remote flush operations (§ 3.5).

2 Background and Motivation

2.1 Disaggregated Persistent Memory

Traditional datacenters consist of a collection of monolithic servers, each of which hosts compute units and memory modules. However, such an architecture suffers from low resource utilization, poor elasticity, and coarse failure domain [57]. For example, even if only more CPU cores are needed, we have to add more servers that waste the memory/storage capacities.

To address these drawbacks, memory disaggregation decouples the compute and memory resources from monolithic servers to independent and RDMA-connected resource pools, in which each compute and memory pool is flexibly deployed and scaled, thus improving the resource utilization, elasticity, and failure isolation [72]. The compute pool contains substantial compute blades (e.g., CPU cores) to execute applications with a small memory as cache. The memory pool consists of many memory blades (e.g., DRAM DIMMs) to store the application data, and contain weak compute units only for memory allocations and network interconnections [57, 72].

The memory pool does not guarantee data persistence when using DRAM as memory blades. Plugging UPS [19, 61] adds "non-volatility" on DRAM, which however increases the costs and energy consumptions. If a power failure occurs, the data in DRAM are flushed to disks with the support of UPS, which incurs I/O overheads. Moreover, it is hard to increase the capacity of one DRAM DIMM due to the limited scalability [53], causing high costs to build a large memory pool.

Persistent memory (PM) addresses the above issues by providing persistence, high density (e.g., 512 GB/DIMM [13]), and low costs (e.g., 39.2% $/GB of DRAM [3]), while exhibiting DRAM-like latency [67]. As memory disaggregation meets the needs of datacenters, disaggregating PM also enjoys the same benefits [54]. Hence, we leverage PM as memory blades to build the disaggregated persistent memory (DPM), which forms a persistent and cost-effective memory pool.

2.2 RDMA-based Distributed Transactions

Due to the benefits of bypassing remote CPU and traditional TCP/IP stack, recent studies leverage RDMA to run distributed transactions (dtxns) [19, 29, 44, 60, 61]. Specifically, a coordinator is leveraged to read remote data, run dtxn logic, and commit the updated data back to remote machines. The concurrency control schemes, such as two-phase locking (2PL) [4] and optimistic concurrency control (OCC) [32], are used to serialize dtxns. 2PL acquires locks for all data before execution, and releases all locks after commit. OCC does not lock data during execution, but acquires (or releases) locks for all the written data before (or after) commit. Many systems adopt OCC due to not locking the read-only data, which benefits read-only dtxns. Moreover, the primary-backup replication (PBR) [33] is incorporated in dtxn processing for high availability [19, 60, 70]. The $(f+1)$-way PBR contains 1 primary and f backups for each data shard, and tolerates at most f replica failures. We assume that the fail-stop failures [25] can occur in arbitrary replicas at any time. The failed replica can be quickly detected and recovered by using RDMA [19]. Like FaRM [18, 19, 44], DrTM [9, 60, 61] and FaSST [29], we currently do not consider the byzantine failures [26].

Our paper focuses on the efficient use of OCC and PBR. Fig. 1 presents how existing RDMA systems [19, 60] process dtxns over OCC and PBR. Without loss of generality, we use 2-way replication as an example. In general, there are 5 phases: 1) Execution. A coordinator reads the required data (i.e., read set = {A, B, C}) from primaries and locally executes a dtxn. The updated data (i.e., write set = {A, B}) are buffered in a local cache. 2) Locking. After execution, the coordinator locks the write set in primaries to serialize dtxns. If locking fails, the coordinator aborts the dtxn. 3) Validation. If locking succeeds, the coordinator reads the data versions from primaries to validate that the versions of read and write sets are unchanged. If the validation fails, the coordinator aborts the dtxn. 4) Commit backup. If the validation succeeds, the coordinator sends redo logs to remote backups. 5) Commit primary. After receiving all ACKs from backups, the coordinator updates and unlocks the primaries to commit the dtxn.

2.3 Distributed Transactions on DPM

System Model. In the disaggregated PM architecture, PM is used as remote memory with persistence to durably store the application data (including the primary and backups). The PM pool contains a small number of weak compute units only for memory allocations and RDMA connections during the *initialization* [57, 72]. Afterwards, these compute units are not used during the *execution* since they are too weak to frequently and efficiently handle substantial tasks. Moreover, there is no PM in the compute pool that uses RDMA to access the data stored in remote PMs at the byte granularity (no page swap). To ensure the atomicity, the compute pool uses coordinators to run transactions that read/write data across remote PMs. All transactions are hence distributed, and the

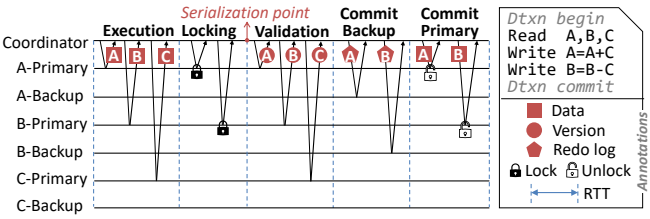


Figure 1: Using OCC and PBR to process dtxns.

replication is accessed by multiple coordinators, which use RDMA to commit each dtxn. Two-sided RDMA-based RPC reduces the network round trips by consuming remote CPUs to handle multiple operations in one round trip [29]. But the PM pool does not contain CPUs to process requests during execution, and RPC fails to work. Hence, the coordinators need to use one-sided RDMA to bypass remote CPUs.

Legacy RDMA-based dtxn systems become inefficient on disaggregated PM since they are not designed for memory disaggregation and real PM. Directly using one-sided RDMA will incur extensive round trips that decrease the performance:

1) As shown in Fig. 1, due to no locks in the execution phase, the intersected data between read and write sets (i.e., read-write set = {A, B}) are operated in execution, locking, and validation phases, which consume 3 round trip times (RTTs) before updating the replicas. In general, the read-write set is equal to the write set, since the data need to be read before being written back [29]. Hence, these round trips widely exist in read-write dtxns, causing extra latency. Moreover, if the locking (or validation) fails, the dtxn aborts, which wastes the execution (or execution+locking) phases. As a result, the coordinator consumes useless round trips before processing the next dtxn, thus decreasing the throughput. DrTM+H [60] merges the locking and validation phases, but still consumes an RTT to validate the read-write set.

2) Fig. 1 shows that existing systems [19, 29, 44, 60] consume 2 RTTs to first write backups (redo logs) and then write primaries (in-place updates) for high availability. By doing so, the dtxn is ensured to commit after receiving all ACKs from backups, since even if the primary fails, the new data can be recovered from redo logs in the backup. In the monolithic architecture, the coordinator can co-locate with a primary or backup, and hence the local commit can save an RTT. But in the disaggregated architecture, the compute pool does not store any replica. Hence, each read-write dtxn inevitably spends 2 RTTs to commit, which incurs high latency.

Moreover, prior systems work on DRAM+SSD. FaRM [19] and DrTM [61] regard the battery-backed DRAM as PM, but the bandwidth and persistence properties of real PM are overlooked, causing inefficiency on the disaggregated PM:

1) Prior systems [9, 19, 44, 60] do not allow backups to serve read requests, since in backups the redo logs are asynchronously migrated to the in-place locations after updating the primary. Hence, only the primary can serve the latest data after commit [44]. As a result, all requests from coordinators are sent to the primary, causing a high load on the primary's

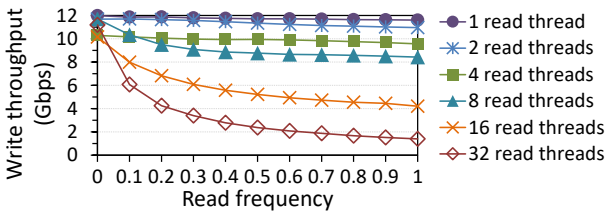


Figure 2: The throughput of RDMA writes to remote PM when mixing different frequencies of RDMA reads, e.g., 0.5 means that 5 reads are mixed with every 10 writes.

PM. However, PM shows lower write bandwidth than the new generations of RNICs, as mentioned before. We use 128GB Optane PM DIMMs and ConnectX-5 RNIC with 100Gbps InfiniBand to evaluate the throughput of RDMA writes when mixing different frequencies of RDMA reads. As shown in Fig. 2, when using 32 threads to concurrently issue read requests, the write throughput decreases by up to 87.5%. Hence, only using the primary to serve all requests makes the PM bandwidth become a performance bottleneck.

2) Lack of remote persistency guarantee. Current RDMA verbs have no persistency semantic [17]. For RDMA writes, the data are first buffered in a volatile cache in remote RNIC, which acknowledges (ACK) the writes once validated [59]. Hence, even if the client receives all ACKs, some data may not be persisted to remote PM in case of a crash. This misleads the client into considering that the data are durably stored in the remote PM. Hence, it is important to guarantee the remote persistency for RDMA writes, which is however not considered in prior dtxn systems due to using DRAM.

In summary, state-of-the-art dtxn systems become inefficient on the disaggregated PM due to causing substantial round trips and overlooking the PM properties. Our paper proposes FORD, an efficient one-sided RDMA-based dtxn processing system for the new disaggregated PM architecture.

3 The FORD Design

3.1 Overview

Fig. 3 shows the overview of FORD. The compute blades run dtxns and access application data in PM blades. The compute and PM pools communicate using connection managers (CMs), which maintain the RDMA queue pair connections.

FORD's workflow contains two stages. 1) The `Init` stage: ❶ The clients use the weak compute units in the PM pool (by RPCs) to allocate and register memory for subsequent RDMA operations [57, 72], and then load database (DB) tables. The DB tables are organized by indexes (§ 4.1). ❷ The compute and PM pools build RDMA connections using CMs. To calculate the remote address for one-sided RDMA in the compute pool, the CM in PM pool sends the metadata of all the indexes to each compute blade. These metadata only consume several MBs and are buffered in the compute pool (§ 4.1). Moreover, each memory blade notifies the compute blade about the roles (i.e., primary or backup) of its stored tables, so that the coordinator can correctly access the data during processing. 2) The

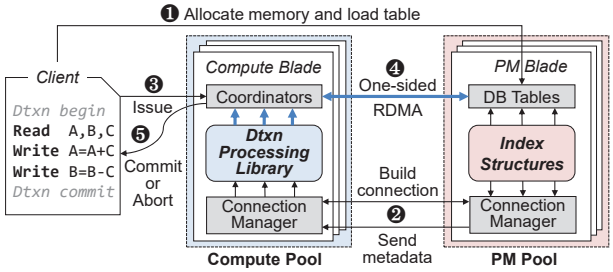


Figure 3: The system overview of FORD.

`Run` stage: ❸ The clients issue substantial dtxns to the compute blades, which spawn threads as coordinators to leverage our runtime library for fast dtxn processing. This library contains our novel designs in § 3.2–§ 3.5, and exposes easy-to-use interfaces (§ 4.2). ❹ Each coordinator uses one-sided RDMA to process dtxns, which are serialized by locking and version validations. Hence, there is no consistency requirement among compute blades. ❺ After processing, the coordinators report "tx_commited" or "tx_aborted" to clients. The `Init` stage performs only once before the `Run` stage, and the weak compute units in PM pool are not involved in the `Run` stage.

3.2 Hitchhiked Locking

As analyzed in § 2.3 and shown in Fig. 4a, prior works consume 3 RTTs to separately read, lock, and validate data to process a general read-write dtxn in Fig. 3.

To reduce the heavy round trips, FORD proposes a *hitchhiked locking* scheme to lock the data that belong to the *read-write set* when reading them in the execution phase. The read and write sets are known according to the transaction logic. FORD sends the lock request together with the read request in a hitchhiker manner. In this way, the read-write data do not need to be locked and validated after execution, since other transactions cannot modify the locked data. Therefore, the total round trips of processing a dtxn are efficiently reduced.

Due to not using the CPUs in PM pool, it is hard to lock and read data using one-sided RDMA in one round trip. To address this issue, FORD adopts the doorbell mechanism [28] to batch the RDMA `CAS` followed by an RDMA `READ` in one request, which is delivered and ACKed in one round trip, instead of being separately issued in two round trips, as shown in Fig. 4b. The RDMA `CAS` first tries to lock the remote data, and RDMA `READ` further fetches the data. Since the transport mode is *reliable connection*, the two RDMA operations are reliably delivered to the remote RNIC in order [51]. Then the batched operations are executed by RNIC as the delivering order to ensure correctness. After receiving the ACK of the batched request, the coordinator checks whether the locking is successful by comparing the return value of RDMA `CAS` with the previously sent lock value, i.e., only equality means a success. If the locking fails, the coordinator aborts the dtxn and unlocks the previously locked data to avoid deadlocks. Fig. 4c shows our hitchhiked locking scheme, which locks and reads the read-write data (e.g., {`A`, `B`}) using one-sided RDMA in one round trip, thus reducing the latency.

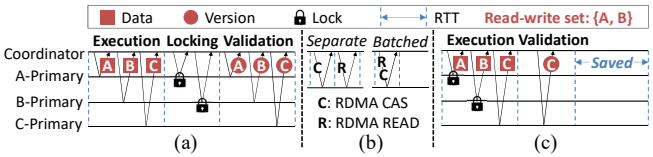


Figure 4: (a) Legacy schemes serially read, lock, and validate the read-write set. (b) Adopting doorbell batching reduces the number of round trips. (c) Our hitchhiked locking avoids extra round trips for locking and validating the read-write set.

Our hitchhiked locking is different from 2PL, which locks all the data before execution. FORD still maintains the optimistic feature of OCC to avoid contentions for the data that are only read. Specifically, the read-only data (e.g., data C in Fig. 4c) are not locked in the execution phase, and the locked read-write data can still be read by other coordinators (but cannot be locked). There is a validation phase to guarantee the version correctness for the read-only data. If a dtxn does not have the read-only data, the validation is eliminated.

Enabling hitchhiked locking requires remote data addresses for one-sided RDMA. FORD leverages a hash indexing scheme for the coordinator to compute the remote address of a bucket and read it (§ 4.1). Due to hash collisions, it is hard to accurately lock the slot in a remote bucket at the first read. However, directly locking the entire bucket prevents other coordinators from locking different slots in the same bucket, causing unnecessary contentions. Hence, the hitchhiked locking is disabled when the data are first read. After reading, the coordinator obtains the remote data addresses and buffers them in its local cache. Each time the previous data are read again, the local cache provides the addresses to enable hitchhiked locking. If some remote data addresses in the PM pool are changed by a coordinator (e.g., some data are deleted and then inserted to different places), another coordinator can easily discover that its buffered addresses become stale, since the key of the fetched data mismatches the queried key. In this case, the coordinator re-reads the bucket to obtain the correct data and updates its buffered addresses.

Our hitchhiked locking is different from: 1) FaRM [18, 19, 44] and DrTM+H [60], that consume a dedicated RTT to lock data. 2) DrTM+R [9], that exclusively locks all the data in the read and write sets. 3) FaSST [29], that uses RPC to lock data, which fails to work on the disaggregated memory. Unlike FaSST, FORD leverages one-sided RDMA to read and lock data in one round trip. Hitchhiked locking does not lengthen the lock duration due to eliminating the locking and validation phases for the read-write data. In the above systems that support OCC and primary-backup replication, we summarize the *lock duration*: 1) **FaRM** [18,19,44]. 4 phases = lock + validate + commit backup + commit primary&unlock. 2) **FaSST** [29]. 5 phases = lock + validate + log + commit backup + commit primary&unlock. 3) **DrTM+R** [9]. 4 phases = lock + validate + update + unlock. 4) **DrTM+H** [60]. 3 phases = lock&validate + commit backup + commit primary&unlock. 5) **Our FORD**. 3 phases (or 4 phases) w/o (or w/) read-only data = read&lock read-write set (or + validate read-only set) + commit all replicas (§ 3.3) + background unlock (§ 3.3.2).

Though our lock duration experiences 4 phases w/ read-only data, the coordinator can immediately detect lock conflicts in the execution phase, and run the next dtxn as early as possible. Hence, FORD avoids the aforementioned wastes of the execution (or execution+locking) phases due to the lock (or validation) failures in prior systems [9, 19, 60]. This trade-off is beneficial for improving the transaction throughput.

3.3 Coalescent Commit

As analyzed in § 2.3 and shown in Fig. 5a, existing dtxn systems spend 2 RTTs to separately write redo logs to the backup and then update the primary to finish commit. Hence, if the primary crashes, the dtxn can roll forward by using the redo logs in backups. However, this incurs high network overheads on the disaggregated PM, since each read-write dtxn needs 2 RTTs to replicate the updated data.

To reduce latency, FORD proposes a *coalescent commit* protocol to update the primaries and backups together in only one round trip. The coordinator commits the dtxn if the ACKs from all replicas are received. Otherwise, the dtxn aborts and rolls back. In fact, there is a trade-off between the replica commit latency and recovery state (i.e., 2 RTTs + roll forward, or 1 RTT + roll back). In practice, the commit latency is more important for the disaggregated PM, since we need to decrease the number of round trips to accelerate dtxn processing in common cases, in which no ACK is lost. Hence, we choose to commit all replicas together to improve the performance, and support to roll back dtxns in case of failures.

In the disaggregated PM architecture, we need to consider how to update replicas when using coalescent commit. For primaries, it is efficient to in-place update data, since the coordinators can directly read and lock the remote data without address redirections. But for backups, it is inefficient to send redo logs like FaRM [19, 44] and DrTM+H [60]. Because the CPUs in the PM pool are not involved in processing dtxns, the new data in redo logs will not be installed after commit. As a result, the backup cannot work after the memory is filled up by logs. Hence, we choose to in-place update the backup.

3.3.1 Parallel Undo Logging

In general, it is challenging to in-place update the backups and primaries in one round trip. Because in case of a crash, the remote old data could be partially overwritten, which prevents the dtxn from being rolled back. To tackle this challenge, FORD sends *undo logs* to all the replicas before in-place updates. Hence, the dtxns can roll back using the old data in undo logs. Unlike redo logs, the undo logs are simply discarded by setting the log status to be "committed" after the dtxn commits, which is completed by coordinators in the background. Hence, undo logging meets the requirement of PM pool, i.e., not involving the remote CPU to move data.

The next question is how to send undo logs to remote replicas. One solution is to spend a dedicated round trip to

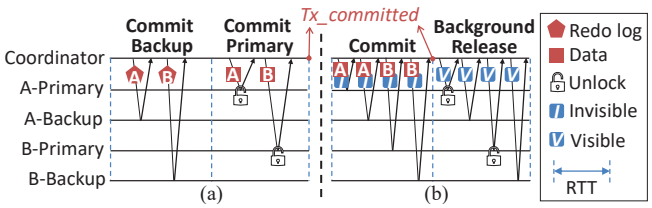


Figure 5: The comparisons between (a) separate commit, and (b) coalescent commit.

send logs, which however causes extra RTTs. We observe that undo logs can be immediately generated once the old data of the read-write set have been read in the execution phase. Based on this observation, we design a *parallel undo logging* scheme to generate and send undo logs in parallel with the transaction logic execution. Therefore, it is unnecessary to consume an extra round trip to send undo logs. To ensure atomicity, the coordinator only needs to check that all the ACKs of log writes (i.e., RDMA `WRITE`) are returned before updating the replicas. Note that the redo logs cannot be sent in the execution phase, because we have to wait for completing the transaction execution to obtain the newest data to generate redo logs, which heavily weakens the parallelism especially in the transaction that goes through a long-time execution logic, e.g., the `New Order` transaction in TPCC [15].

3.3.2 Visibility Control

In order to ensure consistency, our coalescent commit protocol guarantees that the data that being updated in the replicas are not partially read. Since our hitchhiked locking scheme does not block read-only requests, a coordinator possibly reads some remote data that are being updated, causing inconsistency. To avoid this, FORD proposes a one-sided RDMA-based *visibility control* to decide whether the data are visible to coordinators, as shown in Fig. 5b. The idea is to batch an invisible request followed by an RDMA `WRITE` into one request to update the remote replicas: 1) The invisible request prevents other coordinators from reading data by setting the invisible flag to 1. FORD implements the 1-bit invisible flag and 63-bit lock value in an 8B value, called `VLock`, which is atomically modified via an RDMA `CAS`. 2) The RDMA `WRITE` in-place updates the remote replica. After receiving *all* ACKs, the coordinator reports "tx_committed" to clients. Otherwise, e.g., a replica fails, the coordinator rolls back the dtxn by using undo logs. After commit or rollback, the coordinator unlocks data and makes them visible by writing an 8B zero to `VLock` in a background *release* phase.

The release phase does not exist on the critical path of the dtxn commit. It incurs only 0.5 round trip time (RTT) or can be fully overlapped: 1) Once the remote RNIC receives the RDMA `CAS` request and clears the `VLock`, other coordinators can immediately access the remote data. It is unnecessary to wait for returning the ACK, thus only consuming 0.5 RTT to make data visible. If some data are currently invisible, a coordinator can re-read them until visible. After all the required data become visible, the coordinator continues to process dtxns to guarantee the atomic visibility. 2) If there is no coordinator currently reading the invisible data, the background release phase is completely overlapped with other in-flight dtxns, thus avoiding re-read operations.

3.4 Backup-enabled Read

As discussed in § 2.3, only leveraging the primary to handle all the requests decreases the throughput due to the limited write bandwidth of PM. To tackle this challenge, FORD *enables the backups to serve read requests* for the read-only (RO) data, i.e., the coordinators are allowed to read the RO data from backups. This frees up the PM in the primary to serve other requests (e.g., lock and write), thus balancing the load to improve throughput. Based on our coalescent commit that *in-place* updates all replicas, it is easy for a coordinator to read the RO data from backups due to no address redirection.

FORD guarantees the correctness of the RO data that are read from backups. If a dtxn (e.g., `dtxn1`) reads all its RO data before (or after) another dtxn (e.g., `dtxn2`) commits the replicas, `dtxn1` will obtain the old (or new) data, which guarantees the correctness since `dtxn2` is uncommitted (or committed). However, if `dtxn1` reads multiple RO data and goes through `dtxn2`'s execution and commit phases, the data that `dtxn1` has read are possibly stale after `dtxn2` updates the replicas. To address this issue, FORD validates the versions of all `dtxn1`'s RO data before `dtxn1` commits, as guaranteed by our hitchhiked locking scheme in Fig. 4c. If the validation fails, the coordinator aborts `dtxn1` to ensure correctness.

Existing systems unfortunately fail to efficiently read data from backups: 1) For legacy database systems, e.g., Microsoft Azure [16] and Amazon Aurora [56]. The primary (or backup) replica handles the write (or read) requests from clients. After a client writes data to the primary, the backup needs to wait for receiving and installing the new data that are sent from the primary. Hence, after updating the primary, the clients are delayed to read the latest data from backups, thus causing extra latency. Moreover, in the disaggregated PM, the CPUs in the primaries and backups are not involved in dtxn processing. Hence, the data send/receive operations between replicas fail to work in the PM pool. 2) For prior RDMA-based dtxn systems [19, 44, 60]. The coordinator writes updated data to the primaries (i.e., in-place updates) and backups (i.e., redo logs) to commit a dtxn. However, other coordinators cannot read the backups after the dtxn commits, since the latest data in backups have not been transmitted from the redo logs to the in-place locations. Moreover, in the disaggregated PM, the backups fail to extract the updated data in redo logs and transmit these updates due to involving the CPU in PM pool during dtxn processing. Unlike the above systems, our coalescent commit protocol in-place updates the backups and primaries together without involving the CPU in PM pool. Hence, the coordinators are allowed to read the latest in-place data from backups after the dtxn commits, which alleviates the load on primary's PM to improve the throughput.

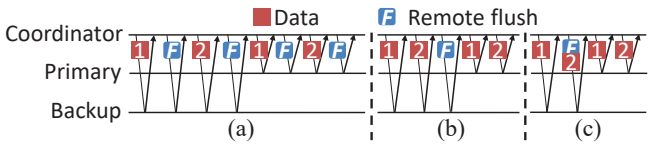


Figure 6: Ensuring remote persistency using (a) full flush, (b) selective flush, and (c) selective flush with request batching.

3.5 Selective Remote Flush

It is important to guarantee the remote persistency when committing the data updates to the PM pool, which is however overlooked in prior dtxn systems that use DRAM as the memory. Recently, the one-sided RDMA `FLUSH` [23, 48] is being proposed to persist data from remote RNIC to PM. However, flushing each data to each remote replica (i.e., full flush) consumes many round trips. As shown in Fig. 6a, updating 2 data incurs 8 round trips after using remote flushes.

In order to guarantee remote persistency with low network overhead, we propose a *selective remote flush* scheme, as shown in Fig. 6b. The idea is to issue an RDMA `FLUSH` after the final RDMA `WRITE` and only to backups, since: 1) RDMA `FLUSH` supports to flush all the previous written data. Hence, it is sufficient to use one RDMA `FLUSH` after the final write to one replica. 2) In the $(f+1)$-way primary-backup replication, once the data are persisted in all backups, even if the primary crashes, we can recover the primary by using backups. Hence, it is sufficient to issue RDMA `FLUSH` to only backups. Note that if all the $f+1$ replicas fail, the data cannot be recovered [19]. FORD guarantees remote persistency with at most f replica failures. Thus, by issuing necessary flush operations, FORD significantly reduces the round trips.

As RDMA `FLUSH` is currently unavailable in programming due to the needs of modifying RNIC and PCIe [48], we leverage one-sided RDMA `READ`-after-`WRITE` to flush the data in RNIC to memory like [27, 31]. Specifically, the RDMA `READ` fetches any size (e.g., 1B) of the data that are written by RDMA `WRITE`. Then, the remote RNIC will issue all PCIe writes before issuing PCIe reads to satisfy the RDMA `READ`. In this way, the data in RNIC are written to PM. We further optimize this procedure by batching the write and read into one request to eliminate the extra read round trip. This implementation is compatible with the future one-sided RDMA `FLUSH`, i.e., replacing RDMA `READ` with RDMA `FLUSH`, as shown in Fig 6c. In essence, our selective remote flush scheme aims to reduce the round trips when ensuring remote data persistency. Hence, this scheme is not affected by the specific implementation of remote data flushing, e.g., using the future RDMA `FLUSH` primitive or current `READ`-after-`WRITE` method.

3.6 Failure Tolerance

The replica fails in PM pool. Due to supporting replication in dtxn processing, FORD recovers the data in the failed replicas from other replicas that are alive. If any primary or backup fails: i) Before commit, the coordinator aborts the transaction and unlocks the data. ii) During commit, the coordinator aborts the transaction, reads the remote undo logs to revoke data updates and unlocks data. iii) In the release phase, the transaction has already committed. The coordinator clears the `VLock` in the replicas. If some replicas that cannot be recovered, we add new replicas to maintain the $(f+1)$-way replication, and migrate data to the new replicas.

The coordinator fails in compute pool. Due to writing undo logs to remote replica, FORD handles coordinator failures by rolling back dtxns. Like FaRM [19] and DrTM [61], FORD supports to use leases [22] to detect failures. After the leases expire (e.g., 5 ms [19]), a failure possibly occurs. However, once a coordinator fails before it reports "tx_committed", it is unknown whether the remote replicas have been updated. To address this issue, FORD reads the undo logs in replicas to revoke all the updates and reports “tx_aborted" to clients.

The network communication fails. Due to network partitions, either availability or consistency is sacrificed based on the CAP theorem [6, 21]. In this case, FORD only allows the primary partition [5] to serve requests, which guarantees the strong consistency of ACID dtxns for OLTP workloads.

3.7 Put It All Together

Fig. 7 illustrates how our designs (§ 3.2–§ 3.5) work together to process dtxns by using one-sided RDMA primitives. The requests in one RTT are issued and ACKed in parallel.

1) Execution. A coordinator reads and locks the required read-write data from primaries using batched RDMA `CAS+READ` in one round trip. The read-only data can be fetched from backups or primaries using RDMA `READ`. The undo logs are immediately generated and written to remote replicas by RDMA `WRITE` in parallel with the execution. The concurrent dtxns that have conflicting accesses to the same remote data are serialized by locks. If any lock operation fails, the coordinator aborts the dtxn and unlocks the remote data.

2) Validation. After execution, the coordinator reads the versions of the read-only data (if any) using RDMA `READ`, and verifies that the data versions are not modified by other coordinators. If a version changes, the coordinator aborts the dtxn and unlocks the remote data.

3) Commit. After validation, the coordinator checks that all the ACKs of undo logs are received, and then commits the updated data to all the replicas in one round trip. The data in primaries are marked to be invisible and updated with the batched RDMA `CAS+WRITE`. The data in backups are updated and further flushed from RNIC to PM using remote data flushing operations. Therefore, the coordinator uses batched RDMA `CAS+WRITE+FLUSH` to update backups. After receiving all the ACKs from replicas, the coordinator reports “tx_committed" to the client. Afterwards, the coordinator starts processing the next dtxn.

4) Release. After the dtxn commits, the coordinator uses RDMA `CAS` to release the remote data by setting them visible and unlocking them in the background.

FORD efficiently handles different types of dtxns. 1) For read-only dtxns, FORD reads remote data and validates ver-

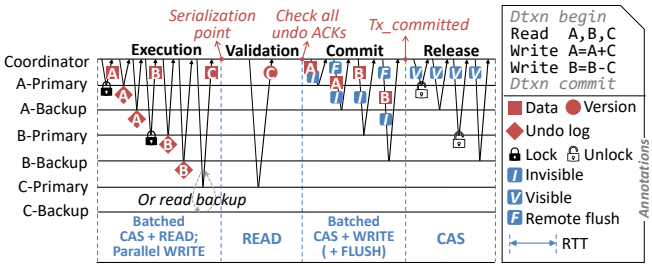


Figure 7: Distributed transaction processing in FORD.

sions before commit. Prior systems [19, 60] adopt similar operations. However, the difference is that FORD supports coordinators to read backups to improve the throughput while prior systems do not support this. 2) For read-write dtxns, only 2 RTTs (i.e., Execution+Commit, w/o read-only data) or 3 RTTs (i.e., Execution+Validation+Commit, w/ read-only data) are on the critical path. Compared with existing designs that require 5 RTTs [19, 29] or 4 RTTs [60] to process a read-write dtxn (as analyzed in § 2.3), FORD significantly improves the performance for the disaggregated PM architecture.

3.8 Correctness and Overhead Analysis

Serializability. FORD leverages locks and validations to guarantee serializability. The committed read-write dtxns are serializable at the point where all the written data are successfully locked. The committed read-only dtxns are serializable at the point of their last read. FORD guarantees these serializability points by ensuring that *the data versions at the serialization point are equal to the versions during execution*, i.e., locking ensures this for the written data since other coordinators cannot modify the versions of locked data, and validation ensures this for the read data since a version change will abort the dtxn. Moreover, to guarantee serializability across failures, the coordinator waits for all ACKs from all replicas before commit. Once a replica fails during the coalescent commit, FORD aborts the dtxn since an ACK is not received.

ACID. FORD ensures the ACID properties for dtxns: (1) **A**tomicity. FORD records undo logs, which are used to revoke the partial updates if a failure occurs before commit. (2) **C**onsistency. All the data versions are consistent before the dtxn starts and after it commits. (3) **I**solation. FORD uses locks and version validations to guarantee the serializability among the read-write and read-only dtxns. (4) **D**urability. The updated data are persistently stored in PM after commit.

The Number of RDMA Operations. Due to fully using one-sided RDMA to bypass the CPUs in PM pool, FORD inevitably increases the number of RDMA operations to commit a dtxn. It is worth noting that the new RNICs (e.g., ConnectX-5 [52]) are efficient to handle one-sided RDMA operations including `CAS` [60]. Hence, slightly increasing the number of RDMA operations has negligible impacts on performance. In fact, FORD focuses on reducing the number of RDMA round trips, which is more important to improve the performance since the RDMA round trip still suffers from higher latency (e.g., 3–8 µs [3]) than local access (e.g., 62–305 ns [67]).

4 Implementations

4.1 Data Store in Memory Pool

FORD supports different indexes to organize database (DB) tables in PM pool, e.g., hash tables and B$^+$-trees. These indexing schemes form the data store of FORD, called *FStore*. Our transaction techniques are independent of the specific index used in FStore, since these techniques aim to reduce network round trips and balance loads, and regard remote data as general objects. For example, when using B$^+$-trees, our hitchhiked locking scheme reads and locks the leaf nodes, and our coalescent commit scheme writes the updated tree nodes back to all replicas together. The internal pointer nodes are cached to reduce remote tree traversing. Moreover, since the hash table is widely used in fast RDMA operations [18, 54, 61, 72], we use hash table as an example to present the implementations of FStore. Each hash table maintains a DB table and supports read/update/insert/delete operations.

The records in DB tables are persistently stored in FStore. Existing hashing schemes that support fixed-size and variable-size records can be used in FORD, e.g., RACE hashing [72]. When supporting fixed-size records, the records are stored in the hash table for direct access. When supporting variable-size records, the pointers of records are stored in the hash table. In this case, our hitchhiked locking scheme reads and locks the pointer, and then fetches the record. Hence, FORD is flexibly to adapt different hash schemes to support fixed or variable record sizes. For simplicity, we show an implementation of storing fixed-size records. Like FaSST [29], the record consists of an 8B key and a maximum sized value (e.g., 1KB). Such record meets many OLTP workloads (e.g., TPCC [15]). To further support dtxns, FORD packs the record with the following information into an object, called *FObj*.

- `Occupy` (1B): Whether this FObj occupies a slot.
- `TableID` (8B): DB table that this record belongs to.
- `Version` (8B): Version number of this record.
- `VLock` (8B): 1-bit (in)visibility flag and 63-bit lock.

After allocating and registering a memory region (MR) in PM pool, FStore enables clients to load records into hash tables before running dtxns. Fig. 8 shows the structure of hash tables. A hash table contains an array of buckets, and each bucket contains several slots and one pointer called `Next`. The numbers of buckets and slots are configured by clients. Each FObj occupies a slot. A client initializes a FObj and hashes its key to obtain the target bucket (e.g., *b*1) to be inserted. After inserting the FObj to an empty slot, its `Occupy` is set to 1. If *b*1 is full, the FObj is inserted to a new bucket (e.g., *b*2) whose address is recorded in the `Next` pointer of *b*1. *b*2 is stored in a reserved space (RS) of MR. The size of RS is set by the client, e.g., 20% of the MR space. The client uses a pointer, called RS-Ptr, to trace the current bucket address in RS. Moving the RS-Ptr forward to a bucket size will generate a new bucket in RS. If the RS is exhausted but the hash collision still occurs, the client re-allocates memory to load tables.

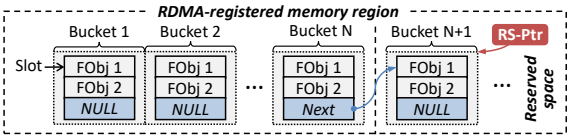


Figure 8: The hash table structure in FStore.

To enable coordinators to calculate remote addresses for one-sided RDMA in dtxn processing, the connection manager in PM pool sends the metadata (as listed below) of each hash table to the compute pool during network interconnections.

- `TableID` (8B): Global unique database table id.
- `Addr` (8B): Virtual start address of this hash table.
- `Off` (8B): Offset between `Addr` and MR's start address.
- `BucketNum` (8B): Bucket number of the hash table.
- `BucketSize` (4B): Size of a bucket (in bytes).
- `SlotNum` (4B): Number of slots per bucket.

Given the key (e.g., `K0`) of a record, if its remote address is buffered in the local cache, the coordinator directly reads the record using an RDMA `READ`. Otherwise, the coordinator reads a remote record as the following Steps:

S1: Calculate the bucket id:
`bucket_id = Hash(K0) mod BucketNum`

S2: Calculate the bucket offset in the remote MR:
`bucket_off = bucket_id × BucketSize + Off`

S3: Read the remote bucket (*bkt*) using `bucket_off`.

S4: Compare `K0` with the `SlotNum` keys in *bkt*. If a key = `K0`, the record is obtained. Then go to S7. Or else go to S5.

S5: If the `next` field of *bkt* is `NULL`, there is no such remote record. Then go to S8. Or else go to S6.

S6: Calculate the next bucket offset as below and go to S3.
`bucket_off = bkt.next − Addr + Off`

S7: Exit if the record is visible. Or else re-read it until visible.

S8: Exit with a `KEY_NOT_EXIST` hint.

Since the metadata size of a hash table is only 40B and each remote address is 8B, the local cache in compute pool can buffer all these metadata and addresses, as shown in Fig. 15a. Caching metadata is scalable, because the compute blades do not need to synchronize their metadata with each other: 1) The metadata of index does not change. 2) If the cached addresses are stale, FORD enables the coordinator to detect this and update its own cached addresses, as discussed in § 3.2.

4.2 Transaction Interfaces

FORD provides a runtime library, called *FLib*, for applications to process dtxns. Flib exposes the following interfaces:

- `TxBegin`: Start to execute a dtxn and record its id.
- `AddRO`: Add an initialized FObj to the read-only set.
- `AddRW`: Add an initialized FObj to the read-write set.
- `TxExecute`: The coordinator reads the remote data specified in read-only and read-write sets, and then executes the dtxn logic. Our hitchhiked locking and backup-enabled read schemes are leveraged.
- `TxCommit`: After execution, the coordinator commits the updated data to remote primaries and backups using our coalescent commit and selective remote flush schemes.

```
1  bool WriteCheck(uint64_t dtxn_id, DTXN* dtxn) {
2    // The `dtxn` invokes FLib interfaces
3    dtxn->TxBegin(dtxn_id);
4    // Use a random account as the key
5    uint64_t acct_id = RandomAccount();
6    FObj* sav_obj = new FObj(SavingsTableID, acct_id);
7    FObj* chk_obj = new FObj(CheckingTableID, acct_id);
8    dtxn->AddRO(sav_obj);
9    dtxn->AddRW(chk_obj);
10   if (!dtxn->TxExecute()) return false;
11   // Get record values and run transaction logic
12   sav_val_t* sav = (sav_val_t*) sav_obj->value;
13   chk_val_t* chk = (chk_val_t*) chk_obj->value;
14   if (sav->balance + chk->balance < PredefinedAmount)
15     chk->balance -= (PredefinedAmount + 1);
16   else chk->balance -= PredefinedAmount;
17   bool status = dtxn->TxCommit();
18   delete sav_obj; delete chk_obj;
19   // Report commit (true) or abort (false) to client
20   return status;
21 }
```

Figure 9: The example of C++ code using FLib interfaces.

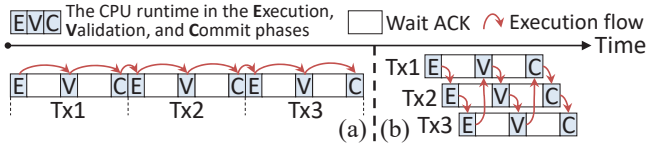


Figure 10: The comparisons between (a) sequential processing, and (b) interleaved processing, in one thread.

Our transaction interfaces support general transaction processing. Specifically, the developers are not required to know all the read/write sets at the beginning of each transaction. Instead, developers call `AddRO`, `AddRW`, and `TxExecute` multiple times when reading/writing data occurs during a transaction. Fig. 9 illustrates an example of using our interfaces in the `Write Check` transaction of the SmallBank benchmark [50]. This transaction reads the balances from the `Savings` and `Checking` tables, and updates the balance in the `Checking` table. It shows that our interfaces are easy-to-use.

4.3 Interleaved Transaction Processing

As shown in Fig. 10a, sequentially processing dtxns in a thread wastes the CPU cycles due to waiting for RDMA ACKs, which significantly decreases the throughput. To avoid CPU idling in the compute pool, FORD leverages an interleaved processing model that enables multiple coordinators in one thread to process different dtxns in pipeline, as presented in Fig. 10b. In this way, the network RTTs are efficiently hided and the CPU cores in the compute pool are fully utilized to improve the throughput.

We use coroutines [29, 60] to implement the interleaved processing. Each CPU thread generates several coroutines and each coroutine acts as a coordinator to execute dtxns. After issuing the RDMA requests, a coroutine yields its CPU core to another coroutine to process the next dtxn. A dedicated coroutine in each thread polls RDMA ACKs. If all ACKs of a coroutine arrived, FORD schedules this coroutine to occupy the CPU core to resume execution. The results in Fig. 16 show that using a proper number of coroutines improves the throughput without heavily increasing the latency.

5 Performance Evaluation

5.1 Experimental Setup

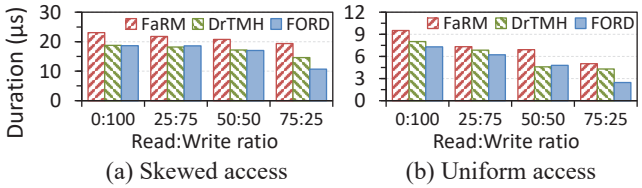
Testbed. We use three machines, each of which contains a 100Gbps Mellanox ConnectX-5 IB RNIC. They are connected via a 100Gbps Mellanox SB7890 IB switch. One machine equipped with the Intel Xeon Gold 6230R CPU and 8GB DRAM is leveraged as the compute pool to run coordinators. Other two machines form the PM pool, each of which contains 6 interleaved 128GB Intel Optane DC PM modules. Each database table is stored in the two PM machines to maintain a 2-way replication, i.e., one primary and one backup.

Benchmarks. We leverage a key-value store (`KVS`) as the ***micro-benchmark*** to analyze how different factors affect each design of FORD. `KVS` stores 1 million key-value pairs in one table, in which the key is 8B and value is 40B. The transaction in `KVS` accesses a specific number of objects with different read:write ratios and different access patterns as configured in § 5.2. `KVS` supports the skewed and uniform access patterns, in which the skewed access uses the Zipfian distribution with the default skewness 0.99 [10]. We further adopt three OLTP benchmarks, i.e., `TATP` [49], `SmallBank` [50], and `TPCC` [15], as the ***macro-benchmarks*** to examine the end-to-end performance. These benchmarks are widely used in prior studies [19, 29, 60, 61]. `TATP` models a telecom application and contains 4 tables, in which 80% of the transactions are read-only, and the record size is up to 48B. `SmallBank` simulates a banking application that includes 2 tables, in which 85% of transactions are read-write, and the record size is 16B. `TPCC` models a complex ordering system that consists of 9 tables, in which 92% of transactions are read-write, and the record size is up to 672B. We generate 8 warehouses in `TPCC`. We have implemented all the workloads of each macro-benchmark and run the standard transaction mix in § 5.3. We run 1 million dtxns in each benchmark, and report the throughput by counting the number of *committed* dtxns per second. We report the processing time of the committed dtxn as the latency, including the 50th and 99th percentile latencies.

Comparisons. We implement FORD[1] using C++ (13.1k lines of codes) and compare it with two state-of-the-art RDMA-based dtxn processing systems, i.e., FaRM [19] and DrTM+H [60] (called DrTMH). We use one-sided RDMA to re-implement their dtxn processes for the disaggregated PM. Moreover, our selective remote flush scheme is applied to FaRM and DrTMH to make them compatible with remote PM. We do not compare against FaSST [29] that fully uses two-sided RDMA, which is difficult to work in the disaggregated memory architecture due to consuming the remote CPUs in memory pool throughout the entire dtxn processing.

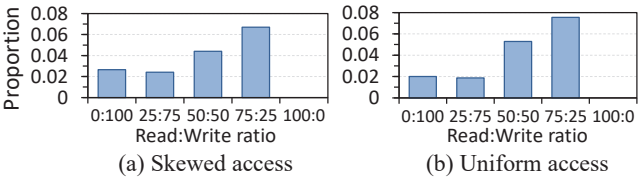
5.2 Micro-Benchmark Results and Analysis

Lock Duration. Locks are generally used to serialize dtxns. However, a long lock duration causes frequent aborts and leads to low throughputs. To obtain the lock duration, we configure the coordinator to not abort dtxn but wait for the data to be unlocked if the locking fails. We compare the lock duration in FORD, FaRM and DrTMH by using 64 coordinators to concurrently run dtxns in which each dtxn processes one data. Fig. 11 shows the average lock duration of each coordinator at different read:write ratios in the dtxn mix, e.g., 25:75 means that 25% of the dtxns are read-only while 75% are read-write. The results show that the reduction of lock duration is larger in the uniform access when reducing the write ratio, since the uniform access has lower locality than the skewed access, which decreases the data hotness. Hence, the total time for the coordinator to wait for unlocking the hot data significantly decreases. Compared with DrTMH and FORD, FaRM suffers from longer lock durations since the data are locked across 4 phases, i.e., locking, validation, commit backup, and commit primary. DrTMH reduces the lock duration by merging the locking and validation into one phase. Our FORD uses the hitchhiked locking scheme to lock the read-write data in the execution phase, but the lock duration does not become longer, since the read-write data do not need to be locked or validated again, and the dtxn commits earlier.



(a) Skewed access (b) Uniform access

Figure 11: The lock durations.



(a) Skewed access (b) Uniform access

Figure 12: The proportions of invisibility durations.

Invisibility Duration. FORD leverages the coalescent commit scheme to update the primaries and backups together in one round trip. To avoid partial reads, the data are temporarily marked as invisible after commit until the background release phase. To analyze the overheads of the data invisibility, we record the total time spent for re-reading the invisible data until visible (i.e., invisibility duration) in 64 coordinators, and then calculate the proportion of the invisibility duration in the entire dtxn running time. As shown in Fig. 12, the proportion slightly decreases when increasing the read ratio from 0% to 25%, since the invisible data are reduced when decreasing writes. As the read ratio continues to increase, the proportion increases, since there are substantial read-only dtxns that wait for the data to be visible, which increases the total invisibility durations in all coordinators. When the read:write ratio is 100:0, all data are visible and the proportion becomes 0. The results show that the proportion is only less than 8% in different read:write ratios and access patterns, since the

[1] Open source code: https://github.com/minghust/ford

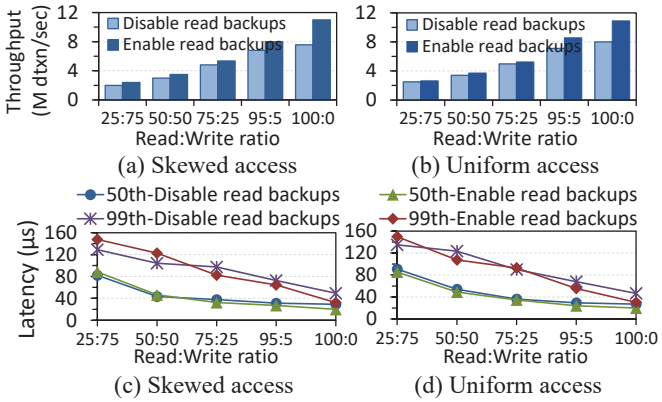


Figure 13: The dtxn throughput and latency when disabling/enabling coordinators to read the backup replicas.

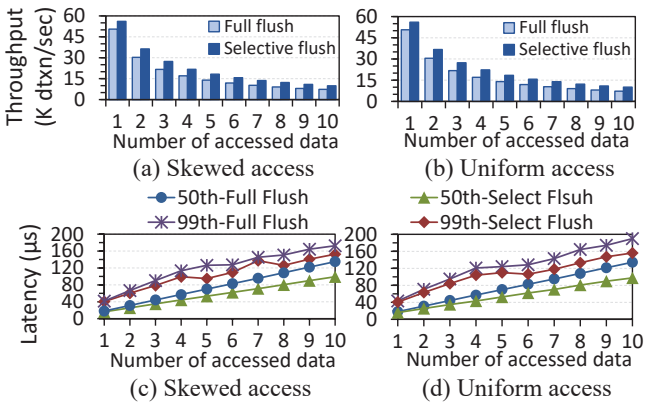


Figure 14: The dtxn throughput and latency using full/selective flush when accessing different numbers of data per dtxn.

background release phase consumes at most 0.5 RTT to make data visible. Therefore, the data invisibility in our coalescent commit design exhibits low performance overheads.

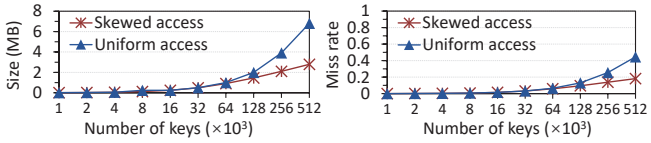
Read from Backups. Due to the limited write bandwidth of PM, FORD enables the coordinators to read the read-only data from backups to alleviate the load on the primary's PM. To demonstrate the benefits of this design, we run 224 coordinators to increase the load, and examine the dtxn throughput and latency when disabling/enabling the coordinators to read backups. Fig. 13 shows that as the read ratio increases, enabling coordinators to read the backup replica improves the throughput by up to 1.5×, and reduces the 50th/99th percentile latencies by up to 31.7%/35.3%. The backup absorbs substantial read requests to prevent all the coordinators from competing for the primary's PM, thus improving the throughput. When increasing the number of backup nodes, FORD will gain higher performance improvements since all the backups can be read to balance loads.

Remote Flush. FORD guarantees the remote persistency in dtxn processing by flushing the data from the RNIC cache to PM. We compare the dtxn throughput and latency when adopting the full flush and selective flush schemes discussed in § 3.5. To show the overheads of remote data flush, we use one coordinator to avoid extra contention overheads. We increase the number (1–10) of written data per dtxn to add the flush operations. The results in the skewed and uniform accesses exhibit similar trends. Fig. 14a and 14b show that our selective flush scheme improves the throughput by 28.7%/29.5% over the full flush scheme in skewed/uniform access. Fig. 14c and 14d show that the selective flush mitigates the 50th/99th percentile latencies by 22.5%/12.4% (22.8%/14%) in the skewed (uniform) access. Our selective flush scheme performs better due to only issuing flushes to the backups after the final RDMA `WRITE`, thus reducing the number of flush operations. Moreover, Fig. 14 shows that the performance of using the selective flush decreases when the number of accessed data increases. This is because other operations in the dtxn increase (e.g., data reads, validations, and remote writes), thus decreasing the overall performance.

Local Cache. The coordinator has a local cache to buffer remote data addresses for efficient one-sided RDMA operations. To evaluate the overheads (including the size and miss rate) of the local cache, we change the maximum number of accessible keys from 1k to 512k to obtain the average sizes of buffered addresses, and the average miss rates during address lookups. Fig. 15a shows that the buffered addresses only consume 6.8 MB even if uniformly accessing 512k keys with poor locality. Hence, a small MB-scale cache is sufficient for a coordinator to buffer remote addresses. Since a GB-scale DRAM is leveraged in the compute pool to store the metadata [45], it is unnecessary to limit the size of the coordinator-local cache in practice. Fig. 15b shows that the miss rate is 18.2%/44.6% when accessing 512k keys in skewed/uniform access. For a cache hit, the coordinator uses the buffered address to directly read the record. However, if a cache miss (or a hash bucket collision) occurs, the coordinator needs to calculate the remote bucket address and read a bucket to find the record, which incurs more latency. In general, the miss rate depends on the locality of workloads. If some remote addresses are not buffered, the cache misses are inevitable in dtxn processing. However, FORD provides a sufficiently large local cache for each coordinator to avoid evicting the buffered addresses from the cache, thus reducing the miss rate as much as possible.

5.3 Macro-Benchmark Results and Analysis

Coroutine Execution. To improve the throughput, FORD leverages coroutines to process dtxns to avoid CPU idling. A thread generates at least 2 coroutines since a specified coroutine in each thread is used to poll the RDMA ACKs. Fig. 16 shows the dtxn throughput and median latency in macro-benchmarks when changing the number of coroutines in one thread. The throughputs increase by 3.4×/2.2×/2.5× on `TATP`/`SmallBank`/`TPCC` until the CPU is saturated. On the other hand, the latency continues to increase when using more coroutines, since the execution pipeline becomes deeper, and the coroutines are scheduled to wait for occupying the CPU to resume execution. From the experimental results, we learn that using 6–8 coroutines is helpful to significantly improve the throughput without heavily increasing the latency.



(a) The sizes of buffered addresses (b) The miss rates on address lookups

Figure 15: The size and miss rate of the local cache.

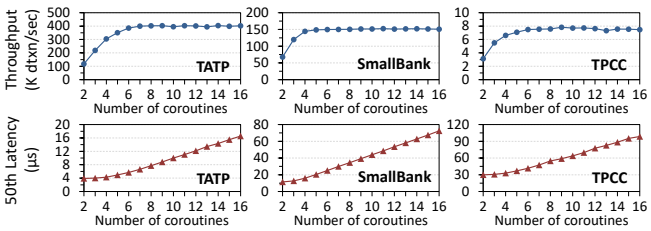


Figure 16: The dtxn throughput and 50th percentile latency in one thread when using different numbers of coroutines.

End-to-End Performance. We use 16 threads and each thread generates 8 coroutines (1 coroutine polls ACKs), as a total of $16 \times (8-1) = 112$ coordinators, to evaluate the end-to-end throughput and latency in FaRM, DrTMH, and FORD using the macro-benchmarks. In real experiments, due to different system scales (e.g., 90 machines [19]), the overall throughput in our small-scale testbed becomes lower than [19, 60]. However, our testbed can accurately evaluate the performance in different system configurations. Our selective remote flush scheme is applied to three systems to ensure remote persistency. As shown in Fig. 17, compared with FaRM/DrTMH, FORD improves the throughput by 1.4×/1.3×, and reduces the 50th (99th) latency by 12%/9.1% (54.8%/46.8%) in `TATP`, improves the throughput by 1.6×/1.3×, and reduces the 50th (99th) latency by 34.3%/30.9% (64.6%/32.4%) in `SmallBank`, and improves the throughput by 2.3×/1.4×, and reduces the 50th (99th) latency by 74.3%/66.2% (63.8%/28.7%) In `TPCC`. DrTMH outperforms FaRM by merging locking and validation phases. FaRM and DrTMH show high performance in `TATP`, since 80% of the dtxns are read-only and the uses of one-sided RDMA `READ`s accelerate the processing. However, in `SmallBank` and `TPCC` that contain extensive read-write dtxns, the performance decreases due to their long dtxn processing paths. Unlike them, our FORD efficiently mitigates the round trips to shorten the processing path, and balances loads on the replicas, thus improving the performance.

6 Related Work

Fast Distributed Transactions. Many systems have been proposed for efficient distributed transaction processing. Some designs leverage RDMA to handle transactions [9, 18, 19, 29, 31, 41, 44, 60, 61]. Storm [41] proposes a transactional API to operate remote data based on one-sided reads and write-based RPCs. HyperLoop [31] offloads some computations to RNIC and requires remote CPU to operate the metadata. Moreover, application locality [7, 30, 37] is exploited to convert a distributed transaction to a local one, which however sacrifices the generality. New transaction abstractions [63], replication protocols [70], and concurrency controls [40, 58, 64, 69] are

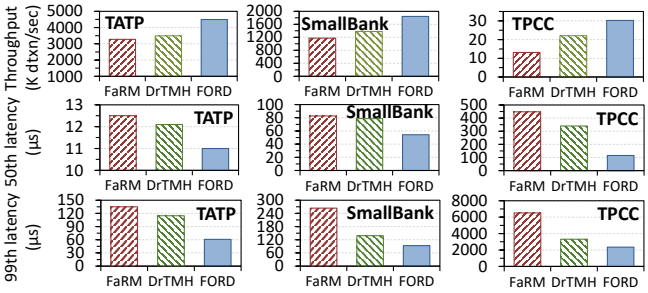


Figure 17: The end-to-end performance.

also proposed to improve the performance. The above systems work on the monolithic architecture, while our FORD focuses on the new disaggregated PM architecture and fully leverages one-sided RDMA to process transactions.

Distributed Persistent Memory. PM has been recently exploited in the distributed environments. These studies employ PM in a symmetric way, where each server in a cluster hosts the PM that can be accessed locally or remotely by other servers. Some designs expose interfaces of file system [3, 38, 65, 66] and memory management [46, 71]. Some studies provide optimization hints on system implementations when using RDMA and PM [27, 62]. In general, the symmetric deployment supports fast local accesses, but suffers from poor resource scalability and coarse failure domain due to using monolithic servers. Unlike these works, FORD provides transaction interfaces, and deploys PM in the disaggregated way to improve the scalability and failure isolation.

Disaggregated Memory. The disaggregated memory becomes popular in datacenters due to high resource utilization and elasticity. Existing works explore memory disaggregation in hardware architectures [35,36], networks [20,47], operating systems [45], KV stores [54], hash indexes [72], data swapping [2,8,24,43], and memory managements [1,34,39,55,57]. Our proposed FORD is orthogonal to these systems to build a fast transaction processing system for the disaggregated PM.

7 Conclusion

Our paper proposes FORD, a fast distributed transaction processing system that leverage one-sided RDMA for the new disaggregated persistent memory (PM) architecture. To accelerate transaction processing, FORD explores and exploits the request batching and parallelization to eliminate extra locking and validations, and commit all remote replicas together in a single round trip. Moreover, to efficiently utilize the remote PM, FORD enables the backup replicas to serve read requests to balance loads, and guarantees the remote persistency with low network overheads. Experimental results demonstrate that FORD significantly outperforms the state-of-the-art systems in terms of transaction throughput and latency.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant 62125202. We are grateful to our shepherd, Rong Chen, and anonymous reviewers for their feedbacks 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, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 775–787. USENIX Association, 2018.

[2] 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.

[3] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostic, 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 2020, Virtual Event, November 4-6, 2020*, pages 1011–1027. USENIX Association, 2020.

[4] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[5] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[6] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

[7] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with RDMA and caching. *Proc. VLDB Endow.*, 11(11):1604–1617, 2018.

[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] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 26:1–26:17. ACM, 2016.

[10] 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.

[11] Hewlett Packard Corporation. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine/, 2021.

[12] Intel Corporation. Intel optane persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2021.

[13] Intel Corporation. Intel optane persistent memory 200 series (512gb pmem) module. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory/optane-persistent-memory-200-series-512gb-pmem-module.html, 2021.

[14] Intel Corporation. Intel rack scale design architecture. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html, 2021.

[15] The Transaction Processing Council. Tpc-c benchmark. http://www.tpc.org/tpcc/, 2021.

[16] Azure SQL Database. Use read-only replicas to offload read-only query workloads. https://docs.microsoft.com/en-us/azure/azure-sql/database/read-scale-out, 2021.

[17] Chet Douglas. Rdma with pm: Software mechanisms for enabling persistent memory replication. In *Storage Developer Conference (2015)*, 2015.

[18] 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 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.

[19] 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.

[20] 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 2016, Savannah, GA, USA, November 2-4, 2016*, pages 249–264. USENIX Association, 2016.

[21] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[22] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, 1989.

[23] Paul Grun, Stephen Bates, and Rob Davis. Persistent memory over fabrics. In *Persistent Memory Summit (2018)*, 2018.

[24] 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.

[25] Doug Hakkarinen, Panruo Wu, and Zizhong Chen. Fail-stop failure algorithm-based fault tolerance for cholesky decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1323–1335, 2015.

[26] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. Nysiad: Practical protocol transformation to tolerate byzantine failures. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 175–188. USENIX Association, 2008.

[27] Anuj Kalia, David G. Andersen, and Michael Kaminsky. Challenges and solutions for fast remote persistent memory access. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 105–119. ACM, 2020.

[28] 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.

[29] 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.

[30] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: locality-aware distributed transactions. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 145–161. ACM, 2021.

[31] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 297–312. ACM, 2018.

[32] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[33] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pages 312–313. ACM, 2009.

[34] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: in-network memory management for disaggregated data centers. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 488–504. ACM, 2021.

[35] 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.

[36] 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.

[37] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2pc transaction management in distributed database systems. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1659–1674. ACM, 2016.

[38] 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 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 773–785. USENIX Association, 2017.

[39] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 757–773. ACM, 2020.

[40] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 479–494. USENIX Association, 2014.

[41] 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.

[42] Vmware Research. Remote memory. https://research.vmware.com/projects/remote-memory, 2021.

[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 2020, Virtual Event, November 4-6, 2020*, pages 315–332. USENIX Association, 2020.

[44] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 433–448. ACM, 2019.

[45] 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.

[46] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 323–337. ACM, 2017.

[47] 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 2019, Boston, MA, February 26-28, 2019*, pages 255–270. USENIX Association, 2019.

[48] Tom Talpey, Tony Hurson, Gaurav Agarwal, and Tom Reu. RDMA Extensions for Enhanced Memory Placement. https://tools.ietf.org/html/draft-talpey-rdma-commit-01, 2020.

[49] TATP. Telecom application transaction processing benchmark. http://tatpbenchmark.sourceforge.net/, 2011.

[50] The H-Store Team. Smallbank benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/, 2021.

[51] Mellanox Technologies. Rdma aware networks programming user manual. rev 1.7. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2015.

[52] Mellanox technologies. Connectx-5 vpi card. https://www.mellanox.com/files/doc-2020/pb-connectx-5-vpi-card.pdf, 2020.

[53] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pages 51–62. IEEE Computer Society, 2008.

[54] 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.

[55] 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.

[56] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1041–1052. ACM, 2017.

[57] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D. Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280. USENIX Association, November 2020.

[58] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. Polyjuice: High-performance transactions via learned concurrency control. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 198–216. USENIX Association, 2021.

[59] Live Webcast. Extending rdma for persistent memory over fabrics. In *SNIA Networking Storage Forum (2018)*, 2018.

[60] 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.

[61] 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.

[62] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and optimizing remote persistent memory with RDMA and NVM. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 523–536. USENIX Association, 2021.

[63] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining ACID and BASE in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 495–509. USENIX Association, 2014.

[64] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 279–294. ACM, 2015.

[65] 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 2019, Boston, MA, February 25-28, 2019*, pages 221–234. USENIX Association, 2019.

[66] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Filemr: Rethinking RDMA networking for scalable persistent memory. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 111–125. USENIX Association, 2020.

[67] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.

[68] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, February 2017.

[69] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 511–526. ACM, 2020.

[70] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 263–278. ACM, 2015.

[71] Yiying Zhang, Jian Yang, Amirsaman 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 '15, Istanbul, Turkey, March 14-18, 2015*, pages 3–18. ACM, 2015.

[72] 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.

Closing the B$^+$-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression

Yifan Qiao[†]*, Xubin Chen*[‡]*, Ning Zheng*[*]*, Jiangpeng Li*[*]*, Yang Liu*[*]*, and Tong Zhang*[†*]

[†] *Rensselaer Polytechnic Institute, NY, USA*

[‡] *Google Inc., WA, USA* [*] *ScaleFlux Inc., CA, USA*

Abstract

This paper studies how B$^+$-tree could take full advantage of modern storage hardware with built-in transparent compression. Recent years witnessed significant interest in applying log-structured merge tree (LSM-tree) as an alternative to B$^+$-tree, driven by the widely accepted belief that LSM-tree has distinct advantages in terms of storage cost and write amplification. This paper aims to revisit this belief upon the arrival of storage hardware with built-in transparent compression. Advanced storage appliances and emerging computational storage drives perform hardware-based lossless data compression, transparent to OS and user applications. Beyond straightforwardly reducing the storage cost gap between B$^+$-tree and LSM-tree, such storage hardware creates new opportunities to re-think the implementation of B$^+$-tree. This paper presents three simple design techniques that can leverage such modern storage hardware to significantly reduce the B$^+$-tree write amplification. Experiments on a commercial storage drive with built-in transparent compression show that the proposed design techniques can reduce the B$^+$-tree write amplification by over 10×. Compared with RocksDB (a key-value store built upon LSM-tree), the enhanced B$^+$-tree implementation can achieve similar or even smaller write amplification.

1 Introduction

This paper investigates the implementation of B$^+$-tree upon a growing family of data storage hardware that internally carry out hardware-based lossless data compression, transparent to the host OS and user applications. Modern all-flash array products (e.g., Dell EMC PowerMAX [9], HPE Nimble Storage [14], and Pure Storage FlashBlade [28]) always come with the built-in hardware-based transparent compression capability. Commercial solid-state storage drives with built-in transparent compression are emerging (e.g., computational storage drive from ScaleFlux [31] and Nytro SSD from Seagate [13]). Moreover, Cloud vendors have started to integrate hardware-based compression capability into their storage infrastructure, e.g., Microsoft Corsia [7] and emerging DPU (data processing unit) [5], leading to imminent arrival of cloud-based storage hardware with built-in transparent compression. With dedicated hardware compression engines, such storage hardware support high-throughput data (de)compression at very low latency and zero host CPU overhead.

As the most widely used indexing data structure, B$^+$-tree [12] powers almost all the relational database management systems (RDBMs) today. Recently, log-structured merge tree (LSM-tree) [25] has attracted significant interest as a contender to B$^+$-tree, mainly because its data structure could enable better storage space usage efficiency and lower write amplification. The arrival of storage hardware with built-in transparent compression could straightforwardly reduce or even eliminate the storage cost gap between B$^+$-tree and LSM-tree. This paper shows that such storage hardware can also be leveraged to significantly reduce B$^+$-tree write amplification. The key is to exploit the fact that in-storage transparent compression allows data management software employ *sparse data structure* without sacrificing the true physical storage cost. When running on such storage hardware, data management software could leave 4KB LBA (logical block address) blocks partially filled or even completely empty, without wasting the physical storage space usage. Intuitively, the feasibility of employing sparse data structure creates a new spectrum of design space for innovating data management systems [36].

This paper shows that B$^+$-tree could employ sparse data structure enabled by in-storage transparent compression to largely reduce its write amplification. We note that write amplification is measured based on the amount of data being written to the physical storage media (i.e., after in-storage compression), other than the amount of data being written by the host (i.e., before in-storage compression). In particular, this paper presents three simple yet effective design techniques: (1) *deterministic page shadowing* that can ensure B$^+$-tree page update atomicity without incurring extra write overhead, (2) *localized page modification logging* that can reduce the write amplification caused by the mismatch between the B$^+$-tree page size and the size of data modification,

and (3) *sparse redo logging* that can reduce the write amplification caused by B$^+$-tree redo logging (or write-ahead logging). With significantly reduced write amplification, B$^+$-tree can support much higher insert/update throughput, and more readily accommodate low-cost, low-endurance NAND flash memory (e.g., QLC NAND flash memory).

Accordingly, we implemented a B$^+$-tree (called B$^-$-tree) that incorporates the three design techniques. We further compared it with LSM-tree (RocksDB [30]) and normal B$^+$-tree (WiredTiger [33]). We carried out experiments on a commercial computational storage drive with built-in transparent compression [31]. The results well demonstrate the effectiveness of the proposed design techniques on reducing the B$^+$-tree write amplification. For example, under random write workloads with 128B per record, RocksDB and WiredTiger (with page size of 8KB) have write amplification of 14 and 64, respectively, while our B$^-$-tree (with 8KB page size) has a write amplification of only 8, representing 43% and 88% reduction compared with RocksDB and WiredTiger, respectively. The smaller write amplification can directly translate into a higher write throughput. For example, our results show that, under random write workloads, B$^-$-tree can achieve about 85K TPS (transactions per second), while the TPS of RocksDB and WiredTiger is 71K and 28K, respectively. Moreover, we note that the proposed design techniques mainly confine within the I/O module of B$^+$-tree and are largely orthogonal to the other modules. Hence, it is relatively easy to incorporate these techniques into existing B$^+$-tree implementations. For example, upon a baseline B$^+$-tree implementation, we only modified/added about 1,200 LoC to realize the B$^+$-tree.

2 Background

2.1 B$^+$-tree Data Compression

B$^+$-tree manages its data storage in the unit of page. To reduce data storage cost, B$^+$-tree could apply block compression algorithms (e.g., lz4 [23], zlib [37], and ZSTD [38]) to compress each on-storage page (e.g., the page compression feature in MySQL and MongoDB/WiredTiger). In addition to the obvious CPU overhead, B$^+$-tree page compression suffers from compression ratio loss due to the 4KB-alignment constraint, which can be explained as follows: Modern storage devices serve I/O requests in the unit of 4KB LBA blocks. As a result, each B$^+$-tree page (regardless of compressed or uncompressed) must entirely occupy one or multiple 4kB LBA blocks on the storage device (i.e., no two pages could share one LBA block). When B$^+$-tree applies page compression, the 4KB-alignment constraint could incur noticeable storage space waste. This can be illustrated in Fig. 1: Assume one 16KB B$^+$-tree page is compressed to 5KB; the compressed page must occupy two LBA blocks (i.e., 8KB) on the storage device, wasting 3KB storage space. Therefore, due to the CPU overhead and storage space waste caused by the 4KB-alignment constraint, B$^+$-tree page compression is not widely used in production environment. Moreover, it is well-known that, under workloads with random writes, B$^+$-tree pages tend to be only 50%∼80% full [12]. Hence, B$^+$-tree typically has a low storage space usage efficiency. In contrast, LSM-tree has a much more compact data structure and is free from the 4KB-alignment constraint in case of compression, which leads to a higher storage space usage efficiency than B$^+$-tree.

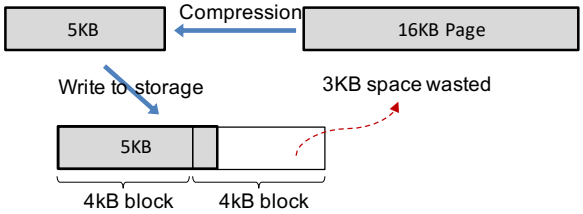


Figure 1: An example to show the storage space waste caused by 4KB-alignment constraint for B$^+$-tree page compression.

2.2 In-Storage Transparent Compression

Fig. 2 illustrates a computational storage drive (CSD) with built-in transparent compression: Inside the CSD controller chip, compression and decompression are carried out directly on the I/O path by the hardware engine, and the FTL (flash translation layer) manages the mapping of all the variable-length compressed data blocks. Since the compression is carried out inside the storage drive, it is not subject to 4KB-alignment constraint (i.e., all the compressed blocks are packed tightly in flash memory without any space waste).

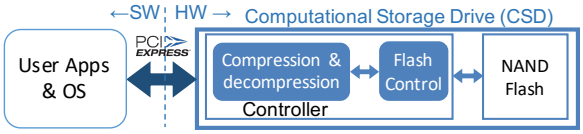


Figure 2: Illustration of a CSD with transparent compression.

As illustrated in Fig. 3, storage hardware with built-in transparent compression has the following two properties: (a) The storage hardware can expose an LBA space that is much larger than its internal physical storage capacity. This is conceptually similar to the thin provisioning. (b) Since certain data patterns (e.g., all-zero or all-one) can be highly compressed, we can leave one 4KB LBA partially filled with valid data without wasting the physical storage space. These two properties decouple the logical storage space utilization efficiency from the physical storage space utilization efficiency. This allows data management software to employ *sparse data structure* in the logical storage space without sacrificing the true physical storage cost, which creates a new spectrum of design space for data management systems [36].

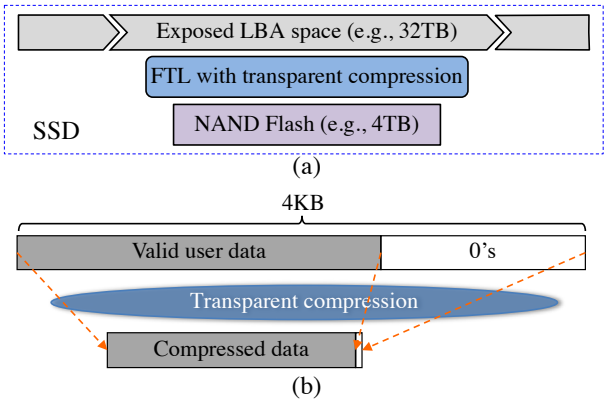


Figure 3: Illustration of the decoupled logical and physical storage space utilization efficiency enabled by storage hardware with built-in transparent compression.

2.3 B$^+$-tree vs. LSM-tree

LSM-tree has recently received significant interest (e.g., see [3, 15, 21, 22, 29, 35]) because of its advantages in terms of storage space usage and write amplification. If B+ tree has a very large cache memory (e.g., enough to hold the entire dataset) and uses very large redo log files, its write amplification could be much smaller than that of LSM-tree. Moreover, under large record size (e.g., 1KB and above), B+ tree tend to have smaller write amplification than LSM-tree. Hence, this work focuses on the scenarios where dataset is far bigger than the cache memory capacity and meanwhile the record size tends to be small (e.g., few hundred bytes or less), under which B+ tree tends to suffer from much higher write amplification than LSM-tree.

For the purpose of demonstration, we use RocksDB and WiredTiger as representatives of LSM-tree and B$^+$-tree, and carried out experiments on a 3.2TB storage drive with built-in transparent compression from ScaleFlux [31]. We run random write-only workloads with 128-byte record size over a 150GB dataset. For WiredTiger, we set its B$^+$-tree leaf page size as 8KB. Table 1 lists both the logical storage usage on the LBA space (i.e., before in-storage compression) and physical storage usage (i.e., after in-storage compression). Since LSM-tree has a more compact data structure, RocksDB has a smaller logical storage space usage than WiredTiger (i.e., 218GB vs. 280GB). Nevertheless, after in-storage transparent compression, WiredTiger consumes even less physical storage space than RocksDB, most likely due to the space amplification of LSM-tree. Fig. 4 shows the write amplification under different number of client threads. We measured the write amplification as the ratio between the volume of post-compression data being physically written to NAND flash memory inside the storage drive and the total amount of data written into database. The results show that RocksDB consistently has about 4× less write amplification than WiredTiger.

Table 1: Storage space usage comparison.

	Storage space usage	
	Logical	Physical
RocksDB	218GB	129GB
WiredTiger	280GB	104GB

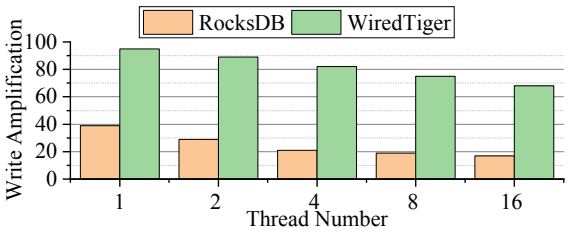


Figure 4: Measured write amplification.

The above results suggest that, with in-storage transparent compression, we could close the physical storage cost gap between B$^+$-tree and LSM-tree, while LSM-tree still maintains its significant advantage in terms of write amplification. The goal of this work is to further close the write amplification gap by appropriately modifying the B$^+$-tree implementation.

2.4 B$^+$-tree Write Amplification

Under current I/O interface protocols, storage devices only guarantee write atomicity over each 4KB LBA block. As a result, when the page size is larger than 4KB, B$^+$-tree must on its own ensure page write atomicity, which can be realized via two different strategies: (i) In-place page update: Although the convenient in-place update strategy simplifies the page storage management, B$^+$-tree must accordingly use page journaling (e.g., double-write buffer in MySQL) to survive partial page write failures, leading to about 2× higher write volume. (ii) Copy-on-write (or shadowing) page update: Although copy-on-write obviates the use of page journaling and readily supports snapshot, it complicates the page storage management. Meanwhile B$^+$-tree must employ certain mechanisms (e.g., page mapping table) to keep track of the page location, which still incurs extra storage write traffic.

Accordingly, we could classify B$^+$-tree storage write traffic into three categories: (1) logging writes that ensure transaction atomicity and isolation, (2) page writes that persist in-memory dirty B$^+$-tree pages to storage devices, and (3) extra writes that are induced by ensuring page write atomicity (e.g., page journaling in the case of in-place updates, or page mapping table persist in the case of page shadowing). Let W_{log}, W_{pg}, and W_e denote the total data write amount of these three categories, and W_{usr} denote the total amount of user data written into the B$^+$-tree. We can express the B$^+$-tree write amplification as

$$WA = \frac{W_{log}}{W_{usr}} + \frac{W_{pg}}{W_{usr}} + \frac{W_e}{W_{usr}} = WA_{log} + WA_{pg} + WA_e. \quad (1)$$

When B$^+$-tree runs on storage hardware with built-in transparent compression, let α_{log}, α_{pg}, and α_e denote the average compression ratio of the three categories of writes. Here we calculate the compression ratio by dividing the post-compression data volume with the before-compression data volume. Hence the compression ratio always falls into $(0,1]$, and a higher data compressibility leads to a smaller compression ratio. Therefore, the overall B$^+$-tree write amplification becomes

$$WA = \alpha_{log} \cdot WA_{log} + \alpha_{pg} \cdot WA_{pg} + \alpha_e \cdot WA_e. \quad (2)$$

3 Proposed Design Techniques

According to Eq. (2), we can reduce the B$^+$-tree write amplification by either reducing WA_{log}, WA_{pg}, and/or WA_e (i.e., reducing the B$^+$-tree write data volumes), or reducing α_{log}, α_{pg}, and/or α_e (i.e., improving the write data compressibility). By applying sparse data structure enabled by in-storage transparent compression, this section presents three design techniques to reduce the B$^+$-tree write amplification: (1) deterministic page shadowing that eliminates WA_e, (2) localized page modification logging that reduces both WA_{pg} and α_{pg}, and (3) sparse redo logging that reduces α_{log}.

3.1 Deterministic Page Shadowing

In order to eliminate WA_e, B$^+$-tree should employ the principle of page shadowing. Nevertheless, in conventional implementation of page shadowing, the new on-storage location of each updated B$^+$-tree page is dynamically determined during the runtime and must be recorded/persisted, leading to extra write overhead and management complexity. To eliminate the extra write overhead and meanwhile simplify the storage management, we propose a technique called *deterministic page shadowing* as illustrated in Fig. 5: Let l_{pg} denote the

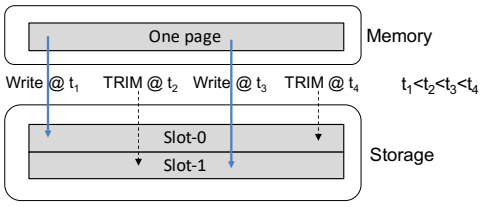


Figure 5: Illustration of deterministic page shadowing: two slots at the fixed location on the logical storage LBA space alternatively serve the memory-to-storage flush of one page.

B$^+$-tree page size (e.g., 8KB or 16KB). For each page, B$^+$-tree allocates $2l_{pg}$ amount of logical storage area on the LBA space and partitions it into two size-l_{pg} slots (slot-0 and slot-1). For each B$^+$-tree page, the two slots at the fixed location on the logical storage space serve memory-to-storage page flush alternatively in the ping-pong manner. Once a page has been flushed from memory into one slot, B$^+$-tree will issue a *TRIM* command over the other slot. This is conceptually the same as the conventional page shadowing with the difference that the location of the shadow page is now fixed. Although B$^+$-tree occupies $2\times$ larger logical storage space, only half of the storage space store valid data and the other half are trimmed (hence do not consume physical flash memory storage space). As pointed out above in Section 2.2, storage hardware with built-in transparent compression could expose a logical LBA storage space that is much larger than its internal physical storage capacity. Hence, such storage hardware can readily support the deterministic page shadowing. We note that deterministic page shadowing solely aims at ensuring page write atomicity without extra write overhead. To support multi-version concurrency control (MVCC), B$^+$-tree could use conventional methods such as undo logging.

With the proposed deterministic page shadowing, B$^+$-tree uses an in-memory bitmap to keep track of the valid slot for each page. Compared with page table being used in conventional page shadowing, bitmap consumes much less memory resource. Moreover, B$^+$-tree does not need to persist the bitmap. In case of system re-start, B$^+$-tree can gradually rebuild the in-memory bitmap: When B$^+$-tree loads one page for the first time, it reads both slots from the storage device. For the trimmed slot, storage device simply returns an all-zero block, based on which B$^+$-tree can easily identify the valid slot. When B$^+$-tree reads both slots of a page, the storage device internally only fetches the valid (i.e., untrimmed) slot from the physical storage media. Hence, compared with reading one slot, reading both slots will only incur more data transfer through the PCIe interface, without any extra read latency inside the storage device. This should not be an issue as the upcoming PCIe Gen5 will support 16GB/s$\sim$32GB/s, which is significantly larger than the back-end flash memory access bandwidth inside storage devices and hence can readily accommodate the extra data transfer. In case of system crash, B$^+$-tree needs to handle the following two possible scenarios: (i) A slot is partially written before the system crash: B$^+$-tree can easily identify the partially written slot by verifying the page checksum. (ii) A slot has been successfully written but the other slot has not been trimmed before the system crash: B$^+$-tree can identify the valid slot by comparing the page LSN (logical sequence number) of the pages on both slots. Since it is not necessary to persist the in-memory bitmap, deterministic page shadowing eliminates the $\alpha_e \cdot WA_e$ component from the total B$^+$-tree write amplification.

3.2 Localized Page Modification Logging

The second technique aims at reducing both α_{pg} and WA_{pg} components in Eq. (2). It is motivated by a simple observation: For a B$^+$-tree page, let Δ denote the difference between its in-memory image and on-storage image. If the difference is significantly smaller than the page size (i.e., $|\Delta| << l_{pg}$), we can largely reduce the write amplification by logging the

page modification Δ, instead of writing the entire in-memory page image, to the storage device. This is conceptually the same as the similarity-based data deduplication [2] and delta encoding [24]. Unfortunately, when B$^+$-tree runs on normal storage devices without built-in transparent compression, this approach is not practical due to significant operational overhead: Given the 4KB block IO interface, we must coalesce multiple Δ's from different pages into one 4KB LBA block in order to materialize the write amplification reduction. To enhance the gain, we should apply the page modification logging multiple times for each page, before resetting this process to construct the up-to-date on-storage page image. Accordingly, multiple Δ's associated with the same page will spread over multiple 4KB blocks on the storage device, which however will cause two problems: (1) For each page, B$^+$-tree must keep track of all its associated Δ's and also periodically carry out garbage collection, leading to a high storage management complexity. (2) To load a page from storage, B$^+$-tree has to read the existing on-storage page image and multiple Δ's from multiple non-contiguous 4KB LBA blocks, which leads to a long page load latency. Therefore, to our best knowledge, this simple design concept has not been used by real-world B$^+$-tree implementations ever reported in the open literature.

Storage hardware with built-in transparent compression for the first time makes the above simple idea practically viable. By applying sparse data structure enabled by such storage hardware, we no longer have to coalesce multiple Δ's from different pages into the same 4KB LBA block. Leveraging the abundant logical storage LBA space, for each B$^+$-tree page, we can simply dedicate one 4KB LBA block as its modification logging space to store the Δ, which is referred to as localized page modification logging. Under the 4KB I/O interface, to realize the proposed page modification logging for each page, B$^+$-tree writes $D = [\Delta, \mathbf{O}]$ (where $\mathbf{O}$ represents an all-zero vector, and $|D|$ is 4KB) to the 4KB block associated with the page. Inside the storage device, all the zeros in D will be compressed away and only the compressed version of Δ will be physically stored. Therefore, when serving each memory-to-storage page flush with page modification logging, we reduce WA_{pg} by writing 4KB instead of l_{pg} amount of data to the logical storage LBA space, and reduce the compression ratio α_{pg} since the written data $[\Delta, \mathbf{O}]$ can be highly compressed by the storage device. By dedicating one 4KB modification logging space for each B$^+$-tree page, we do not incur extra B$^+$-tree storage management complexity. The read amplification is small for two main reasons: (1) B$^+$-tree always reads only one additional 4KB LBA block. Moreover, each page and its associated 4KB logging block contiguously reside on the LBA space. Hence, in order to read both the page and its associated 4KB logging block, B$^+$-tree only issues a single read request to the storage device. (2) The storage device internally fetches very small amount of data from flash memory in order to reconstruct the 4KB LBA block $[\Delta, \mathbf{O}]$.

To practically implement this simple idea, B$^+$-tree must carry out two extra operations: (1) To load a page from storage into memory, B$^+$-tree must construct the up-to-date page image based on the on-storage page image and Δ. (2) To flush a page from memory to storage, B$^+$-tree must obtain Δ and accordingly decide whether it should invoke the page modification logging. To minimize the B$^+$-tree operational overhead, we propose the following implementation strategy: Let P_m and P_s denote the in-memory and on-storage images of one B$^+$-tree page. We logically partition P_m and P_s into k segments, i.e., $P_m = [P_{m,1}, \cdots, P_{m,k}]$ and $P_s = [P_{s,1}, \cdots, P_{s,k}]$, and $|P_{m,i}| = |P_{s,i}|\ \forall i$ (i.e., the two segments $P_{m,i}$ and $P_{s,i}$ at the same position have the same size). For each page, B$^+$-tree keeps a k-bit vector $f = [f_1, \cdots, f_k]$, where f_i is set to 1 if $P_{m,i} \neq P_{s,i}$. Accordingly, we construct Δ by concatenating all the in-memory segments $P_{m,i}$ with $f_i = 1$. During the runtime, whenever the i-th segment in one in-memory page is modified, B$^+$-tree will set its corresponding f_i as 1. When B$^+$-tree flushes a page from memory to storage, it first calculates the size of Δ as

$$|\Delta| = \sum_{\forall i, f_i = 1} |P_{m,i}|. \tag{3}$$

We define a fixed threshold T that is not larger than 4KB. If $|\Delta| \leq T$, then B$^+$-tree will invoke the page modification logging, where Δ can be obtained through simple memory-copy operations. We note that the k-bit vector f should be written together with Δ into the dedicated 4KB page modification logging block. When B$^+$-tree loads a page from storage into memory, it fetches $l_{pg} + 4KB$ amount of data from the storage device, where the size-l_{pg} space contains the current on-storage page image P_s and the additional 4KB block contains the associated f and Δ. Accordingly, we could easily construct the up-to-date page image through simple memory-copy operations. For each B$^+$-tree page, the size of its Δ will monotonically increase as B$^+$-tree undergoes more write operations. Once $|\Delta|$ becomes larger than the threshold T, we will reset the process by flushing the entire up-to-date page to storage with $\Delta = \emptyset$ and f being an all-zero vector. We note that the threshold T configures the trade-off between write amplification reduction and storage space amplification: As we increase the value of T, we can less frequently reset the page modification logging process, leading to a smaller write amplification. Meanwhile, under a larger value of T, more page modifications will accumulate in the logging space and cause a larger storage cost overhead.

Fig. 6 further illustrates this implementation strategy. Among the all the k segments, the first segment $P_{m,1}$ is the page header and the last segment $P_{m,k}$ is the page trailer, both of which can be much smaller than the other segments. Suppose a page update causes modification of the segment $P_{m,3}$ and page header/trailer. When B$^+$-tree evicts this page from the memory, it constructs the Δ as $[P_{m,1}, P_{m,3}, P_{m,k}]$, and writes Δ and the k-bit vector f to the dedicated 4KB block logging block, which is further compressed inside the storage device.

We note that, if B$^+$-tree treats in-memory pages as im-

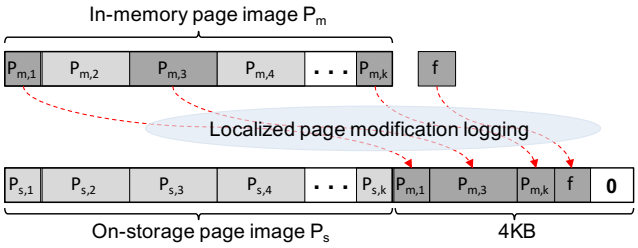


Figure 6: Illustration of the localized page modification logging, where the to-be-flushed in-memory page P_m contains three modified segments $P_{m,1}$, $P_{m,3}$, and $P_{m,k}$.

mutable and uses in-memory delta chaining to keep track of the in-memory page modification (which is used in the Bw-tree [19, 20] to achieve latch-free operations), we can most likely further reduce $|\Delta|$ and hence improve the effectiveness of the localized page modification logging on reducing the write amplification. However, such delta-chaining approach can largely complicate the B$^+$-tree implementation [32] and incur noticeable memory usage overhead. Hence, this work chooses the above simple intra-page segment-based tracking approach in our implementation and evaluation.

3.3 Sparse Redo Logging

The third design technique aims at reducing the component α_{log} in Eq. (2) (i.e., improving the redo log data compressibility). To maximize the reliability, B$^+$-tree flushes the redo log with *fsync* or *fdatasync* at every transaction commit. In order to reduce the log-induced storage overhead, conventional practice always tightly packs log records into the redo log. As a result, multiple consecutive redo log flushes may write to the same LBA block on the storage device, especially when transaction records are significantly smaller than 4KB and/or the workload concurrency is not very high. This can be illustrated in Fig. 7: Suppose three transactions TRX-1, TRX-2, and TRX-3 (with log records L_1, L_2, and L_3) commit at the time t_1, t_2, and t_3, respectively, where $t_1 < t_2 < t_3$. As illustrated in Fig. 7, at the time t_1, 4KB data $[L_1, \mathbf{O}]$ is flushed from the in-memory redo log buffer to the LBA 0x0001 on the storage device that further internally compresses the data. Later on, the log record L_2 is appended into the redo log buffer, and at the time t_2, the 4KB data $[L_1, L_2, \mathbf{O}]$ is flushed to the same LBA 0x0001 on the storage device. Similarly, at the time t_3, the 4KB data $[L_1, L_2, L_3, \mathbf{O}]$ is flushed to the same LBA 0x0001 on the storage device. As illustrated in Fig. 7, the same log record (e.g., L_1 and L_2) are written to the storage device multiple times, leading to a higher write amplification. Equivalently, as more log records are accumulated inside each 4KB redo log buffer block, the redo log data compression ratio α_{log} will become worse and worse over the multiple consecutive redo log flushes.

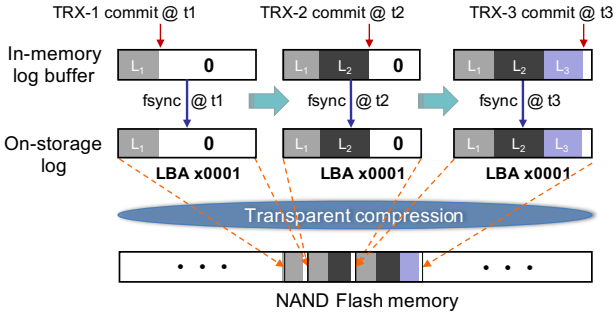


Figure 7: Conventional implementation of redo logging where log records are tightly packed into redo log and consecutive transactions commits could flush redo log to the same LBA (e.g., LBA 0x0001 in this example) multiple times.

By applying sparse data structure enabled by storage hardware with built-in transparent compression, we propose a design technique called sparse redo logging that can enable the storage hardware most effectively compress the redo log and hence reduce the logging-induced write amplification. Its basic idea is very simple: At each transaction commit and its corresponding redo log memory-to-storage flush, we always pad zeros into the in-memory redo log buffer to make its content 4KB-aligned. As a result, the next log record will be written into a new 4KB space in the redo log buffer. Therefore, each log record will be written to the storage device only once, leading to a lower write amplification compared with the conventional practice. This can be further illustrated in Fig. 8: Assuming the same scenario as shown above in Fig. 7, after the transaction TRX-1 commits at the time t_1, we pad zeros into the redo log buffer and flush the 4KB data $[L_1, \mathbf{O}]$ to the LBA 0x0001 on the storage device. Subsequently, we put the next log record L_2 in a new 4KB space in the redo log buffer. At the time t_2, the 4KB data $[L_2, \mathbf{O}]$ is flushed to a new LBA 0x0002 on the storage device. Similarly, at the time t_3, the 4KB data $[L_3, \mathbf{O}]$ is flushed to another new LBA 0x0003 on the storage device. Clearly, each redo log record

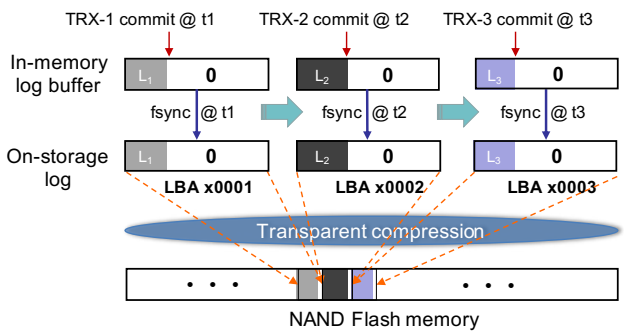


Figure 8: Illustration of the proposed sparse logging where each redo log flush always writes to a new LBA block.

is written to the storage device only once, and redo log writes can be (much) better compressed by the storage hardware, leading to a (much) smaller α_{log} and hence lower write amplification. Since each transaction commit always invokes one 4KB write to the storage device in both conventional logging and proposed sparse logging, the total redo log write volume W_{log} in Eq. (2) will remain the same. Therefore, by reducing the log compression ratio α_{log}, the proposed sparse logging reduces the component $\alpha_{log} \cdot W_{log}$ in the total B$^+$-tree write amplification.

4 Evaluation

For the purpose of demonstration, we implemented a B$^+$-tree (referred to as B$^-$-tree) that incorporates our proposed three simple design techniques. To facilitate the comparison, we also implemented a baseline B$^+$-tree that uses the conventional page shadowing, where we persist the page table after each page flush. Since the proposed three design techniques mainly confine within the I/O module and are largely orthogonal to the other modules in B$^+$-tree implementation, we obtained the B$^-$-tree by simply integrating the proposed design techniques into the baseline B$^+$-tree with 1,200 LoC added/modified. Moreover, we also considered RocksDB and WiredTiger as representatives of LSM-tree and normal B$^+$-tree. For RocksDB, we set its maximum number of compaction and flush threads as 12 and 4, and set the Bloomfilter as 10 bits per record. For WiredTiger and our own baseline B$^+$-tree and B$^-$-tree, we use 4 background write threads that flush dirty in-memory pages to the storage device.

4.1 Experimental Setup

We ran all the experiments on a server with 24-core 2.6GHz Intel CPU, 64GB DDR4 DRAM, and a 3.2TB ScaleFlux computational storage drive with built-in transparent compression. This 3.2TB drive carries out hardware-based zlib compression on each 4KB block directly along the internal I/O path, being transparent to the host. The per-4KB (de)compression latency of the hardware zlib engine is around 5μs, which is over 10$\times$ shorter than the TLC/QLC NAND flash memory read latency ($\sim$50μs and above) and write latency ($\sim$1ms and above). Operating with PCIe Gen3$\times$4 interface, this computational storage drive can achieve up to 3.2GB/s sequential throughput and 650K (520K) random 4KB read (write) IOPS (I/O per second) over 100% LBA span. In comparison, leading-edge commodity NVMe SSDs (e.g., Intel P4610) achieve similar sequential throughput and random 4KB read IOPS, but have much worse random 4KB write IOPS (e.g., below 300K). This is because built-in transparent compression can significantly reduce the garbage collection overhead inside the storage drive. This computational storage drive is already in volume production and has been deployed in data centers worldwide.

This computational storage drive can report the amount of post-compression data being physically written to the NAND flash memory, which are used in the calculation of write amplification. Before measuring the write amplification for each case, we populate the B$^+$-tree/LSM-tree data store by inserting all the data records in a fully random order. Once after the data store has been fully populated, we subsequently run random write-only workloads over one hour in order to measure the write amplification. In all our experiments, we generate the content of each record by filling its half content as all-zero and the other half content as random bytes in order to mimic the runtime data content compressibility.

We note that the effectiveness of the proposed sparse redo logging strongly depends on the redo log flush policy. As discussed above Section 3.3, when redo log flushes at every transaction commit to maximize the system reliability, sparse redo logging is very effective. However, for applications that can tolerate the loss of certain amount of most recent data, one could relax the redo log flush policy (e.g., flush every one minute) under which the proposed sparse redo logging will be much less useful. Therefore, we considered two scenarios in our evaluation: (1) redo log flush per transaction commit (denoted as *log-flush-per-commit*), and (2) redo log flush per minute (denoted as *log-flush-per-minute*).

4.2 Experiments with Log-Flush-Per-Minute

We first carried out experiments without taking into account of the benefit of sparse redo logging by setting the redo log flush policy as per-minute. We considered two different dataset size: (1) 150GB dataset with 1GB cache memory, and (2) 500GB dataset with 15GB cache memory. We also considered three different record size (including 8B key): 128B, 32B, and 16B. For B$^+$-tree implementations, following the popular RDBMs such as Oracle and MySQL, we considered two different page size, including 8KB and 16KB. For our B$^-$-tree, the implementation of the proposed page modification logging involves the following two parameters: (1) the threshold T that determines the maximum $|\Delta|$ per page, and (2) the segment size (denoted as D_s) when partitioning each page into multiple segments for tracking page modification, as discussed in Section 3.2.

Fig. 9 and Fig. 10 show the measured write amplification for 150GB and 500GB datasets, respectively. In each experiment, we use either 1, 2, 4, 8, or 16 client threads to cover a wide range of runtime workload concurrency. For B$^-$-tree, we set the threshold T as 2KB, and set the segment size D_s as either 128B or 256B. Since both WiredTiger and our own baseline B$^+$-tree use page shadowing, they have very similar write amplification as shown in Fig. 9 and Fig. 10. Compared with RocksDB, normal B$^+$-tree (i.e., WiredTiger and our own baseline B$^+$-tree) has a much larger write amplification, while our B$^-$-tree can essentially close the B$^+$-tree vs. LSM-tree write amplification gap. For example, in the case of 500GB dataset and 32B record size and 4 client threads, the write am-

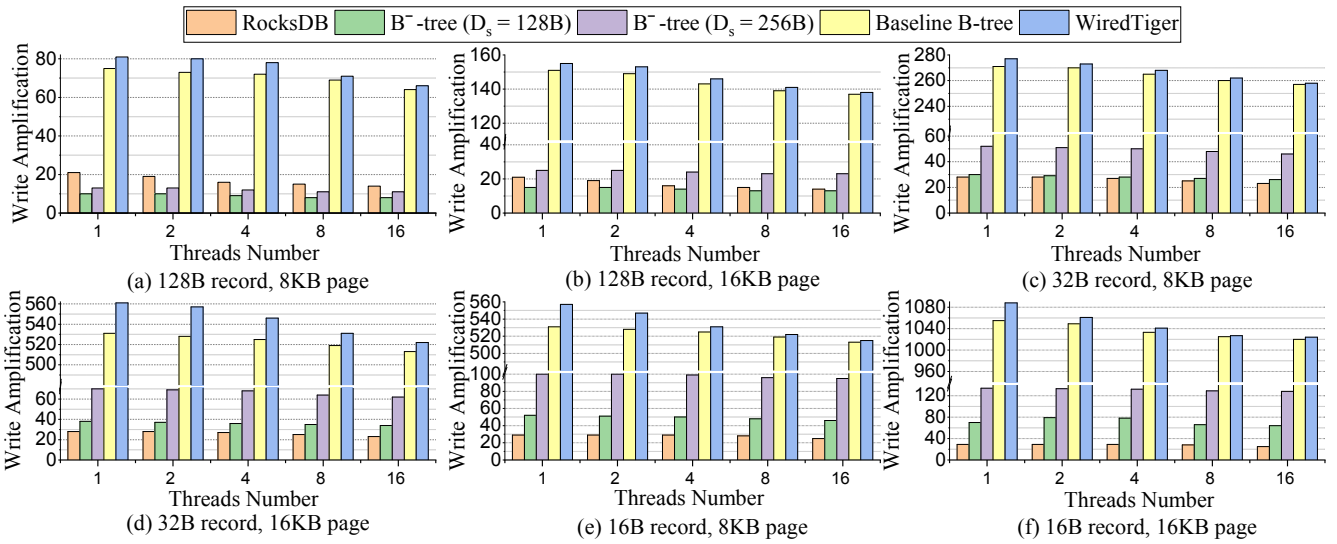


Figure 9: Write amplification under the log-flush-per-minute policy, where the dataset size is 150GB and cache size is 1GB.

plification of RocksDB is 38, while the write amplification of WiredTiger is 268 under 8KB page size and 530 under 16KB page size, respectively, which are 7.1× and 13.9× larger than that of RocksDB. In comparison, the write amplification of B⁻-tree with D_s=128B is 28 under 8KB page size (which is only 73.7% of RocksDB's write amplification) and 36 under 16KB page size (which is almost the same as RocksDB).

As shown in both Fig. 9 and Fig. 10, the write amplification of both normal B⁺-tree and B⁻-tree will increase as we reduce the record size (e.g., from 128B per record to 16B per record) and/or increase the B⁺-tree page size (i.e., from 8KB to 16KB). Since we use the log-flush-per-minute policy, the overall write amplification of both normal B⁺-tree and B⁻-tree tends to be dominated by the $\alpha_{pg} \cdot WA_{pg}$, as shown in Eq. (2). In the case of normal B⁺-tree, WA_{pg} proportionally increases as we reduce the record size and/or increase the page size. Therefore, the write amplification of normal B⁺-tree almost linearly scale with the page size and the inverse of the record size. In the case of B⁻-tree, its $\alpha_{pg} \cdot WA_{pg}$ not only depends on the record size and page size, but also depends on the threshold T and segment size D_s. Hence, the write amplification of B⁻-tree tends to sub-linearly scale with the page size and the inverse of the record size, as shown in both Fig. 9 and Fig. 10. In contrast, due to the nature of LSM-tree, the write amplification of RocksDB is weakly dependent on the record size.

As the number of client threads increases, the write amplification of normal B⁺-tree noticeably reduces, because of the larger probability of page flush coalescing under higher workload concurrency. In comparison, the write amplification of B⁻-tree is much more weakly dependent on the number of client threads, because the probability that different client threads modify the same segment inside a page is much smaller than the probability that different client threads modify the same page. Moreover, the write amplification of B⁻-tree increases as we increase the segment size D_s, simply because the page modification logging is done in the unit of segments. The impact of segment size D_s on the write amplification is more significant under smaller record size, as shown in both Fig. 9 and Fig. 10.

The write amplification of LSM-tree may noticeably increase as the dataset size increases, which can be observed by comparing the results in Fig. 9 and Fig. 10. This is because a larger dataset size results in more levels in LSM-tree, while the write amplification of LSM-tree tends to be proportional to the number of levels. In contrast, the write amplification of B⁺-tree is very weakly dependent on the dataset size. As a result, the write amplification comparison of RocksDB vs. B⁻-tree is noticeably different between the 150GB dataset and 500GB dataset. In the case of 150GB dataset as shown in Fig. 9, the write amplification of RocksDB can be up to 2× larger than that of B⁻-tree (under 128B per record and 8KB page size), and can be up to 4× smaller than that of B⁻-tree (under 16B per record and 16KB page size). In comparison, in the case of 500GB dataset as shown in Fig. 10, the write amplification of RocksDB can be up to 3× larger than that of B⁻-tree (under 128B per record and 8KB page size), and can be up to 2× smaller than that of B⁻-tree (under 16B per record and 16KB page size). The results clearly show that, even without taking into account of the effectiveness of sparse redo logging, the proposed B⁻-tree can already close the write amplification gap between B⁺-tree and LSM-tree.

4.3 Experiments with Log-Flush-Per-Commit

We carried out further experiments by switching to the log-flush-per-commit policy, under which the proposed sparse

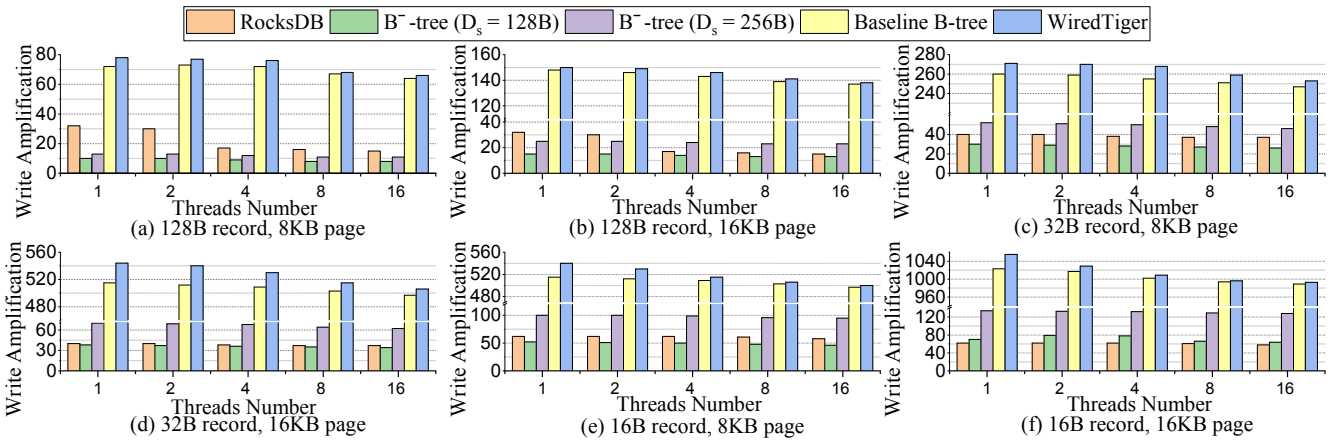


Figure 10: Write amplification under the log-flush-per-minute policy, where the dataset size is 500GB and cache size is 15GB.

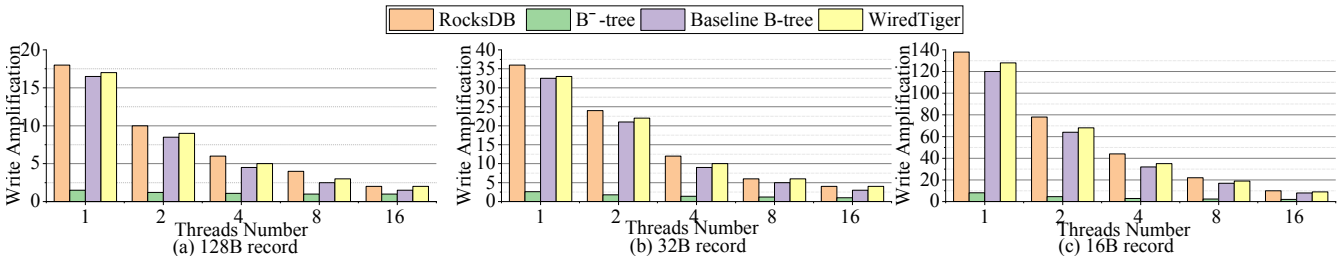


Figure 11: Log-induced write amplification when using the log-flush-per-commit policy.

redo logging can noticeably contribute to reducing the write amplification. First, Fig. 11 shows the measured write amplification caused by the log flush, i.e., the $\alpha_{log} \cdot WA_{log}$ component in Eq. 2. Given the record size, except the case of B$^-$-tree, the log-induced write amplification significantly reduces as we increase the number of client threads. This is because, under higher workload concurrency, more transaction commits can be coalesced in each log flush. In contrast, the log-induced write amplification of B$^-$-tree is much more weakly dependent on the number of client threads, because of its use of the sparse redo logging. As the record size reduces, the log-induced write amplification almost proportionally increases when the sparse redo logging is not being used. The results in Fig. 11 clearly demonstrate the effectiveness of the proposed sparse redo logging design technique when data management systems use the log-flush-per-commit policy to improve the data reliability.

Fig. 12 further shows the total write amplification under the log-flush-per-commit policy, where the dataset size is 150GB and cache size is 1GB. Compared with the experiments under the log-flush-per-minute policy (as shown in Fig. 9), the write amplification of B$^-$-tree remains almost the same, while the write amplification of the other three cases (i.e., RocksDB, our own baseline B$^+$-tree, and WiredTiger) noticeably increases, especially when the number of client threads is small, because of the higher log-induced write amplification. As a result, B$^-$-tree can more effectively close the B$^+$-tree vs. LSM-tree write amplification gap and be able to achieve better-than-RocksDB write amplification under more scenarios.

4.4 Impact of Threshold T

As discussed earlier in Section 3.2, the proposed page modification logging design approach is subject to a write amplification vs. storage usage trade-off that is configured by the threshold $T \in (0, 4\text{KB}]$. As we increase the value of T, we can pack more modification logs into each dedicated 4KB log space in order to further reduce the total write amplification, which nevertheless meanwhile induces higher storage usage overhead. All the experiments above were carried out with T as 2KB. We carried out further experiments under different values of threshold T to study its impact on the write amplification vs. storage usage trade-off. For each B$^+$-tree page P_i, let $|\Delta_i|$ denote the size of its associated modification log. Let N denote the total number of B$^+$-tree pages and recall that l_{pg} denotes the page size, we can express the average storage usage overhead factor as

$$\beta = \frac{\sum_{i=1}^{N} |\Delta_i|}{N \cdot l_{pg}}. \quad (4)$$

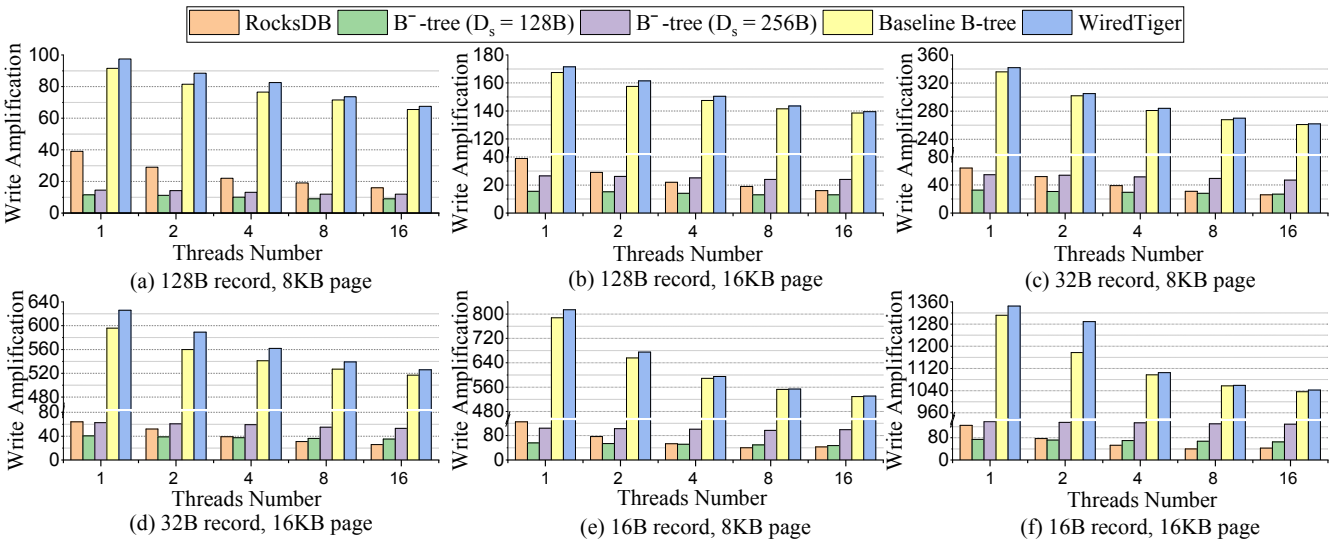


Figure 12: Write amplification under the log-flush-per-commit policy, where the dataset size is 150GB and cache size is 1GB.

Under a sufficiently large N, the value of β mainly depends on the page size l_{pg}, the threshold T, and the workload characteristics (in particular the write request distribution over all the pages). It also weakly depends on the segment size D_s. Assuming the fully random write request distribution across all the pages, we carried out experiments to measure the average value of β, and the results are summarized below in Table 2. The results clearly show that the storage usage overhead will reduce as we reduce the threshold T and/or increase the page size. In comparison, the impact of the segment size D_s is much more insignificant.

Table 2: Storage usage overhead factor β of B⁻-tree.

Page size	D_s	Threshold T		
		4KB	2KB	1KB
8KB	128B	27.0%	12.4%	5.6%
	256B	26.3%	11.5%	4.8%
16KB	128B	12.7%	6.0%	2.8%
	256B	12.3%	5.6%	2.3%

Fig. 13 further compares the total storage usage in terms of both logical storage usage on the LBA space (i.e., before in-storage compression) and physical usage of flash memory (i.e., after in-storage compression). Since LSM-tree has a more compact data structure than B⁺-tree, RocksDB has a (much) smaller logical storage usage than the others as shown in Fig. 13. Since B⁻-tree allocates one 4KB block for each page in order to implement the localized modification logging, its logical storage usage is much larger than that of normal B⁺-tree. Nevertheless, after the in-storage compression, WiredTiger and our baseline B⁺-tree consume less physical flash memory capacity than RocksDB (most likely because of the space amplification of LSM-tree) and B⁻-tree (because of the storage overhead caused by page modification logging). Due to the storage space overhead caused by page modification logging, B⁻-tree has slightly larger physical storage usage than RocksDB. For example, in the case of 500GB dataset size, the physical storage usage of RocksDB is 431GB, while the physical storage usage of B⁻-tree with T=2KB is 452GB, only about 5% larger than that of RocksDB.

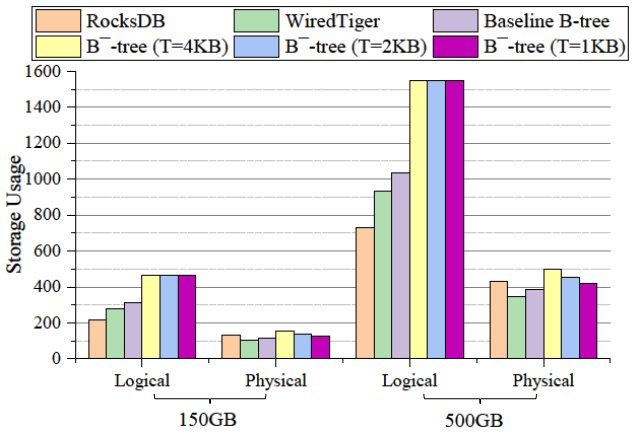


Figure 13: Comparison of logical and physical storage space usage where B⁺-tree page size is 8KB.

Fig. 14 compares the write amplification of B⁻-tree under different value of the threshold T, where we use the log-flush-per-minute policy in order to better show the impact of T. The segment size D_s is 128B. The results clearly show that we can reduce the write amplification by increasing the threshold T. Moreover, the reduction on the write amplification tends to become less and less as we continue to increase the threshold T. This is because, as the page modification log size $|\Delta|$

becomes larger, the write amplification caused by flushing the modification log will accordingly increase. Combining the results shown in Fig. 13 and Fig. 14, we can observe the impact of the threshold T on the trade-off between the write amplification and storage usage overhead. The setting of T=2KB appears to achieve a reasonable balance on the trade-off and hence has been used in all the experiments presented above in Sections 4.2 and 4.3.

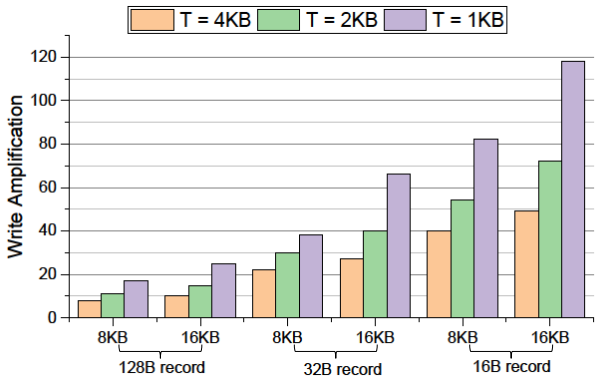


Figure 14: B$^-$-tree write amplification under different T.

4.5 Speed Performance Evaluation

Finally, we studied the speed performance of B$^-$-tree. Compared with normal B$^+$-tree, B$^-$-tree tends to have lower read speed performance because of the following two overheads when fetching each page from the storage: (1) B$^-$-tree has to fetch an extra 4KB block from the storage, and (2) B$^-$-tree has to consolidate the modification log with the current on-storage page image in order to construct the up-to-date in-memory page image. Using the 150GB dataset with 128B per record as the test vehicle, we run random read-only workloads with either point read or range scan queries. The B$^+$-tree page size is 8KB in all the experiments. Fig. 15 shows the measured TPS performance under random point read queries. The results show that normal B$^+$-tree (WiredTiger and our own baseline B$^+$-tree) have the best point read throughput performance. RocksDB and B$^-$-tree achieve almost the same random point read throughput performance. By using the Bloomfilter, RocksDB almost completely obviates the read amplification problem of classical LSM-tree. Nevertheless, when serving read requests, RocksDB still has to search the memtable and check the Bloomfilter. As shown in Fig. 15, the point read throughput gap between normal B$^+$-tree and RocksDB/B$^-$-tree is not significant. For example, under 16 client threads, WiredTiger can achieve 71K TPS, while RocksDB/B$^-$-tree can achieve 57K TPS, about 19.7% less than that of WiredTiger.

Fig. 16 shows the measured TPS when running random range scan queries, where each range scan covers 100 consecutive records. Compared with the case of random point reads, the normal B$^+$-tree and B$^-$-tree have noticeably smaller difference in terms of range scan throughput performance. This is because the two overheads of B$^-$-tree (i.e., fetching an extra 4KB, and in-memory page reconstruction) can be amortized among the records covered by each range scan. In comparison, RocksDB has noticeably worse range scan throughput performance than the others, because range scan invokes reads over all the levels in LSM-tree, leading to very high read amplification.

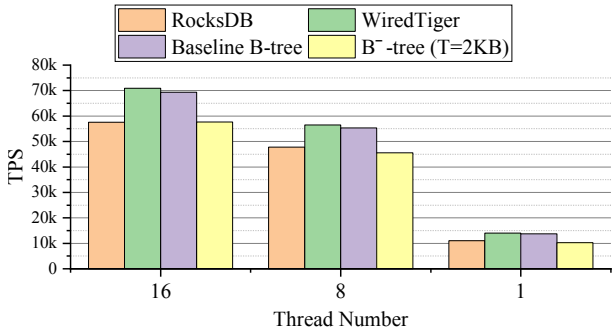


Figure 15: Random point read speed performance measured on 150GB dataset with 1GB cache and 128B per record.

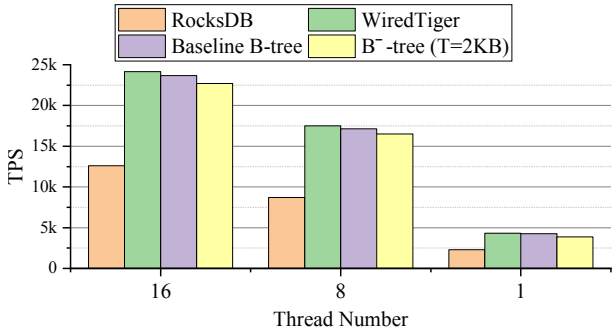


Figure 16: Random range scan speed performance measured on 150GB dataset with 1GB cache and 128B per record, where each range scan covers 100 consecutive records.

We also studied the speed performance under random write-only workloads. The random write speed performance of B$^+$-tree and LSM-tree is fundamentally limited by the write amplification. Therefore, by significantly reducing the write amplification, B$^-$-tree should be able to achieve much higher write speed performance. Fig. 17 shows the measured random write TPS on 150GB dataset with 128B per record, where the B$^+$-tree page size is 8KB. We set the log-flush-per-minute policy in the experiments. Even without the help of the sparse redo logging, B$^-$-tree achieves 19% higher write throughput than RocksDB, and about 2.1× higher write throughput than WiredTiger and our baseline B$^+$-tree. Although the workload

is write-only, the I/O traffic is heavily read/write-mixed because the cache memory capacity is much smaller than the total dataset. Because the localized page modification logging invokes read-modify-write operations, our B$^-$-tree incurs higher read I/O traffic than normal B$^+$-tree. As a result, the TPS gain of our B$^+$-tree is less than the WA reduction of B$^+$-tree as shown above in Fig. 9. Nevertheless, the random write speed results still correlate with the write amplification results, and our B$^+$-tree can achieve the highest random write speed performance.

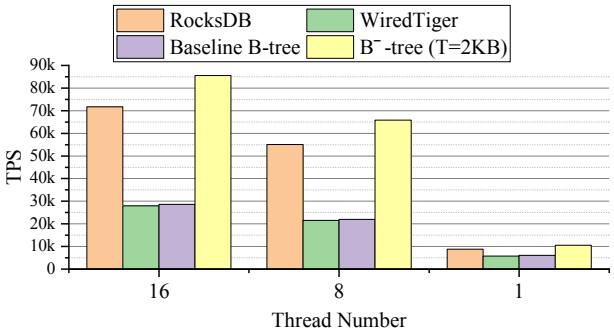


Figure 17: Random write speed performance measured on 150GB dataset with 1GB cache and 128B per record.

5 Related Work

Graefe [11] surveyed a variety of design techniques (e.g., I/O optimization, buffering, and relaxing transaction guarantee) that can improve the B$^+$-tree write throughput, some of which accomplish the goal by reducing the B$^+$-tree write amplification. Nevertheless, I/O optimization techniques that mainly aim at converting random page writes to sequential page writes are only useful to HDDs, since modern SSDs achieve almost the same random vs. sequential write speed performance. Many techniques surveyed in [11] (e.g., buffering, relaxing transaction guarantee) are orthogonal to the solutions presented in this paper, and hence can be applied altogether to further reduce the B$^+$-tree write amplification. Moreover, copy-on-write or page shadowing [1, 18] is a well-known technique to achieve B$^+$-tree data atomicity and durability. Compared with B$^+$-tree using in-place update, it can reduce the write amplification by about 2×.

Levandoski *et al.* [19, 20] proposed the Bw-tree that can better adapt to modern multi-core CPU architecture and meanwhile reduce the write amplification. Bw-tree treats each in-memory page as immutable and uses delta chaining to keep track of the changes made to each page. This can enable latch-free operations and hence better utilize multi-core CPUs. Meanwhile, by only flushing the delta records, Bw-tree can reduce the write amplification. Bw-tree uses a log-structured store to persist all the pages and deltas, which however suffers from read amplification and background garbage collection overheads. When running Bw-tree on storage hardware with build-in transparent compression, one could enhance Bw-tree by replacing the log-structured store with the localized page modification logging presented in this work.

B$^\varepsilon$-tree [4] is another well-known variant of B$^+$-tree that can significantly reduce the write amplification through data buffering at non-leaf nodes. It has been used in the design of filesystem [10, 16, 17, 34] and key-value store [8, 26]. In essence, B$^\varepsilon$-tree cleverly mixes the key design principles of B$^+$-tree and LSM-tree. Similar to LSM-tree, B$^\varepsilon$-tree has worse range scan speed performance than B$^+$-tree. Percona TokuDB [27] is one publicly known database product that is built upon B$^\varepsilon$-tree.

Little prior research has been done on studying how data management systems could take advantage of modern storage hardware with built-in transparent compression. Recently, Zheng *et al.* [36] discussed some possible options on leveraging such modern storage hardware to improve data management software design. Chen *et al.* [6] presented a hash-based key-value store that can leverage such modern storage hardware to obviate the use of costly in-memory hash table.

6 Conclusions

This paper presents three simple yet effective design techniques that enable B$^+$-tree take better advantages of modern storage hardware with built-in transparent compression. By decoupling logical vs. physical storage space utilization efficiency, such modern storage hardware allows data management systems employ sparse data structure without sacrificing the true physical data storage cost. This opens a new but largely unexplored spectrum of opportunities to innovate data management software design. As one small step towards exploring this design spectrum, this paper presents three design techniques that can appropriately embed sparsity into B$^+$-tree data structure to largely reduce the B$^+$-tree write amplification. Experimental results show that the proposed design techniques can reduce the B$^+$-tree write amplification by over 10×, which essentially closes the B$^+$-tree vs. LSM-tree gap in terms of write amplification. This work suggests that the arrival of such new storage hardware warrants a revisit on the role and comparison of B$^+$-tree and LSM-tree in future data management systems.

Acknowledgments

We would like to thank our shepherd Randal Burns and the anonymous reviewers for their insight and suggestions that help us to improve the quality and presentation of this paper. This work was supported by the National Science Foundation under Grant No. CNS-2006617.

References

[1] R. Agrawal and D. J. Dewitt. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Transactions on Database Systems (TODS)*, 10(4):529–564, 1985.

[2] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein. The design of a similarity based deduplication system. In *Proceedings of ACM International Systems and Storage Conference (SYSTOR)*, pages 1–14, 2009.

[3] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 363–375, 2017.

[4] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *SODA*, volume 3, pages 546–554. Citeseer, 2003.

[5] I. Burstein. Nvidia Data Center Processing Unit (DPU) Architecture. In *IEEE Hot Chips Symposium (HCS)*, pages 1–20, 2021.

[6] X. Chen, N. Zheng, S. Xu, Y. Qiao, Y. Liu, J. Li, and T. Zhang. KallaxDB: A table-less hash-based key-value store on storage hardware with built-in transparent compression. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–10, 2021.

[7] D. Chiou, E. Chung, and S. Carrie. (Cloud) Acceleration at Microsoft. *Tutorial at Hot Chips*, 2019.

[8] A. Conway, A. Gupta, V. Chidambaram, M. Farach-Colton, R. Spillane, A. Tai, and R. Johnson. SplinterDB: Closing the bandwidth gap for nvme key-value stores. In *USENIX Annual Technical Conference (ATC)*, pages 49–63, 2020.

[9] Dell EMC PowerMax. https://delltechnologies.com/.

[10] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. The TokuFS streaming file system. In *HotStorage*, 2012.

[11] G. Graefe. B-tree indexes for high update rates. *ACM Sigmod Record*, 35(1):39–44, 2006.

[12] G. Graefe and H. Kuno. Modern B-tree techniques. In *IEEE International Conference on Data Engineering*, pages 1370–1373. IEEE, 2011.

[13] E. F. Haratsch. SSD with Compression: Implementation, Interface and Use Case. In *Flash Memory Summit*, 2019.

[14] HPE Nimble Storage. https://www.hpe.com/.

[15] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 651–665. ACM, 2019.

[16] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, et al. BetrFS: A right-optimized write-optimized file system. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, 2015.

[17] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, et al. BetrFS: Write-optimization in a kernel file system. *ACM Transactions on Storage (TOS)*, 11(4):1–29, 2015.

[18] J. Kent, H. Garcia-Molina, and J. Chung. An experimental evaluation of crash recovery machanisms. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 113–122, 1985.

[19] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *IEEE International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

[20] J. J. Levandoski, S. Sengupta, and W. Redmond. The Bw-tree: A latch-free B-tree for log-structured flash storage. *IEEE Data Eng. Bull.*, 36(2):56–62, 2013.

[21] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):5, 2017.

[22] C. Luo and M. Carey. LSM-based storage techniques: a survey. *The VLDB Journal*, 29:393–418, 2020.

[23] LZ4. https://github.com/lz4/.

[24] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication*, pages 181–194, 1997.

[25] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[26] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. Tucana: Design and implementation of a fast

and efficient scale-up key-value store. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 537–550, 2016.

[27] Percona TokuDB. https://www.percona.com/software/mysql-database/percona-tokudb.

[28] Pure Storage FlashBlade. https://purestorage.com/.

[29] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 497–514, 2017.

[30] RocksDB. https://github.com/facebook/rocksdb.

[31] ScaleFlux Computational Storage. http://scaleflux.com.

[32] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a Bw-tree takes more than just buzz words. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 473–488, 2018.

[33] WiredTiger. https://github.com/wiredtiger/.

[34] J. Yuan, Y. Zhan, W. Jannen, P. Pandey, A. Akshintala, K. Chandnani, P. Deo, Z. Kasheff, L. Walsh, and M. Bender. Optimizing every operation in a write-optimized file system. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2016.

[35] Y. Yue, B. He, Y. Li, and W. Wang. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):961–973, 2016.

[36] N. Zheng, X. Chen, J. Li, Q. Wu, Y. Liu, Y. Peng, F. Sun, H. Zhong, and T. Zhang. Re-think data management software design upon the arrival of storage hardware with built-in transparent compression. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2020.

[37] zlib. http://zlib.net.

[38] Zstandard (ZSTD). https://github.com/facebook/zstd.

TVStore: Automatically Bounding Time Series Storage via Time-Varying Compression

Yanzhe An[1], Yue Su[2†], Yuqing Zhu[1‡], Jianmin Wang[1]
[1]Tsinghua University, [2]Huawei Technologies Co., Ltd.

Abstract

A pressing demand emerges for storing extreme-scale time series data, which are widely generated by industry and research at an increasing speed. Automatically constraining data storage can lower expenses and improve performance, as well as saving storage maintenance efforts at the resource-constrained conditions. However, two challenges exist: 1) how to preserve data as much and as long as possible within the storage bound; and, 2) how to respect the importance of data that generally changes with data age.

To address the above challenges, we propose time-varying compression that respects data values by compressing data to functions with time as input. Based on time-varying compression, we prove the fundamental design choices regarding when compression must be initiated to guarantee bounded storage. We implement a storage-bounded time series store TVStore based on an open-source time series database. Extensive evaluation results validate the storage-boundedness of TVStore and its time-varying pattern of compression on both synthetic and real-world data, as well as demonstrating its efficiency in writes and queries.

1 Introduction

Time series databases are becoming the most popular type of databases in recent years [2]. We are witnessing a growing demand for time-series-specific storage and processing from many fields such as cluster monitoring [91], Internet of Things [6], finances [80], medicine [51], and scientific research [63]. In fact, the fast increasing volume of time series data has placed an unprecedented requirement on computing resources, especially storage space [6, 79].

An effective storage management strategy that can constrain the storage space is desirable and important for time series databases. While large organizations can afford the storage to hold the ever-growing time series data, small or medium-sized entities prefer to strike a good balance between data volume and storage cost [45]. Besides, storage space is restricted in some specific deployments, e.g., real-time monitoring at far remote sites [8, 19, 78]. On the other hand, as the significance of time series data is highly correlated with the age of the data [22, 37, 89], it is desirable to have a storage management strategy that takes data ages into account [3, 7].

Significant prior work has addressed the storage-control problem by compression, which can be lossless or lossy. Lossless compression [10, 33, 57, 71, 73] preserves the complete data, but its achievable upper bound on compression ratio [93] might not be satisfactory for applications. Hence, time series databases commonly control storage consumption by directly discarding data older than a given time [43] or exceeding a storage threshold [67]. But discarding historical data causes a loss [94]. For example, historical data are crucial for long-term observations and enabling new scientific knowledge creation in the future [63]. Besides, time-based retention policy might not bound the data volume in case of unevenly spaced time series with unknown arrival intervals. Another common approach is to exploit lossy compression [15, 41, 65], which preserves partial data and trades off precision for space. But existent approaches to lossless and lossy compression are only best-effort about the final size of compressed data size [13, 24, 99].

In this paper, we take a new approach towards controlled storage space for time series stores. We consider the problem of automatically bounding the storage of a time series store by compression. To enable this, our key insight is that time series data can be compressed *losslessly* or *lossily* according to its *importance,* which is in turn related to its age, as users commonly accept information loss on less important old data [12, 14, 23, 38, 40]. We control the storage space by time-varying compression, which compresses data in a sequence of ratios defined by a time-dependent function. Inspired by time-decayed windowing of stream processing [9, 22], our design of time-varying compression takes the chunking-and-varied-segmentation approach, accepting user-defined time-dependent functions and fixed-ratio compressors.

To automatically bound time series storage by compression, three fundamental challenges exist. The first is deciding when to start the time-varying compression, i.e., the proper moment when 1) it is not too late that the storage space is exceeded during compression; and, 2) it is not too early that unnecessary compression is applied to some recent data, for preserving as much information as possible. The second challenge is computing the proper compression ratio r, given which the

†Co-first author. Work done at Tsinghua University.
‡The corresponding author (zhuyuqing@tsinghua.edu.cn).

sequence of compression ratios can be deduced using a time-dependent function. r should not be too large to prevent discarding information unnecessarily and meantime not be too small to exceed the storage bound. The third challenge is finding out how to run the time-varying compression, i.e., whether to compress data in an online stream processing manner or in a batch processing manner. The goal is to reduce computing resource consumption and improve performance.

To address these challenges, we propose TVStore, a storage-bounded time series store built upon time-varying compression. TVStore can automatically and effectively bound the time series storage even if data keep being ingested. We implement TVStore by extending the storage engine of an open-source time series database named Apache IoTDB [96]. Hence, all the database functions and operations remain supported in TVStore. We evaluate TVStore in extensive experiments based on synthetic data and real-world data. Results validate the storage-boundedness of TVStore and its time-varying pattern of compression. The compression technique employed TVStore incurs low overhead compared to its baseline. It is efficient in writes and reads, 3×(25×) and 35×(8.7×) faster than the state-of-the-art(state-of-the-practice) related works [3, 67] respectively. Under the same conditions, TVStore can respond to queries with much lower error rates in most cases than the related work.

In sum, we make the following contributions in this paper:

- We propose a time-varying compression framework TVC, which can compress data by varied ratios complied with a given time-dependent function that corresponds to the age-varying importance of time series data.
- We design a time series store TVStore that can automatically run the time-varying compression framework TVC at the *proper* time, effectively bounding the storage space to a specific threshold while preserving data according to the time-varying importance for applications. To the best of our knowledge, TVStore is the first time series store that can automatically bound its storage space by time-varying compression patterns.
- We implement TVStore based on an open-source time series database[1], introducing a three-layer data reduction scheme and exploiting a line generalization algorithm as the fixed-ratio compressor for TVC.
- We run extensive experiments using synthetic and real-world data to demonstrate the efficiency and advantage of TVStore in comparison to three related time series stores, as well as to validate its storage-boundedness and time-varying pattern of compression.

2 Background and Motivation

2.1 Why Constrain Storage

Time series databases are gaining an increasing popularity [1]. Rather than processing time series as streams and analyzing only once, mounting demands have emerged for keeping time series data for future analysis [94]. But time series data are generated at a growing speed that is outpacing the increase of computing capabilities [17, 79]. Many application scenarios cannot afford enough computing resources such as storage and network bandwidth to accommodate the processing needs for time series data. Storage-bounding compression can enable the control of storage cost.

[1] https://github.com/thulab/TVStore

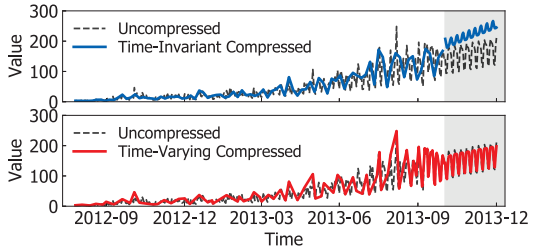


Figure 1: Time series predictions by data compressed in the *time-varying (TV)* vs. *time-invariant (TI)* manner. Predictions lie in the gray area. TV-compressed data have varied compression ratios for data at different ages, while TI-compressed data have the same compression ratio at all times. Both cases have the same overall compression ratio.

Limited storage expense. Many medium or small entities have to limit their expense on storage in their daily operations, even though the public clouds have the capacity to keep all their data [94]. As value is yet to be extracted from the huge volume of time series data, it is desirable to automatically keep as much data as possible within the storage constraint.

Sensors of a connected car can generate about 30 terabytes (TB) of data per day [62, 77]. Time series data is among the major components of the generated data. To hold all the data on such moving vehicles, large disks are installed. Since a 30TB disk can cost around $1200, a month's worth of data can fill up a 960TB disk, causing a cost of $30,000. This adds an unrealistic amount to a vehicle's price, but keeping as much data as possible can enable valuable data analytics [77, 83].

Limited computing resources. In the oil and gas industry, a typical offshore oil platform generates more than 1TB of data [19] daily. But common data transmission via satellite connection allows only a speed from 64 Kbps to 2Mbps for these offshore oil platforms. If all data are transmitted back for processing, it would take more than 12 days to move 1 day's worth of data to the processing backend [8]. Data compression is demanded for reducing data in both transmission and storage.

Scientific research applications nowadays are producing too much data to be stored or processed efficiently. For example, cosmological simulations generate petabytes of data per simulation run [34] and climate simulations generate tens of terabytes per second [29]. Such large volumes of data are imposing an unprecedented burden on storage and computation. Data reduction is necessary to enable data processing and analytics within a reasonable amount of resource and time [92].

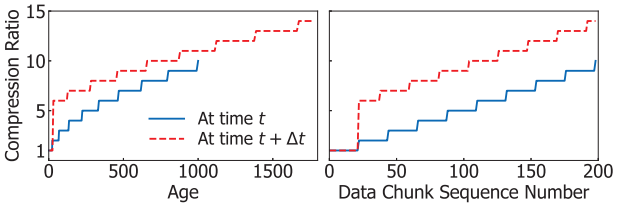


Figure 2: Compression ratio sequences generated by a function of the data age at time t and $t+\Delta t$, for reducing different data volumes to the same size, i.e., 200 data chunks. Data volumes increase with time.

2.2 Time-Varying Importance of Data

The importance of time series data changes along with time, as reflected by applications' favoring recent data over old data [5, 18, 31], or favoring some events at certain moments over others [49, 83]. Time-changing importance of data in fact commonly exists in natural and scientific phenomena [75, 86, 98]. As a result, we have seen a plethora of research on data series analysis and prediction considering the time-changing pattern [9, 22, 36, 37, 89]. Figure 1 illustrates how the importance of data varies with time in time series prediction, which is widely used in applications [39, 48, 59, 76, 88]. Recent data have dominant impact on the result of prediction, making the time-varying compression outperform the common time-invariant compression.

Time-changing importance of time series data can be exploited to form time-varying compression. For important data, we compress them *losslessly* or with a low ratio by *lossy* compression. For unimportant data, we compress them by a high compression ratio. As time series data can be identified by timestamps, we use a time-dependent function to denote the changing importance of data. Hence, the compression ratios can also be deduced from the function. Time-varying compression can suit users' requirements on data analysis well and save storage space to the most extent. As shown in Figure 2, the power-law function t^{β} with $\beta = 1$ is used for depicting time-changing importance of data and exploited to define time-varying compression ratios in both graphs. As data keep arriving, the compression at a later time reduces more data to the same volume as that at an early time, but by higher ratios as generated according to function t^{β}.

2.3 Automatic Compression and Bounding

To meet the above requirements of time series applications, we propose time-varying compression that respects the time-varying importance of time series data. Furthermore, we propose the design of a time series store that automatically bounds the total storage space to effectively control costs. To this end, compression must be initiated at proper moments to cap the overall storage space, as data increase. The compression ratios must be computed automatically, with compressions initiated at proper moments. These moments must be computed carefully such that users can keep data to its highest precision as long as possible. We must deduce the proper moments for compression initiation when 1) it is not too late that the storage space is exceeded during compression; and, 2) it is not too early that unnecessary compression is applied to some recent data or that an improperly high compression ratio is used. Besides, when lossy compression is used, users would need the overall error rates for understanding the data analysis results they could expect. The error bound computation must evolve along with time-varying compression. And, users are allowed to request the removal of data with high error rates.

Challenges: As a result, two main challenges exist in automatically bounding the time series storage by time-varying compression: 1) how time-varying compression can be executed on an ever-increasing volume of data and with error bounds computed, when the compression ratios keep changing as shown in Figure 2 (§3); and, 2) how to automatically decide the conditions for running time-varying compression such that storage space is always bounded but not too much (§4).

3 Time-Varying Compression

Given a time series, time-varying compression (TVC) compresses it to an overall compression ratio no smaller than a user-specified threshold r. TVC compresses data by the unit of chunk, which is time series data within a time interval. The compression ratios vary for different chunks according to a time-dependent function $r(t)$'s definition, where the input t is a data chunk's age as relative to the most recent timestamp of the time series. TVC enforces the compliance to different compression ratios defined by $r(t)$. The benefits of this compliance is that different $r(t)$ can be used for various use cases [22]. Provided with properly designed $r(t)$ and compressor, TVC can even achieve functionally lossless compression [54] for a long range of data.

The key challenge of time-varying compression is *how to continuously preserve the compliance with any $r(t)$ definition, when a data chunk's age increases along with the data volume*. To address this challenge, TVC initiates rounds of compression on data chunks iteratively. Figure 3 overviews time-varying compression in rounds. Later rounds of compression must execute on differently compressed data. **Two problems must be tackled**: 1) how to compute the correct sequence of compression ratios to enforce the compliance to a given $r(t)$ (§3.1, §3.2); and, 2) what properties a compressor must have to guarantee a feasible time-varying compression process, besides the fixed-ratio requirement(§3.3).

3.1 Ratio Sequencing and Data Chunking

For an overall compression ratio $\bar{r}$, TVC first finds a sequence of compression ratios $r_1, r_2, ..., r_k$ defined by the time-dependent function $r(t)$. The average of the compression ratio sequence $r_1, r_2, ..., r_k$ should approximate $\bar{r}$. The time-

dependent function $r(t)$ produces a compression ratio when given an integer t. Here, a smaller t is a time interval closer to the most recent time of a time series. Decay functions [3, 22, 75] commonly used in time series analysis can be used for $r(t)$, e.g., exponential function ($e^{\alpha t}$) and polynomial/powerlaw function (t^{β}). $r(t)$ can also be a constant function (C), but then TVC degrades to a common lossless/lossy compressor.

We assume the data for compression is kept in the unit of chunks, as time series stores commonly keep data in units like chunk [96] or block [4, 43]. TVC executes compression by the unit of chunk. To guarantee that data are compressed to the compression ratio sequence $r_1, r_2, ..., r_k$, TVC groups r_i data **chunks** into the ith **segment**. Each segment is then compressed to an output chunk. Hence, the ith chunk of the compression output has a compression ratio r_i, complying to the definition of $r(t)$.

Moreover, the compression ratio sequence must guarantee that 1) all data chunks to be compressed are actually processed; and, 2) the actual compression ratio is no smaller than $\bar{r}$ to avoid exceeding the storage bound. Hence, the sum of compression ratios must be no smaller than the number m of raw data chunks to be compressed. Besides, the average of compression ratios must be no smaller than $\bar{r}$. We then approximate $\bar{r}$ by the average of the smallest sequence of $r_1, r_2, ..., r_k$ that satisfy the following equations:

$$\Sigma_i^k r_i \geq m \quad (1)$$

$$m/k \geq \bar{r} \quad (2)$$

It is possible that no such sequence complied with $r(t)$ is found to satisfy both of the two equations, if $r_1 = r(1)$. Hence, TVC allows the compression ratio sequence r_1 to be $r(i)$, with $r_k = r(k+i-1)$. But TVC requires that the sequence $r_1, r_2, ..., r_k$ is non-decreasing, which means that $r(t)$ must be a non-decreasing function. The condition is necessary to avoid that some data chunks have a lower compression ratio in later rounds, while they are compressed in a higher ratio in a previous round. In fact, this condition naturally follows from the fact that data are aging and must be compressed with no lower ratios in later compression rounds.

3.2 Virtual Decompression and Compressions

TVC initiates a new round of compression when necessary, e.g., when conditions for constraining storage are met (discussed in Section 4). In rounds other than the first, the compression is executed on differently compressed data. It is difficult to compute the actual compression ratios based on data chunks compressed to different ratios. But the actual compression ratios are needed in enforcing the compliance to the time-dependent function $r(t)$, according to equation (2).

To compute actual compression ratios, TVC adopts the technique of *virtual decompression*. For the compression round n, TVC does not compute the compression ratios based on the compressed data from the last round. Rather, given the data chunks to be compressed in round n, TVC

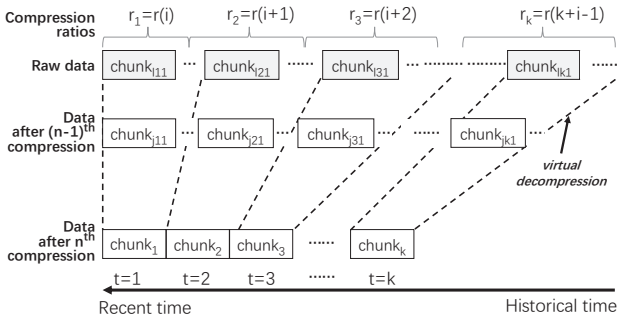


Figure 3: Time-varying compression in rounds. In each round, data of different ages are compressed to ratios that change according to a time-dependent function.

virtually decompresses them, by *mathematical mapping*, to the original raw data chunks for computing the sequence of compression ratios. Then, the conditions for ratio sequencing are considered. Thanks to the chunk-based data unit, virtual decompression can be supported by recording the number of original raw data chunks in every compression round.

Virtual decompression enables the generation of a compression ratio sequence based on the original raw data even after rounds of compression. Only by virtual decompression could the compressed data always follow the time-dependent function $r(t)$'s definition. Otherwise, data can only be compressed following the exponentially decaying pattern, as compression on compressed data leads to the multiplication of compression ratios. This would limit the applications of TVC, as $r(t)$ can only be an exponential function.

Algorithm 1 presents the main algorithm of time-varying compression for a time series. The input to the algorithm includes the number of actual chunks to be compressed and the target overall compression ratio $\bar{r}$. The algorithm consists of three parts. The first two parts guarantee the two conditions as specified by Eq. (1) and (2) while approximating $\bar{r}$ by $r_1, r_2, ..., r_k$. The third part actually compresses the data chunks by the ratio sequence.

In the first part of Algorithm 1, virtual compression is applied to the actual data chunks such that the corresponding number m of raw data chunks is obtained (line 2). According to Eq. (1), an initial sequence of compression ratios is obtained (line 4-8). As discussed above, the condition specified by Eq. (2) is not necessarily satisfied, even if Eq. (1) is met; and vice versa. Therefore, we refine the sequence to satisfy both Eq. (2) (line 9-12) and Eq. (1) (line 13-17) by approximation.

3.3 Compressor and Error Bounding

In a compression round, TVC compresses r_i data chunks by varied compression ratios into a single data chunk. To guarantee that data in an output chunk are actually compressed by the same compression ratio, we can have TVC decompress all the input chunks and then apply the same compressor by the same compression ratio. But two problems exist. First,

Algorithm 1: Time-varying compression.

```
Input: m_a: number of data chunks to be compressed;
r̄: overall compression ratio
ratioSeqQueue ← ∅;
/* To ensure the condition of Eq.(1)                    */
m ← virtualDecompress(m_a);
seqSum ← 0, j ← 0, i ← 0;
while seqSum < m do
    j+=1;
    seqSum+=r(j);
    ratioSeqQueue.enqueue(r(j))
end
/* To guarantee the condition of Eq.(2)                 */
while j-i+1 > m/r̄ do                    // j−i+1=k for Eq.(2)
    j+=1;
    seqSum+=r(j);
    ratioSeqQueue.enqueue(r(j));
    /* To approach Eq.(1)'s equality condition          */
    while seqSum ≥ m do
        i+=1;
        seqSum-=r(j);
        ratioSeqQueue.dequeue();
    end
end
/* To compress chunks by the ratio sequence             */
while ratioSeqQueue.size > 0 do
    compressOneChunk(ratioSeqQueue.dequeue());
end
```

as TVC takes iterative compression rounds, decompression before compression is highly inefficient. Second, if lossy compression is used, the decompressed data is imprecise. Rounds of decompression and compression can lead to a high deviation from the original data. A proper error bound on the lossily compressed data is desirable to users.

To avoid the above two problems, TVC requires the compressor to have the following three properties. First, compression on previously compressed data does not require decompression. Second, decompression on data compressed multiple times works the same way as on data compressed once. Third, the error bounds must be easily computed for the rounds of compression. While these properties seem to be restricted, proper approximation or representation models for time series data [26, 44, 69] are feasible choices, e.g., piecewise linear approximation (PLA) [27, 60, 87].

Among the various lossy compressors, PLA-based compressors compress a time series by approximating it using line segments. According to the related work [68], a line segment built from two line segments is the same as the line segment built from the original time series data, if line segments are properly constructed. Decompression on data at any round only needs to compute the linear function for a given time. Moreover, the mean bias error (MBE) of PLA can be computed easily even after rounds of compression. MBE is a commonly used metric for evaluating approximations [64, 81, 82]. For the ith compression round, MBE_i is the sum of round-relative error $MBE_{j-1,j}$ in previous rounds, i.e., $MBE_i = \Sigma_{j=1}^{i} MBE_{j-1,j}$, where $MBE_{j-1,j} = \frac{1}{n}\Sigma_{k=1}^{n} x_{j-1,k} - x_{j,k}$. Here, $x_{j,k}$ represents the decompressed value.

TVC accepts the specification of PLA compressors currently. TVC records the compression ratio and the error rate for every data chunk. After rounds of compression, there would be a time when some old data chunk has a high compression ratio and thus a high error rate. Keeping data at an extremely high error rate is no better than discarding it. Therefore, TVC allows users to specify a compression ratio r_{max} or an error rate e_{max}. TVC automatically discards data compressed at a ratio higher than r_{max} or at an error rate larger than e_{max}. If the compressor and the compression ratio-defining function $r(t)$ are properly chosen, TVC can achieve functionally lossless compression [54] for a long range of data, as well as supporting advanced analytical workloads [70].

4 TVStore: Automatic Storage Bounding

We propose TVStore that automatically bounds time series storage to a user-provided size using TVC as data keep being ingested. It allows users to set a recent data volume D_o that is not to be compressed. After reaching D_o, TVStore starts the compression at a *proper* time to avoid overrunning the storage bound or losing too much information. It monitors the storage consumption and initiates a process of time-varying compression when needed. Hence, three key design choices are made here:

How to compress: Shall compression be applied continuously to cold data in a batch-processing manner or hot data in a stream processing way [3]?

What ratio to compress: Will all compression ratios be feasible for storage bounding? If not, what is the proper compression ratio interval?

When to compress: When would be the proper time to start a TVC process that is neither too early to lose too much data nor too late to exceed the storage bound?

4.1 Compression on Hot Data or Cold Data?

TVStore exploits time-varying compression to bound the storage space. Compression can be applied to hot data as stream processing does [3]. It can also be applied to cold data in a batch processing mode. As TVStore targets resource limited environments, it is desirable to reduce the number of compression times and I/O accesses such that power consumption, memory utilization, and processor utilization can be reduced.

Consider the procedure of time-varying compression presented in Section 3. The compression ratios are equal to the segment sizes, in terms of data chunk numbers. As data in the largest segment is compressed the most times, it can be shown that such a segment after multiple compression rounds of TVC *has a smaller number of compression times* by cold-data compression than by hot-data compression.

In a compression round, a sequence of k segment sizes of $r(t)$, $t = i, ..., k+i-1$ is generated, as shown in Figure 3. Let

$F(k)$ be the compression times of the kth segment after this compression. For the first round of compression on cold data, k segments are compressed into k chunks, with $F_c(k) = 1$.

As for hot data, in the compression round with the same data volume, the kth segment must be compressed from multiple smaller segment of chunks, since it continuously compresses smaller chunks into larger chunks whenever possible. To obtain the kth segment, we need segments with sizes summarized to $r_k = r(k+i-1)$, i.e.,

$$r_k = \sum_{t=j}^{k+i-2} a_t r(t) \tag{3}$$

Following Eq. (3), the compression times $F_h(k)$ of the kth segment is represented as follows:

$$F_h(k) = 1 + \sum_{t=j}^{k+i-2} a_t F_h(t) \tag{4}$$

As a result, $F_c(k) < F_h(k)$, i.e., $F(k)$ has a smaller value in cold-data compression than in hot-data compression.

For the latter rounds of cold-data compression, the kth segment will also be compressed from segments with smaller sizes. That is, Eq. (4) also applies to the latter rounds of the cold-data compression case. However, when the nth compression round is triggered, the largest segment k_n will be compressed from much smaller segments, the largest of which is k_{n-1}. Segments from $k_{n-1}+1$ to k_n do not exist until the nth compression.

In comparison, the k_n segment of the hot-data compression method must be compressed from segments having the largest one equal to $k_n - 1$. It can be shown that $k_{n-1} < k_n - 1 < k_n$. Considering Eq. (4), it follows that *the cold-data compression has a smaller number of compression times than the hot-data compression*. Hence, we have the following design principle.

Principle 1. *For a given range of time series data and a sequence of compression ratios, iterative compressions over cold data can reduce the compression rounds as compared to the continuous compression method on hot data.*

The result of Principle 1 has two indications for the design of TVStore: 1) TVStore should employ the cold-data compression rather than the hot-data compression to have a smaller number of disk I/Os; and, 2) TVStore can have higher performance using the cold-data compression, as the duration of and the cost of compression are smaller (§6.2 and §6.4).

4.2 Proper Compression Ratio Interval

A proper compression ratio is required to guarantee that the storage bound will never be violated. To compute the overall compression ratio, TVStore monitors the average read throughput v_r from the disk and the average write throughput v_w to the disk, as well as the ingestion throughput v_i by applications. Next, we describe how the proper compression ratio interval can be deduced as a design choice.

Consider when compression is started for the first time. The saved storage size ΔD by compression must be larger than the ingested data volume D_i in the whole compression process. Let D_r be the data volume to be compressed and read from the disk. Let D_w be the data volume after compression and written to the disk. ΔD is equal to the difference of D_r and D_w. Hence, we have the following equations:

$$\Delta D = D_r - D_w \geq D_i \tag{5}$$

Here, D_w is decided by the original data volume D_r and the relative compression ratio r_c, i.e.,:

$$D_w = \frac{1}{r_c} D_r \tag{6}$$

We assume that compression, reads and writes run concurrently for different time series. Reads take the most time. As a result, the time to generate data volume D_i is about the same as that for reading D_r. Thus, we have:

$$\frac{D_i}{v_i} = \frac{D_r}{v_r} \Rightarrow D_i = \frac{v_i}{v_r} D_r \tag{7}$$

Combining the above three equations, we have the following:

$$r_c \geq \frac{v_r}{v_r - v_i} \tag{8}$$

Eq. (8) points to the following two rules. First, the application ingestion throughput must be lower than the disk read throughput to enable the initiation of compressions. Second, the difference $v_r - v_i$ between the disk read throughput and the application ingestion throughput is the throughput that the disk allows for filling more data besides v_i. The ratio between v_r and $v_r - v_i$ is the lower bound on the ratio for compressing the data read from the disk. That is, the following design principle exists.

Principle 2. *To avoid overrunning a storage bound, the compression ratio r_c for each round of compression must be no smaller than $\frac{v_r}{v_r - v_i}$, where v_r is the average read throughput from the disk and v_i is the ingestion throughput by applications.*

Hence, we make the design choice in TVStore regarding the compression ratio r_c for each round of the iterative compression by Principle 2.

The overall compression ratio $\bar{r}$ can be deduced based on the round compression ratio r_c. Since r_c is greater than 1, $\bar{r}$ increases as compression rounds increase. If the user-specified max compression ratio r_{max} is reached and overly-compressed data began to be deleted, then data will need to be deleted in every later round. If deletion exists from the first round, it will greatly reduce the efficacy of TVC. Hence, r_c should be at least smaller than r_{max} to avoid this case. Thus, we have a loosely feasible range for r_c, i.e., $[\frac{v_r}{v_r - v_i}, r_{max}]$.

When $r_c = \frac{v_r}{v_r - v_i}$ and $\Delta D = D_i$, compression will be initiated consecutively. This will not only reduce the system performance but also wear out the storage device. If $r_c = r_{max}$, TVStore is not storing data with as much information as possible. As a general rule, TVStore sets r to the average of the two extremes, i.e., $r_c = \frac{1}{2}(\frac{v_r}{v_r - v_i} + r_{max})$.

4.3 Compression Initiation Time

TVStore initiates compression based on the monitored data storage. Compression is initiated when the data volume reaches a threshold D_c. For a given bound D_u on the storage space, TVStore must guarantee that D_u is not exceeded at any time during any of the compression rounds. The maximum storage consumption in all compression rounds is the key to decide the threshold D_c. We first find out when this maximum storage consumption is reached.

Figure 4 illustrates two compression rounds of TVStore. Consider the first round of compression. The threshold D_c is the data volume that triggers the first compression round. $\bar{r}_1$ is the target compression ratio of this first round. The meanings of D_u, D_o, D_r, D_w, and D_i are given and illustrated in Section 4.2 and Figure 4. v_i and v_r are the ingestion throughput by applications and the average read throughput from the disk respectively. The data D_r to be compressed is the difference between D_c and D_o, while D_r and $\bar{r}_1$ decides the written data D_w after compression, i.e.,:

$$D_r = D_c - D_o \tag{9}$$

$$D_w = D_r/\bar{r}_1 \tag{10}$$

A peak of storage consumption occurs at the time right before a compression round finishes, e.g., before $t2$ in Figure 4. At that time, the original data for compression is not deleted and the compressed data is written to the disk. Let the first peak storage consumption be D_1, we have:

$$D_1 = D_o + D_i + D_w + D_r \tag{11}$$

According to Section 4.2, when compression rounds follow one another consecutively, data is kept with the most information, i.e., taking up the most storage space. Then, we can deduce from the first compression round to the kth compression round. Due to the limit of space, we leave out the straight-forward deduction process. For the kth compression round with the target compression ratio $\bar{r}_k$, the peak storage consumption D_k is:

$$D_k = D_o + (1 + \frac{1}{\bar{r}_k} + \frac{v_i}{v_r})(D_w + D_i)\prod_{x=2}^{k-1}(\frac{1}{\bar{r}_x} + \frac{v_i}{v_r}) \tag{12}$$

From Eq.(12), we can deduce two possible cases for the maximum storage consumption. If $\frac{1}{\bar{r}_x} + \frac{v_i}{v_r}$ is no greater than 1, the maximum storage consumption is D_1; otherwise, it is D_k. From Section 4.2, we can deduce that $\frac{1}{\bar{r}_x} + \frac{v_i}{v_r} \leq 1$. As a result, **the maximum storage consumption occurs at the first round of compression**.

Thereupon, TVStore decides the compression initiation time based on the maximum space consumption. That is, we only need to guarantee that $D_1 \leq D_u$. With Eq.(11), we have:

$$D_1 = D_o + D_i + D_w + D_r \leq D_u \tag{13}$$

Combining Eq.(7), Eq.(9), and Eq.(10), we deduce that:

$$D_o + (\frac{v_i}{v_r} + \frac{1}{r} + 1)(D_c - D_o) \leq D_u \tag{14}$$

Hence, with Eq.(15) deduced from Eq.(14), the following design principle stands.

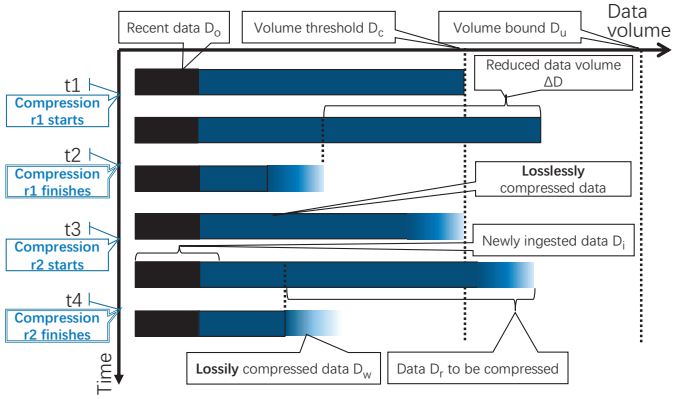


Figure 4: Storage bounding processes of TVStore: 1) at $t1$, compression round $r1$ starts when data volume reaches D_c; 2) when compression round $r1$ finishes at $t2$, storage space ΔD is saved through compression; and, 3) compression round $r2$ starts at $t3$, when data volume reaches D_c again. Data ingested during compression is D_i. Recent data D_o is not compressed lossily. The upper bound D_u of data volume is never exceeded at any time.

Principle 3. *Let D_u be the bound on the storage space and D_o be the recent data not to be compressed. Let v_r be the average read throughput from the disk and v_i the ingestion throughput by applications. Given the compression ratio $\bar{r}$ for a compression round, the threshold D_c of data volume to start a compression must satisfy the following condition.*

$$D_c \leq (D_u - D_o)/(\frac{v_i}{v_r} + \frac{1}{r} + 1) + D_o \tag{15}$$

TVStore initiates compression rounds based on results of Principle 3. Storage size and data volume monitoring is needed in the implementation of TVStore such that compression rounds can be initiated on time. Besides, the proper collection of the average metrics v_i and v_r is equally important to compute the initiation time.

5 Implementation of TVStore

We implement TVStore by extending an open-source time series database (TSDB), Apache IoTDB. The system architecture for TVStore is presented in Figure 5. TVStore replaces the TSDB storage engine by the time-varying compression/decompression storage engine. Ingested data directly go to the underlying TSDB storage. A monitoring thread runs in the background to automatically initiate time-varying compression (§3) on data in the storage when conditions are met (§4). Data are decompressed before being returned to the query engine. Hence, all the database functions originally supported by the TSDB remain supported. This architecture also allows complex analysis functions, which might be implemented in the query engine in the future, to be supported directly.

The TVStore storage engine accepts user-defined compressors for time-varying compression and time-dependent

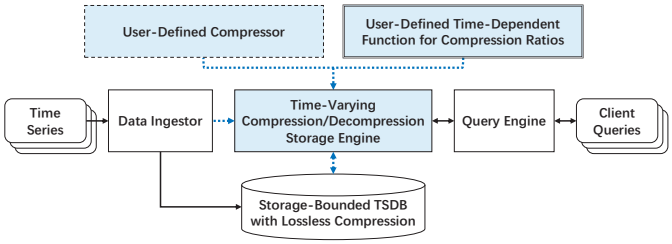


Figure 5: The architecture of TVStore: the filled components are TVStore's extensions over the time series database.

functions for compression ratios, as long as the corresponding Java interfaces are complied with. We have implemented a PLA-based compressor as the default compressor. While exponential, power-law and constant functions are all supported as the time-dependent ratio functions, TVStore uses the power-law function as the default. For time-varying compression, TVStore allows users to set the upper bound of storage space and the largest compression ratio permitted. Data volume monitoring is added to the storage engine to enable automatic storage bounding and to trigger time-varying compression rounds. Besides, we collect average metrics v_i and v_r by periodical monitoring and synopsis [21].

In the following section, we describe how the time-varying compression/decompression storage engine of TVStore integrates with the original TSDB. The choices for the compressor are also discussed as part of the TVStore implementation. The TVStore extension involves about 3000 lines of Java code.

5.1 Storage Engine Integration

In the implementation, the unit of data chunk is a data page in Apache IoTDB, each of whose data files consists of multiple data pages. When TVC compresses pages across multiple files, the involved files will be merged and restructured. Like the original IoTDB, TVStore keeps statistics and metadata on time series, as well as compression ratios of pages.

TVStore adds one layer of lossy compression to the two data-reduction layers of IoTDB. The resulting layers of data reduction are illustrated in Figure 6. Data within a data chunk are first compressed by the user-defined compressor. Then, the encoding techniques are applied to timestamps and values respectively. Encoding techniques include run-length encoding [73], Gorilla encoding [71], and delta encoding [10]. Finally, general compression as LZ4 [20] and snappy [32] is used to further reduce the overall size of stored data. The latter two layers of data reduction are lossless compression.

Although TVStore can be implemented with other TSDB, e.g., BtrDB [4] or InfluxDB [43], we have chosen Apache IoTDB [96] because its storage format enables the co-location of timestamps and values respectively within a data unit such that different encoding methods can be used to reduce data size. Besides, the structure, as well as the statistics and metadata kept within each data file, facilitates the support of TVC's iterative compression procedures.

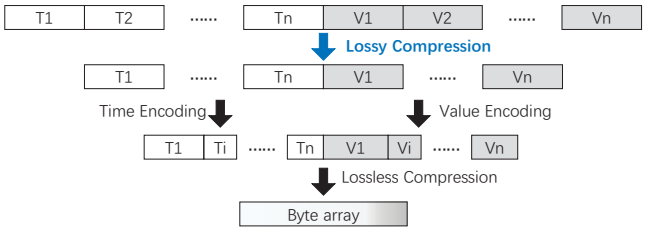


Figure 6: Layers of data reduction for one data chunk.

5.2 Fixed-Ratio Compressor/Decompressor

The time-varying compression of TVStore requires a fixed-ratio compressor to be specified. In the implementation, TVStore adopts a line generalization algorithm as the compressor and uses linear interpolation for decompression. Line generalization algorithms [95] commonly simplify one-dimensional curves by repeated eliminations of visually unimportant points, removing unnecessary details. They are inherently PLA-based compressors [87]. The number of preserved points can be set. Hence, the line generalization algorithm can be used as a fixed-ratio compressor.

Specifically, TVStore leverages the line generalization method *LTTB* (largest triangle three buckets) [85], which is a variant of the widely accepted and used Visvalingam-Whyatt (VW) algorithm [95]. As compared to other line generalization algorithms, LTTB has much lower complexity. It can compress data in almost a single pass, while preserving visually important points like its counterparts. Simplicity and data preservation are two key features that lead to our choice of LTTB, as many users would naturally prefer storing real data values [13, 99], instead of approximate values.

Decompression exploits linear interpolation. Then, the number of interpolated points between preserved points must be decided. For evenly spaced time series with constant spacing of observation times, the number of interpolated points is computed based on time units. For unevenly spaced time series, we assume important points over a data chunk have a similar distribution as points between two important points. Let ρ be the number of preserved points divided by the number of the original points in a data chunk. As LTTB mainly preserves significant points during compression, we interpolate $\frac{k(1-\rho)}{\rho(k-1)}$ points between every two preserved points.

6 Evaluation

To evaluate TVStore, we consider five questions:

1. Can TVStore bound the storage size as expected? (§6.2)
2. How does TVStore's cold-data compression compare to the hot-data compression?(§6.2)
3. Can time-varying compression compress data according to a given time-dependent function? (§6.3)
4. How does compression influence TVStore's performance? (§6.4)
5. Can TVStore answer common queries within reasonable error bounds? (§6.5)

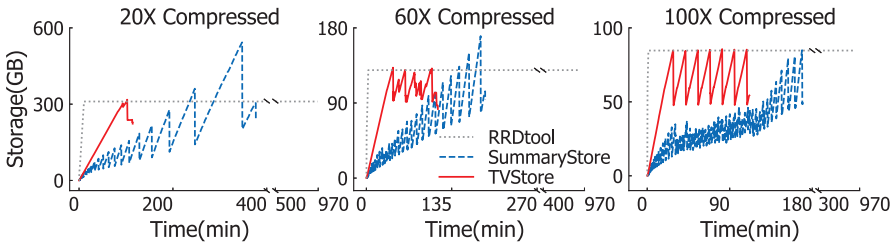


Figure 7: Storage bounding on intensive writes. All time series stores are ingested with 5TB data. TVStore and SummaryStore have the same final overall compression ratios of 20×/60×/100×. RRDtool has the same storage bounds as TVStore.

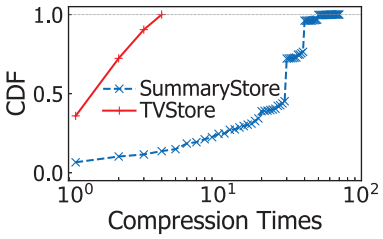


Figure 8: CDF of compression/merging times for cold data (TVStore) and hot data (SummaryStore).

6.1 Evaluation Setup

Compared Time Series Stores: We compare TVStore with three related time series stores. The first is the closest state-of-the-art work SummaryStore [3], which continuously computes predefined summaries on hot data for reducing data to a target ratio. The second is RRDtool [67], which bounds storage by deleting data when the storage quota is reached. Specially designed for monitoring [61], RRDtool has restrictions on aggregation operations, as well as timestamps and their spacing. We tried our best to circumvent the restrictions to enable comparable evaluations. The last is Apache IoTDB [96], the baseline for the TVStore implementation.

Datasets:[2] We evaluate the time series databases on both synthetic data and real-world data. We generate synthetic data with different patterns, including data with even spacing and that with uneven spacing by the Pareto distribution. We also exploit two real-world datasets, which contain regularity patterns and some random noise. One is the public REDD dataset [50]. The other is a private dataset from one of our users, denoted as the train-load dataset. REDD dataset contains several weeks of low-frequency power data for 6 different homes, and high-frequency current/voltage data for the main power supply of two of these homes. The train-load dataset consists of the train load metrics for months. The private dataset is desensitized for the evaluation purpose.

Workloads and configuration settings[2]**:** We exploit the ingestion and the query workloads included in the open-sourced SummaryStore project when testing synthetic workloads. Like SummaryStore's evaluation, our evaluation uses time series database as an integrated component in the testing client, while using python interfaces for RRDtool. We measure data storage by their final on-disk sizes. We tune the parameters of both systems so that they achieve the highest possible performance. The power-law function is used as the windowing function for SummaryStore and the ratio generation function for TVStore.

Environmental settings: We evaluate TVStore in two different settings. The first is simulating the private cloud environments of medium organizations, while the second is evaluating cases for edge computing scenarios. Hardware setup for the first setting includes 2× 12-core 2.2Hz Intel Xeon E5-2650 CPUs, and 370GB DDR4 memory. The operating system is Ubuntu 16.04.6 and the HotSpot Java runtime version 1.8.0 is used. The second type has 32GB memory and an 8-core CPU, providing a 5TB storage space for the time series database.

[2]Data and workloads – https://github.com/thulab/TVStore-benchmark

6.2 Storage Bounding and Compression Cost

We first evaluate whether TVStore can effectively bound its storage as data keep being ingested at a high speed, in comparison to SummaryStore and RRDtool. We ingest each time series store with 5TB data by 10 evenly-spaced synthetic time series. We have not chosen a larger data volume because SummaryStore cannot process more than this size under the environmental settings. We carefully tuned the evaluations on RRDtool to get the best performance, e.g., using rrdcached. Besides, as RRDtool performs 20× faster given a row larger than 4KB blocks than a row with a single column, we write each time series into one file by putting 1000 consecutive columns in one row.

In the ingestion process, we monitor storage consumption by periodically inspecting the on-disk size of database files. TVStore and SummaryStore are set to finally reduce the data by ratios of 20×/60×/100×, while RRDtool and TVStore have the same storage bounds. The changes of database storage sizes and ingestion times are plotted in Figure 7. A curve longer in the x-axis direction means a longer runtime for the corresponding test.

Bounding: TVStore and RRDtool effectively bound their data storage respectively, keeping storage below different thresholds in all cases. The folds of the storage size curves are key to the bounding for TVStore. They occur when the compression process of TVC ends. SummaryStore has also folds in its storage size curves, which occur due to the continuous summarization on hot data for data reduction.

Compression cost: TVStore's cold-data compression involves fewer disk I/Os than SummaryStore's iterative summarization process, i.e., hot-data processing (§4.1), as reflected by the fewer folds in TVStore's storage size curves than in SummaryStore's. This result of Principle 1 is further corroborated by the CDF of compression/merging times for TVStore and SummaryStore in the 60×-compressed case (Figure 8). We record the compression/merging times

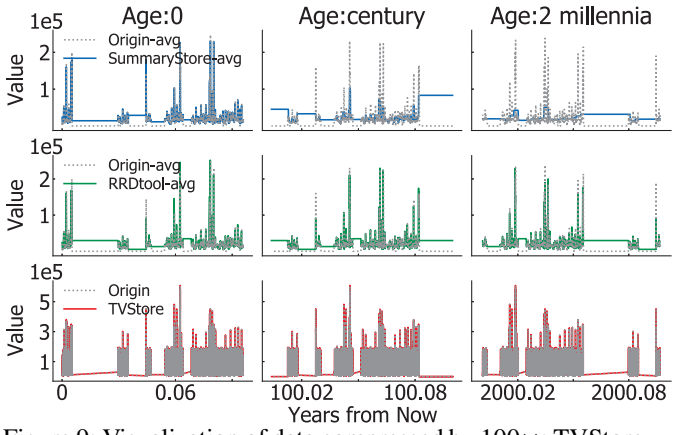


Figure 9: Visualization of data compressed by 100×: TVStore keeps more information than the other systems.

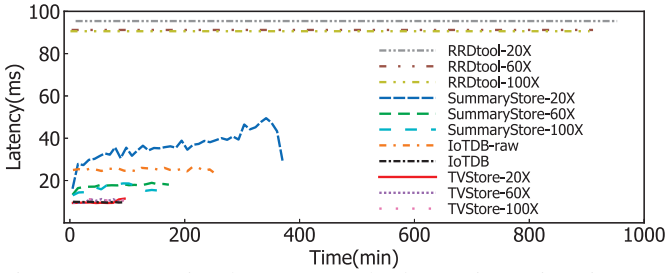


Figure 10: Ingestion latency: much *shorter* ingestion times and *lower* latencies for TVStore than for other stores.

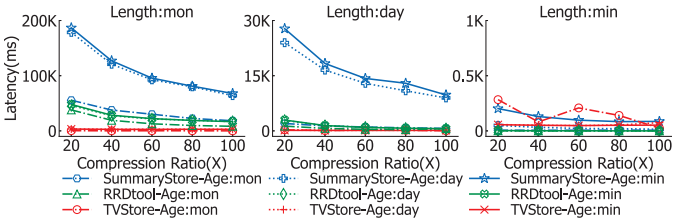


Figure 11: 95-percentile query latencies : TVStore has lower average latencies than SummaryStore and RRDtool in most cases, except for three minute-length cases.

by an extra counter in each window/chunk. The counter is incremented by one for each compression/merging. If multiple windows/chunks with different counter values are merged/compressed, the largest counter is incremented and assigned to the resulting window/chunk. While SummaryStore has windows merged about 70 times in the end, TVStore has only data chunks compressed for 4 times. TVStore has much fewer compression times thanks to its compression based on cold data, instead of hot data, and to its chunking mechanism, instead of point-level windowing.

6.3 Visualization of Compressed Data

We visualize the value patterns of compressed data to see how information is preserved by different systems under the same compression ratio. We experiment with real-world data, the low-frequency REDD data by extending the dataset to 7.5TB. The extended REDD data has a time range of multiple millennia. We visualize data at different ages, i.e., within months, one century, and two millennia. Figure 9 presents the visualization of the dataset.

TVStore and SummaryStore demonstrate time-varying patterns, while RRDtool has the time-invariant curves. SummaryStore and RRDtool keep aggregations only. Thus, only the average values can be visualized for them. In comparison, TVStore enables the visualization of the decompressed data that is compressed by 1×, 11×, and 20× respectively.

TVStore can save storage costs by enabling a high-fidelity overview of the whole range of data using only a storage space as large as 1.5 percentage of the data volume (Figure 9). RRDtool can only support similar visualization on *1.5* percentage of the data for bounding storage by simple deletion, or, have aggregated values too sparse to preserve enough information. SummaryStore has only precise data for recent times and highly different curves for historical times.

Under the same overall data reduction/compression ratio, TVStore can restore data to almost the same as the original, while RRDtool and SummaryStore cannot. The reasons are twofold. First, the implementation of TVStore has exploited three-layer data reduction, while the other two stores apply only general-purpose compressions. Hence, TVStore can have a smaller ratio for and keep more data by its lossy compression than the other two. Second, TVStore has adopted a line generalization algorithm as the compressor, which performs well at curve visualization. This result implies the importance of choosing a good compressor.

6.4 Ingestion and Query Performance

Ingestion: As shown in Figure 7, TVStore has much higher ingestion throughput than SummaryStore and RRDtool in all cases, leading to shorter curves. In the 20×-Compressed case, TVStore ingests about 3× and 25× faster than SummaryStore and RRDtool respectively, achieving a throughput of *766MB/s* or *47.8 million* time-value points per second. RRDtool has ingestion times about the same length because we have achieved its upper performance bound by writing 4K row blocks in all cases. Exploiting cold-data compression is an important reason for TVStore's advantage over SummaryStore, while insufficient compression and thus longer I/O time is a key reason for RRDtool's disadvantage.

The corresponding average ingestion latencies are demonstrated in Figure 10, with IoTDB storing raw data and IoTDB with two data reduction layers as the baselines. TVStore has a write latency around *10ms* per time-value point. Compared to the baselines, TVStore's compression process has little impact on the normal processing of writes. While RRDtool has stable and long latencies, SummaryStore has fluctuating latencies because of summarizations on hot data. We can conclude from the performance results that *higher compression ratios can effectively improve ingestion performance*.

Query latencies: We evaluate queries on data at different ages, i.e., minutes, days and months. Older data are com-

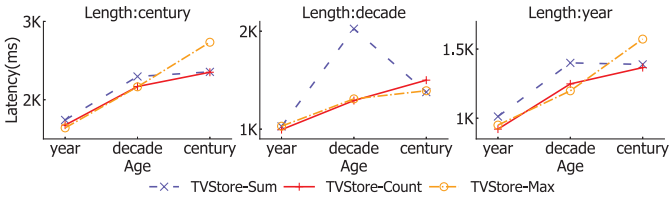


Figure 12: TVStore on the edge: all 95-percentile query latencies are within 2s on 365TB data compressed by 100×.

pressed at higher ratios than younger data. Aggregate queries on different time lengths are issued. For each combination of age and length, we issue 100 queries within random time ranges and record the 95-percentile latencies. Our TVStore implementation can answer queries 35× and 8.7× faster than SummaryStore and RRDtool respectively for the best case (5.4s vs. 194s and 47s). Figure 11 presents the results.

TVStore has lower latencies than SummaryStore and RRDtool in most cases, except for the last three cases of minute-length queries. TVStore's implementation co-locates data for a single query. As compared to SummaryStore, fewer data units need to be accessed for the same query by TVStore. SummaryStore has to read data distributed across on-disk storage for processing one query, leading to costly random disk I/Os. As for the minute-length queries, while SummaryStore only needs to retrieve some individual key-value pair, TVStore still has to access and seek a data file for results, leading to slightly higher latencies.

Query on the edge: We also evaluate the query performance of TVStore under the edge-computing condition, which is a typical application scenario for TVStore. The second experimental setting (§6.1) is exploited. Using the REDD dataset, we extend it to 365TB with 365 time series that span century time and then compress it by 100 times. We issue queries on data aging one year, one decade and one century. The resulting query latencies are presented in Figure 12. All queries can be responded within a 95-percentile cold-cache latency of at most 2.7 seconds, even for the longest length and on the oldest data.

6.5 Query Precision on Compressed Data

We evaluate whether TVStore can respond queries on the lossily compressed data within reasonable error bounds, as compared to the state-of-the-art work SummaryStore. RRDtool is not evaluated as it does not support queries approximating any time range that does not align with intervals with precomputed aggregations. Here, we mainly consider the commonly used aggregation queries, which are the basics of many complex analytical operations.

Synthetic data: We first consider the evenly-spaced synthetic data randomly generated at the 1000Hz frequency. Both TVStore and SummaryStore reduce data by 100 times. As they process count, max and min queries with almost zero errors, we present only the 95-percentile error rates

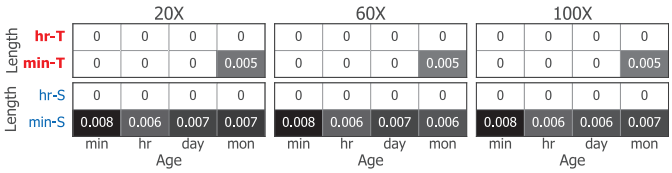


Figure 13: 95-percentile query errors on *evenly-spaced* random time series reduced by 20/60/100 times: TVStore (top two rows) vs. SummaryStore (bottom two rows). Sum-queries are issued on data at different ages and lengths

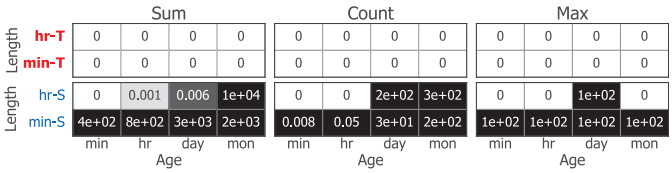


Figure 14: 95-percentile query errors on random time series with *Pareto-distributed spacings*: TVStore (top two rows) vs. SummaryStore (bottom two rows). Queries sum/count/max are issued on data at different ages and lengths.

of sum queries in Figure 13. TVStore answers queries almost precisely in all cases, except for queries with the smallest length on the oldest data. The minor error rates for such queries are mainly due to the high compression ratio and the high requirements on data details. In comparison, SummaryStore has non-zero error rates for queries with the smallest length at all ages due to the summary-based approach with only two data-reduction layers.

We also experiment on data with timestamps generated according to the Pareto distribution ($\alpha = 1.2$) and values generated uniform randomly. Data are compressed by 100 times before querying. TVStore returns query results almost precisely, with approximately zero 95-percentile error rates, while SummaryStore occasionally has extremely high error rates (e.g., $1e+04$ in Figure 14). In comparison to SummaryStore's incapability in handling unevenly-spaced data, TVStore properly compresses and decompresses the data by its point-oriented compressor based on line generalization algorithms. The results demonstrate the feasibility of error bounding by TVC, if proper compressors are employed.

Real-world data: We test queries on the real-world dataset of train load monitoring. The 6.6TB train-load dataset has 100 individual time series, each of which has about 4.5 billion points. We only evaluate TVStore on this dataset, as SummaryStore cannot support simultaneous ingestions by this number of time series streams. Data is compressed 100 times before evaluation. The results are presented in Figure 15. In most cases, TVStore answers queries with error rates below 2%. The error rates of a few max/min queries are slightly higher, at about 1, due to the high compression ratios of the corresponding old data, as well as the irregular patterns of the real-world data.

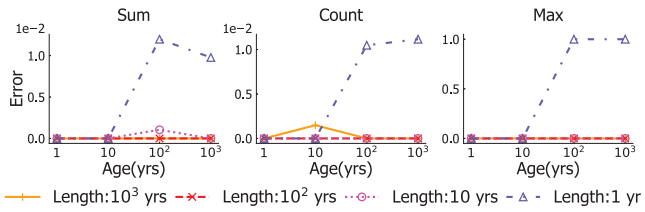


Figure 15: 95-percentile query errors on train-load data by TVStore: queries sum/count/max are issued on data at different ages and lengths.

6.6 Discussion

Our evaluations examine multiple commonly-used statistical operations as supported by the query engine of IoTDB. As high-level analysis operations are mainly implemented in the query engine, TVStore can directly support such operations when the corresponding query engine is used. In comparison, some time series store requires prior knowledge on users analysis requirements for each application [3]. Besides, as TVC can be integrated with different *fixed-ratio compressors* and *time-dependent ratio functions*, better support for learning-based analysis is possible given carefully chosen compressors and ratio functions [7, 70]. We leave the choice and the design of compressors and ratio functions for future work.

7 Related Work

Time series data compression. Lossless and lossy compression methods exist for time series data. Lossless compression can reconstruct the original data accurately. For lossless compression, specialized compressors for integer [10, 33, 55, 73] and for floating-point values [57, 71] outperform general-purpose compressors [20, 28, 32, 58, 66]. It is common practice that general-purpose compression is applied along with specialized compressors [43, 54, 96] in time series stores.

Lossy compression can achieve a much higher compression ratio than lossless compression by giving up partial information. Compression by linear models or PLA (piecewise-linear approximation) [13, 60, 87, 99] has been extensively used in practice for its simplicity. Line generalization algorithms can be considered as variants of the PLA method and used for time series compression [25, 46, 52, 74, 85, 95]. More complex models based on polynomials [30] or transformations [11, 16] are also considered as compressor alternatives in model-based time series stores [44, 56], which select from a set of models to compress time series with the least errors. Lossy compression methods commonly optimize the compression ratio towards specified error bounds, but it is difficult for users to set the bounds beforehand for real-world data. Few research work on compressors exists for optimizing error bounds towards a given compression ratio.

Our time-varying compression framework TVC is orthogonal to the above compression methods. As long as a compressor can compress data by a given ratio and satisfy the three properties (§3.3), TVC can take advantage of it to enable time-varying compression.

Time series stores. To manage the ever-increasing volume of time series data, most time series stores either have a native architecture to support a distributed deployment such as InfluxDB [43], or exploit a distributed storage for scalability, e.g., KairosDB [47] on Cassandra [53], TimescaleDB [90] on PostgreSQL database [72], Druid [100] on HDFS [35], BtrDB [4] on Ceph [97], Chronix [54] on Solr [84], and Peregreen on an object store [94]. While data distribution techniques are also applicable to TVStore, TVStore focuses on data reduction and storage control.

Besides lossless and lossy compression, time series databases commonly exploit the data retention policy to reclaim storage space by removing data exceeding a time period [42] or exceeding a storage quota, e.g., RRDtool [67], for further storage reduction. ModelarDB [44] reduces storage and query latency by time series approximation models with user-defined error bounds. Some models can also be exploited by TVC of TVStore. Adopting common stream processing methods [17, 21], SummaryStore [3] can reduce storage to a specified ratio by keeping only data summaries, if data analytical requirements are known beforehand for each application. SummaryStore works only with a limited set of summaries and cannot support data restoration.

In comparison, TVStore automatically bounds time series storage by TVC. It requires neither prior knowledge on exact retention time nor that on query workloads. TVStore enables users to respect varied data significance by integrating a chosen compressor and a time-dependent function for their applications. It can support data restoration given properly designed compressors/decompressors.

8 Conclusion

The fast increasing volume of time series data is outpacing the increase of users' affordable storage space. It is desirable to have a time series database that can automatically control the time series data storage, while preserving as much information as possible and in a manner considering data ages, which are correlated with data importance. To the best of our knowledge, TVStore is the first time series store that achieves this goal. Leveraging the proposed time-varying compression, TVStore bounds the time series database storage by iterative compressions that are initiated at rigorously chosen proper times and ratios. Extensive evaluations based on synthetic and real-world data validate the storage-boundedness of TVStore and its time-varying pattern of compression. Besides, The advantage and efficacy of TVStore are also demonstrated by its superior performance over three state-of-the-art or state-of-the-practice related works of time series store.

Acknowledgements

We sincerely appreciate the shepherding from Deniz Altınbüken. We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This work was supported by NSFC Grant (No. 62021002).

References

[1] Db-engines ranking of time series dbms. https://db-engines.com/en/ranking/time+series+dbms, 2020.

[2] Dbms popularity broken down by database model–trend of the last 24 months. https://db-engines.com/en/ranking_categories, 2021.

[3] Nitin Agrawal and Ashish Vulimiri. Low-latency analytics on colossal data streams with summarystore. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 647–664, 2017.

[4] Michael P Andersen and David E Culler. Btrdb: optimizing storage system design for time series processing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 39–52, 2016.

[5] Florian M Artinger, Nikita Kozodi, Florian Wangenheim, and Gerd Gigerenzer. Recency: prediction with smart data. In *2018 AMA Winter Academic Conference: Integrating paradigms in a world where marketing is everywhere, February 23-25, 2018, New Orleans, LA. Proceedings*, pages L–2. American Marketing Association, 2018.

[6] Hany Fathy Atlam, Robert Walters, and Gary Wills. Internet of things: state-of-the-art, challenges, applications, and open issues. *International Journal of Intelligent Computing Research (IJICR)*, 9(3):928–938, 2018.

[7] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. Macrobase: prioritizing attention in fast data. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 541–556, 2017.

[8] Brad Bechtold. Beyond the barrel: how data and analytics will become the new currency in oil and gas. https://gblogs.cisco.com/ca/2018/06/07/beyond-the-barrel-how-data-and-analytics-will-become-the-new-currency-in-oil-and-gas/, 2018.

[9] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM international conference on data mining*, pages 443–448. SIAM, 2007.

[10] Davis Blalock, Samuel Madden, and John Guttag. Sprintz: time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):1–23, 2018.

[11] Peter Bloomfield. *Fourier analysis of time series: an introduction*. John Wiley & Sons, 2004.

[12] A Bremler-Barr, E Cohen, H Kaplan, and Y Mansour. Predicting and bypassing internet end-to-end service degradations. In *Proc. 2nd ACM-SIGCOMM Internet Measurement Workshop. ACM*, volume 10, pages 637201–637248, 2002.

[13] EH Bristol. Swinging door trending: adaptive trend recording? In *ISA National Conf. Proc., 1990*, pages 749–754, 1990.

[14] Peter Burge and John Shawe-Taylor. Frameworks for fraud detection in mobile telecommunications networks. In *Proceedings of the Fourth Annual Mobile and Personal Communications Seminar. University of Limerick*. Citeseer, 1996.

[15] Shubham Chandak, Kedar Tatwawadi, Chengtao Wen, Lingyun Wang, Juan Aparicio Ojea, and Tsachy Weissman. Lfzip: lossy compression of multivariate floating-point time series data via improved prediction. In *2020 Data Compression Conference (DCC)*, pages 342–351. IEEE, 2020.

[16] Pimwadee Chaovalit, Aryya Gangopadhyay, George Karabatis, and Zhiyuan Chen. Discrete wavelet transform-based time series analysis and mining. *ACM Computing Surveys (CSUR)*, 43(2):1–37, 2011.

[17] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. Approximate query processing: no silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 511–519, 2017.

[18] Yi-Cheng Chen, Lin Hui, and Tipajin Thaipisutikul. A collaborative filtering recommendation system with dynamic time decay. *The Journal of Supercomputing*, 77(1):244–262, 2021.

[19] Cisco. New realities in oil and oas: data management and analytics. https://www.cisco.com/c/dam/en_-us/solutions/industries/energy/docs/OilGasDigital TransformationWhitePaper.pdf, 2017.

[20] Yann Collet et al. Lz4 - extremely fast compression. https://lz4.github.io/lz4/, 2011.

[21] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. Synopses for massive data: samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[22] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. Forward decay: a practical time decay model for streaming systems. In *2009 IEEE 25th international conference on data engineering*, pages 138–149. IEEE, 2009.

[23] Corinna Cortes and Daryl Pregibon. Giga-mining. In *KDD*, pages 174–178, 1998.

[24] Sheng Di and Franck Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (IPDPS)*, pages 730–739. IEEE, 2016.

[25] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, 10(2):112–122, 1973.

[26] Frank Eichinger, Pavel Efros, Stamatis Karnouskos, and Klemens Böhm. A time-series compression technique and its application to the smart grid. *The VLDB Journal*, 24(2):193–218, 2015.

[27] Hazem Elmeleegy, Ahmed K. Elmagarmid, Emmanuel Cecchet, Walid G. Aref, and Willy Zwaenepoel. Online piece-wise linear approximation of numerical streams with precision guarantees. *Proceedings of the VLDB Endowment*, 2(1):145–156, August 2009.

[28] Facebook. Zstandard home page. https://facebook.github.io/zstd/, 2017.

[29] Ian Foster, Mark Ainsworth, Bryce Allen, Julie Bessac, Franck Cappello, Jong Youl Choi, Emil Constantinescu, Philip E Davis, Sheng Di, Wendy Di, et al. Computing just what you need: online data analysis and reduction at extreme scales. In *European conference on parallel processing*, pages 3–19. Springer, 2017.

[30] Erich Fuchs, Thiemo Gruber, Jiri Nitschke, and Bernhard Sick. Online segmentation of time series based on polynomial least-squares approximations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(12):2232–2245, 2010.

[31] Diksha Garg, Priyanka Gupta, Pankaj Malhotra, Lovekesh Vig, and Gautam Shroff. Sequence and time aware neighborhood for session-based recommendations: Stan. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1069–1072, 2019.

[32] Google. Snappy home page. http://google.github.io/snappy/, 2011.

[33] Network Working Group. Rfc 3229: Delta encoding in http. https://tools.ietf.org/html/rfc3229, 2002.

[34] Salman Habib, Vitali Morozov, Nicholas Frontiere, Hal Finkel, Adrian Pope, and Katrin Heitmann. Hacc: extreme scaling and performance across diverse architectures. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2013.

[35] Hadoop. Apache hadoop distributed file system. https://hadoop.apache.org/, 2020.

[36] Peter MC Harrison, Roberta Bianco, Maria Chait, and Marcus T Pearce. Ppm-decay: a computational model of auditory prediction with memory decay. *PLoS computational biology*, 16(11):e1008304, 2020.

[37] Brian Hentschel, Peter J Haas, and Yuanyuan Tian. General temporally biased sampling schemes for online model management. *ACM Transactions on Database Systems (TODS)*, 44(4):1–45, 2019.

[38] Brian Hentschel, Peter J Haas, and Yuanyuan Tian. General temporally biased sampling schemes for online model management. *ACM Transactions on Database Systems (TODS)*, 44(4):1–45, 2019.

[39] Antony S Higginson, Mihaela Dediu, Octavian Arsene, Norman W Paton, and Suzanne M Embury. Database workload capacity planning using time series analysis and machine learning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 769–783, 2020.

[40] Jie Huang, Fengwei Zhu, Zejun Huang, Jian Wan, and Yongjian Ren. Research on real-time anomaly detection of fishing vessels in a marine edge computing environment. *Mobile Information Systems*, 2021, 2021.

[41] Nathanael Hübbe, Al Wegener, Julian Martin Kunkel, Yi Ling, and Thomas Ludwig. Evaluating lossy compression on climate data. In *International Supercomputing Conference*, pages 343–356. Springer, 2013.

[42] InfluxData. Influxdb data retention. https://towardsdatascience.com/influxdb-data-retention-f026496d708f, 2020.

[43] InfluxDB. Influxdb home page. https://www.influxdata.com/, 2020.

[44] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. Modelardb: modular model-based time series management with spark and cassandra. *Proceedings of the VLDB Endowment*, 11(11):1688–1701, 2018.

[45] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A Chien, Jihong Ma, and Aaron J Elmore. Good to the last bit: data-driven encoding with codecdb. In *Proceedings of the 2021 International Conference on Management of Data*, pages 843–856, 2021.

[46] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. M4: a visualization-oriented time series data aggregation. *Proceedings of the VLDB Endowment*, 7(10):797–808, 2014.

[47] KairosDB. Kairosdb home page. https://kairosdb.github.io/, 2020.

[48] Zahra Karevan and Johan AK Suykens. Transductive lstm for time-series prediction: an application to weather forecasting. *Neural Networks*, 125:1–9, 2020.

[49] Hakkyu Kim and Dong-Wan Choi. Recency-based sequential pattern mining in multiple event sequences. *Data Mining and Knowledge Discovery*, 35(1):127–157, 2021.

[50] J Zico Kolter and Matthew J Johnson. Redd: a public data set for energy disaggregation research. In *Workshop on data mining applications in sustainability (SIGKDD), San Diego, CA*, volume 25, pages 59–62, 2011.

[51] Antti Koski, Martti Juhola, and Merik Meriste. Syntactic recognition of ecg signals by attributed finite automata. *Pattern Recognition*, 28(12):1927–1940, 1995.

[52] Sam Kumar, Michael P Andersen, and David E. Culler. Mr. plotter: unifying data reduction techniques in storage and visualization systems. Technical Report UCB/EECS-2018-85, EECS Department, University of California, Berkeley, May 2018.

[53] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[54] Florian Lautenschlager, Michael Philippsen, Andreas Kumlehn, and Josef Adersberger. Chronix: long term storage and retrieval technology for anomaly detection in operational data. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 229–242, 2017.

[55] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[56] Chunbin Lin, Etienne Boursier, and Yannis Papakonstantinou. Plato: approximate analytics over compressed time series with tight deterministic error guarantees. *Proceedings of the VLDB Endowment*, 13(7):1105–1118, 2020.

[57] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.

[58] Jean loup Gailly and Mark Adler. Gzip home page. https://www.gzip.org/, 2003.

[59] 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, 2020.

[60] Ge Luo, Ke Yi, Siu-Wing Cheng, Zhenguo Li, Wei Fan, Cheng He, and Yadong Mu. Piecewise linear approximation of streaming time series data with max-error guarantees. In *2015 IEEE 31st International Conference on Data Engineering*, pages 173–184. IEEE, 2015.

[61] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[62] Chris Mellor. Data storage estimates for intelligent vehicles vary widely. https://blocksandfiles.com/2020/01/17/connected-car-data-storage-estimates-vary-widely/, 2020.

[63] Angela P Murillo. Data at risk initiative: examining and facilitating the scientific process in relation to endangered data. *Data Science Journal*, pages 12–048, 2014.

[64] Jan Pawel Musial, Michael M Verstraete, and Nadine Gobron. Comparing the effectiveness of recent algorithms to fill and smooth incomplete and noisy time series. *Atmospheric chemistry and physics*, 11(15):7905–7923, 2011.

[65] Rascha JM Nuijten, Theo Gerrits, Judy Shamoun-Baranes, and Bart A Nolet. Less is more: on-board lossy compression of accelerometer data increases biologging capacity. *Journal of Animal Ecology*, 89(1):237–247, 2020.

[66] Markus F.X.J. Oberhumer. Lzo home page. http://www.oberhumer.com/opensource/lzo/, 2008.

[67] Tobi Oetiker. Rrdtool: round robin database tool. http://oss.oetiker.ch/rrdtool/, 2021.

[68] Themis Palpanas, Michail Vlachos, Eamonn Keogh, and Dimitrios Gunopulos. Streaming time series summarization using user-defined amnesic functions. *IEEE Transactions on Knowledge and Data Engineering*, 20(7):992–1006, 2008.

[69] Thanasis G Papaioannou, Mehdi Riahi, and Karl Aberer. Towards online multi-model approximation of time series. In *2011 IEEE 12th International Conference on Mobile Data Management*, volume 1, pages 33–38. IEEE, 2011.

[70] John Paparrizos, Chunwei Liu, Bruno Barbarioli, Johnny Hwang, Ikraduya Edian, Aaron J Elmore, Michael J Franklin, and Sanjay Krishnan. Vergedb: a database for iot analytics on edge devices. In *CIDR*, 2021.

[71] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: a fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[72] PostgreSQL. Postgresql home page. https://www.postgresql.org/, 2020.

[73] AH Robinson and Colin Cherry. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967.

[74] Kexin Rong and Peter Bailis. Asap: prioritizing attention via time series smoothing. *Proceedings of the VLDB Endowment*, 10(11), 2017.

[75] Ariel Rosenfeld, Joseph Keshet, Claudia V Goldman, and Sarit Kraus. Online prediction of exponential decay time series with human-agent application. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, pages 595–603, 2016.

[76] Rohan Basu Roy, Tirthak Patel, Raj Kettimuthu, William Allcock, Paul Rich, Adam Scovel, and Devesh Tiwari. Operating liquid-cooled large-scale systems: long-term monitoring, reliability analysis, and efficiency measures. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 881–893. IEEE, 2021.

[77] SAS. The connected vehicle: big data, big opportunities. https://www.sas.com/content/dam/SAS/en_us/doc/whitepaper1/connected-vehicle-107832.pdf, 2021.

[78] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[79] Theo Schlossnagle, Justin Sheehy, and Chris McCubbin. Always-on time-series database: keeping up where there's no way to catch up. *Communications of the ACM*, 64(7):50–56, 2021.

[80] Omer Berat Sezer, Mehmet Ugur Gudelek, and Ahmet Murat Ozbayoglu. Financial time series forecasting with deep learning: a systematic literature review: 2005–2019. *Applied Soft Computing*, 90:106181, 2020.

[81] Zakhele Phumlani Shabalala, Mokhele Edmond Moeletsi, Mphethe Isaac Tongwane, and Sabelo Marvin Mazibuko. Evaluation of infilling methods for time series of daily temperature data: case study of limpopo province, south africa. *Climate*, 7(7):86, 2019.

[82] Syed Ahsin Ali Shah, Wajid Aziz, Majid Almaraashi, Malik Sajjad Ahmed Nadeem, Nazneen Habib, and Seong-O Shim. A hybrid model for forecasting of particulate matter concentrations based on multiscale characterization and machine learning techniques. *Mathematical Biosciences and Engineering*, 18(3):1992–2009, 2021.

[83] SIBROS. Smart data logging for connected vehicle value creation. https://www.sibros.tech/post/smart-data-blogging-for-connected-vehicle-value-creation, 2019.

[84] Apache Solr. Open source enterprise search platform. http://lucene.apache.org/solr, 2021.

[85] Sveinn Steinarsson. Downsampling time series for visual representation. Master's thesis, 2013.

[86] Joachim Stolze, Angela Nöppert, and Gerhard Müller. Gaussian, exponential, and power-law decay of time-dependent correlation functions in quantum spin chains. *Physical Review B*, 52(6):4319, 1995.

[87] Marco Storace and Oscar De Feo. Piecewise-linear approximation of nonlinear dynamical systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 51(4):830–842, 2004.

[88] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. P-store: an elastic database system with predictive provisioning. In *Proceedings of the 2018 International Conference on Management of Data*, pages 205–219, 2018.

[89] Yinyan Tan, Zhe Fan, Guilin Li, Fangshan Wang, Zhengbing Li, Shikai Liu, Qiuling Pan, Eric P Xing, and Qirong Ho. Scalable time-decaying adaptive prediction algorithm. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 617–626, 2016.

[90] TimescaleDB. Timescaledb home page. https://www.timescale.com/, 2020.

[91] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[92] Robert Underwood, Sheng Di, Jon C Calhoun, and Franck Cappello. Fraz: a generic high-fidelity fixed-ratio lossy compression framework for scientific floating-point data. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 567–577. IEEE, 2020.

[93] Sergio Verdu. Fifty years of shannon theory. *IEEE Transactions on information theory*, 44(6):2057–2078, 1998.

[94] Alexander Visheratin, Alexey Struckov, Semen Yufa, Alexey Muratov, Denis Nasonov, Nikolay Butakov, Yury Kuznetsov, and Michael May. Peregreen–modular database for efficient storage of historical time series in cloud environments. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 589–601, 2020.

[95] Maheswari Visvalingam and James D Whyatt. Line generalisation by repeated elimination of points. *The cartographic journal*, 30(1):46–51, 1993.

[96] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A McGrail, Peng Wang, et al. Apache iotdb: time-series database for internet of things. *Proceedings of the VLDB Endowment*, 13(12):2901–2904, 2020.

[97] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[98] Steven R Wilkinson, Cyrus F Bharucha, Martin C Fischer, Kirk W Madison, Patrick R Morrow, Qian Niu, Bala Sundaram, and Mark G Raizen. Experimental evidence for non-exponential decay in quantum tunnelling. *Nature*, 387(6633):575–577, 1997.

[99] George Edward Williams. Critical aperture convergence filtering and systems and methods thereof, July 2006. US Patent 7,076,402.

[100] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: a real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168, 2014.

Removing Double-Logging with Passive Data Persistence in LSM-tree based Relational Databases

Kecheng Huang[†‡], Zhaoyan Shen[†*], Zhiping Jia[†], Zili Shao[‡] and Feng Chen[§]

[†]Shandong University, [‡]The Chinese University of Hong Kong and [§]Louisiana State University

Abstract

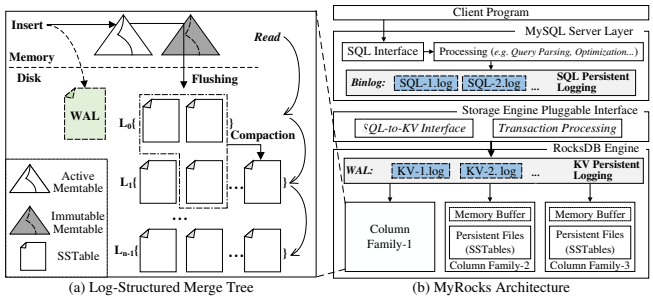
Storage engine is a crucial component in relational databases (RDBs). With the emergence of Internet services and applications, a recent technical trend is to deploy a Log-structured Merge Tree (LSM-tree) based storage engine. Although such an approach can achieve high performance and efficient storage space usage, it also brings a critical *double-logging* problem—In LSM-tree based RDBs, both the upper RDB layer and the lower storage engine layer implement redundant logging facilities, which perform synchronous and costly I/Os for data persistence. Unfortunately, such "double protection" does not provide extra benefits but only incurs heavy and unnecessary performance overhead.

In this paper, we propose a novel solution, called *Passive Data Persistence Scheme* (PASV), to address the double-logging problem in LSM-tree based RDBs. By completely removing Write-ahead Log (WAL) in the storage engine layer, we develop a set of mechanisms, including a passive memory buffer flushing policy, an epoch-based data persistence scheme, and an optimized partial data recovery process, to achieve reliable and low-cost data persistence during normal runs and also fast and efficient recovery upon system failures. We implement a fully functional, open-sourced prototype of PASV based on Facebook's MyRocks. Evaluation results show that our solution can effectively improve system performance by increasing throughput by up to 49.9% and reducing latency by up to 89.3%, and it also saves disk I/Os by up to 42.9% and reduces recovery time by up to 4.8%.

1 Introduction

Since 1970s, relational database (RDB) has been playing a central role in the heart of enterprise systems. The storage engine, as a core component in RDBs, typically adopts a B-tree based structure, which has been heavily tuned and optimized for traditional database workloads, such as online transaction processing and online analytical processing.

*Corresponding author



(a) Log-Structured Merge Tree (b) MyRocks Architecture

Figure 1: Architecture of a typical LSM-tree based RDB.

With the emergence of Internet services and applications, the classic B-tree based storage engine, after dominating database systems for decades, is facing several critical challenges. Unlike conventional database workloads, these new applications and their supporting systems often generate very write-intensive workloads [1]. Many of them use relatively simple and fixed data schema [2]. Some systems adopt expensive flash storage [3–9], and thus they are very sensitive to storage space usage and demand efficient data compression for cost saving. Correspondingly, the storage engine design must meet a set of new requirements—scalability, space efficiency, compressibility, I/O sequentiality, etc.

To address these new challenges and demands, a recent technical trend is to deploy a *Log-structured Merge Tree* (LSM-tree) based storage engine [10–18] in RDBs. A typical example is Facebook's MyRocks [7]. Different from the traditional structure of MySQL, MyRocks replaces the original B-tree based storage engine, InnoDB [19], with an LSM-tree based storage engine, RocksDB [12]. Although such an LSM-tree based storage engine significantly outperforms the B-tree based engine in terms of both performance and storage space usage, it brings a critical new issue, which can incur heavy and unnecessary performance overhead.

As illustrated in Figure 1, integrating an LSM-tree based storage engine in an RDB essentially creates a *two-layer structure*: (1) On the top RDB layer, the RDB logic handles the database related complexities, such as buffer pool management, query optimization, SQL query processing, transaction

support, data recovery, etc; (2) At the bottom storage engine layer, the storage engine processes requests from the RDB layer and is responsible for reliably and efficiently storing data in persistent storage. Such a design enables great flexibility, efficiency, and portability, allowing the two layers to be independently optimized without affecting each other. However, as a complete data store itself, the LSM-tree based storage engine, such as RocksDB [12], has many functions similar to the RDB atop it. Some of these functions are *redundant* and *unnecessary*, which can cause severe resource waste and negative performance impact. One such critical component is *log*, which is the focus of this paper.

Double-logging problem. In an RDB system, a *binlog* records all processed SQL statements. Once system crash happens, the SQL statements in the binlog are replayed for data recovery. With an LSM-tree based storage engine, each SQL statement is translated into a sequence of key-value items (KVs), which are stored in the underlying LSM-trees for persistent storage. In an LSM-tree based storage engine, a *Write-ahead Log* (WAL) is maintained to record all KV update operations. Each KV must be first written to the WAL before being inserted into the tree structure, which is also for the purpose of data recovery upon system crash. In the whole stack, any change made to the database is "protected" twice—one redundant copy is preserved in the database's binlog and another one is in storage engine's WAL. Such redundancy apparently leads to unnecessary space usage, but even worse, these log-related I/O operations are synchronous and reside in the system's critical path, resulting in significant, needless I/O overhead and severely affecting system performance.

We call the above-said issue a *double-logging problem*, which is a system situation that data is over-protected by preserving database changes multiple times more than necessary. To the best of our knowledge, this is the first time this problem is revealed in RDBs with LSM-tree based storage engines.

It is worth noting that the double-logging problem is *not* the "log-on-log" problem [20], which typically appears when running a log-structured file system on a log-structured flash FTL. In the log-on-log problem, the upper-level log is stored on another lower-level log structure. In the double-logging problem, in contrast, the two logs are independent and stored separately (the binlog is not stored in or relies on the WAL). Thus, the double-logging problem does not involve issues known in the log-on-log problem, such as data remapping, unaligned segments, and uncoordinated garbage collection, etc. Rather, it concerns more about the unnecessary redundancy in I/O operations and storage space usage.

Our solution. In order to address the double-logging problem, our key idea is to completely remove WAL from the LSM-tree based storage engine, and solely rely on binlog for data recovery. A naïve solution is to directly disable WAL (e.g., RocksDB provides a configurable option). However, the system integrity cannot be guaranteed due to the uncoordinated SQL and KV operations, which can cause incomplete or erroneous recovery. Even if we perform data recovery by replaying the binlog, we have to replay all the records in the entire binlog, one after another sequentially. This would incur an excessively time-consuming data recovery process, causing a long service outage and system downtime.

In this paper, we propose a novel solution, called *Passive Data Persistence Scheme* (PASV), to address these challenges. It includes three major components: (1) To bridge the semantic gap between the RDB layer and the storage engine layer, we create a special data structure, called *Flush Flag*, to deliver the critical RDB-layer semantics, including the critical data persistence point, each KV item's logical sequence number, etc. (2) To avoid intrusive changes to the current modular system design, we propose a *Passive Memory Buffer Flushing Policy* to pass a flush flag for each LSM-tree along with its regular flush operations in the storage engine. Without requiring to explicitly flush the memory buffers, we can avoid undesirable performance impact caused by flushes and minimize structural changes to the two layers. (3) We also develop an *Epoch-based Persistence* (EBP) policy to determine the global data persistence point, guaranteeing that we only need to perform *Partial Data Recovery* to recover the necessary data and eliminate redundant KV operations that have already been persisted before system crash.

We have implemented a fully functional, open-sourced prototype based on Facebook's MyRocks [8]. Our prototype involves minor changes (only about 500 lines of C/C++ code) and is publicly available [21]. Evaluation results based on LinkBench [1] and TPC-C [22] show that our solution can effectively increase throughput by up to 49.9% and reduce latency by up to 89.3%, and it also saves disk I/Os and recovery time by up to 42.9% and 4.8%, respectively.

The rest of the paper is organized as follows. Section 2 introduces the background. Sections 3 and 4 explain the problem and the challenges. Section 5 describes the design details. Section 6 gives the evaluation results. Section 7 discusses the related work. The final section concludes this paper.

2 Background

2.1 Log-structured Merge Tree

LSM-tree structure. LSM-tree adopts a unique append-only structure [10], which is specially tailored for handling intensive small KVs. In LSM-tree, the incoming small and random KVs are firstly buffered and sorted within a memory buffer, called *MemTable*. Once the MemTable is full, it is transformed into an immutable buffer and flushed to storage as an *SSTable*. SSTable is the basic storage unit in LSM-tree. Each SSTable stores its KVs in the order of the keys.

SSTables on storage are organized in a multi-level structure, each level of which, except the first level, maintains a sequence of SSTables with non-overlapping key ranges. Two different levels may have overlapping key ranges. A lower level typically maintains several times more SSTables (wider)

than its adjacent upper level, forming a structure like a tree. If the number of SSTables at a level exceeds size limit, selected SSTables are merged into the lower level through merge sort, which is called *Compaction*. Upon a query, a binary search is performed, level by level from top to the bottom, until finding the item or returning "Not Found". Figure 1(a) illustrates the structure of a typical LSM-tree.

In order to prevent the loss of in-memory data upon system failures, all updates that are made to the memory buffer (a.k.a. MemTable) must be first written into an on-disk log structure, called *Write-ahead Log* (WAL) [23, 24]. When the system restarts after a crash, the records in the WAL are replayed to reconstruct data that are originally in the memory buffer.

Multi-LSM-tree based structure. Modern KV storage engines often maintain more than one LSM-tree to create I/O parallelism for high-speed storage devices, such as SSDs, to achieve better performance. Let us use RocksDB [12] as an example. In RocksDB, it maintains several so-called *Column Families* (CFs). Each column family corresponds to one LSM-tree [1]. Only one WAL is maintained for all LSM-trees, which is called *Group Logging* [25] or *Group Commit* [26]. Although keeping one WAL for all LSM-trees can bring performance advantages in transaction processing, it leads to an issue, which is that the batched KVs logged in WAL may be inserted into LSM-trees at different speeds, causing an inefficient recovery process upon failures. We will explain the problem in more details later in this paper.

2.2 LSM-tree based Storage Engine

A recent technical trend is to deploy LSM-tree based storage engine in RDBs. A typical example is Facebook's MyRocks [8], which adopts RocksDB as the storage engine in MySQL database. Next, we will first explain the benefits of using LSM-tree as storage engine for RDBs, and then discuss its structure and the inherent problem.

Benefits of LSM-tree as storage engine. There are two major advantages. First, the LSM-tree structure is known for its high performance under write-intensive workloads. In new system environments, such as Internet services, which need to handle huge traffic of constantly incoming data, the performance benefit of LSM-tree is particularly appealing. The second advantage is space efficiency. Traditional storage engines typically use a B-tree based structure. Having been heavily optimized for query performance, such storage engines typically demand more storage space for complex indexes and metadata maintenance. Along with inefficient compression, the disk space usage becomes a concerning issue [4, 6, 8]. In contrast, LSM-tree is a log-structured design, which persists data in an append-only way and stores data in a sorted manner. This allows data to be organized in a more condensed format in storage. Assuming each level is 10 times larger than the upper level, theoretically, the space amplification of LSM-tree structure can be limited at a low level (approximately 1.111... times of the original data size).

Table 1: Comparison between MyRocks and MySQL.

	Total Execution Time (Seconds)	Throughput (KOPS)	Occupied Disk Space (GB)
MyRocks	10,895.8	40.9	70.7
MySQL	13,584.7	32.8	108.6

In Table 1, we show the results of a preliminary test to illustrate the performance and space efficiency of MyRocks compared to MySQL (version 5.6) with InnoDB as the storage engine. Both MyRocks and MySQL use the default configurations. We measure their performance using the same 100GB UDB-style workload [2] generated by LinkBench [1] with one loader. More details about the system setup can be found in Section 6. We can see that MyRocks saves 34.9% disk space compared to MySQL and also substantially outperforms MySQL in both total execution time (19.8%) and throughput (24.7%). This result well demonstrates the performance and storage advantages of adopting an LSM-tree based storage engine in RDBs.

RDBs on LSM-tree. More recently, many RDBs begin to adopt LSM-tree based storage engines. For example, Spanner [5] is an LSM-tree based database, which is a full-featured SQL system for distributing data at global scale and supports distributed transactions [27]. Facebook's MyRocks [7] is a MySQL-based implementation for serving the UDB scenarios [6]. MyRocks replaces the original B-tree based storage engine with RocksDB. Some other databases also construct their storage engines based on the LSM-tree structure [28–30]. Below we use MyRocks as a representative example to explain the basic structure of an LSM-tree based RDB.

Figure 1(b) shows the architecture of MyRocks, which provides an SQL interface but adopts an LSM-tree based storage engine. It is a two-layer structure and has three major components: (1) a generic MySQL server layer, (2) a pluggable SQL-to-KV translator, and (3) an LSM-tree based storage engine layer. The MySQL server layer organizes user requests in SQL transactions, logs SQL statements in the *binlog*, and issues the transactions to the SQL-to-KV translator. The translator converts the SQL statements of each transaction into a *KV batch* [31], which consists of a set of KV items. The KV batch is then sent to the LSM-tree based storage engine, which issues KV items to the corresponding column families. In RocksDB, before inserting a KV item into the LSM-tree's memory buffer (MemTable), it is first written to the WAL. Once all the KV items belonging to a transaction are inserted into the MemTables, the transaction can be safely regarded as "persistent", and a commit flag for this transaction is returned.

For data recovery, a two-phase recovery process is performed. The storage engine first retrieves the batched KV items in the WAL and rewrites them into the MemTables of the corresponding LSM-trees. Then, the MySQL server layer replays all the transactions in the binlog after the safe point

[1] In this paper, we use the two terms, column family and LSM-tree, interchangeably. Unless otherwise specified, they both refer to an LSM-tree structure in the multi-LSM-tree storage engine.

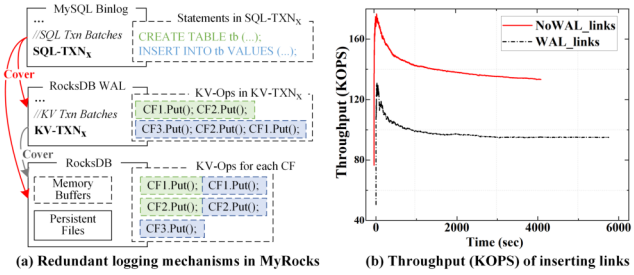


Figure 2: The double-logging problem in MyRocks.

(commit flag). The first phase guarantees that the storage engine itself is recovered to the state before the crash; The second phase guarantees the consistency of the SQL logic in the MySQL server layer.

3 The Double-Logging Problem

An LSM-tree based RDB essentially forms a two-layer structure. As shown in Figure 1(b), the RDB layer and the storage engine layer each maintains a set of complete logging mechanisms, individually, for data persistence and recovery. The two sets of mechanisms are independent of each other and co-exist in the system. Figure 2(a) illustrates the redundant functions for data persistence in this structure. We can see that despite the differences in the data persisted in binlog and WAL, the two logs are for the same purpose.

Interestingly, according to the end-to-end theory [32], such a "double protection" would not bring extra guarantees for data safety but only heavy and unnecessary performance penalties. First, all logging I/Os have to be performed twice. In the upper RDB layer, a transaction needs to be first written into the binlog in the form of SQL statements; In the lower storage engine layer, the KV items translated from the transaction need to be first written into the WAL in the form of KV operations. This is clearly a significant waste of storage I/O resources. Second, and more importantly, due to the requirement for safely committing a log record, the involved I/Os must be synchronous and performed in a serial manner to ensure correctness [33–35]. As a result, these redundant I/O operations are unfortunately in the critical path, which further amplifies the negative effect on system performance. We call it a *double-logging* problem. To the best of our knowledge, this paper is the first work unveiling this hidden, critical problem in LSM-tree based RDBs.

To illustrate the overhead, we perform a preliminary test on MyRocks. We turn off WAL in the RocksDB storage engine and keep the other configurations as default. We use LinkBench to generate a workload with around 437 million SQL requests based on Facebook's UDB distribution. Figure 2(b) shows the throughput of inserting links with 10 loaders. By simply disabling WAL, we can enhance the overall throughput (KOPS) by 44.6%. This result shows a great potential for performance improvement by solving the double-logging problem in LSM-tree based RDBs.

Another possible choice is to remove binlog in the upper RDB layer and rely on WAL for recovery. However, this approach is unreliable for two reasons. First, unlike binlog, the storage engine's WAL persists KVs individually, lacking sufficient semantic information for safe recovery. For example, a transaction written in the binlog is regarded as containing complete information at the SQL level. However, if a crash occurs in the middle, the KVs translated from this transaction could be partially persisted in the WAL. If we removed the binlog, the atomicity of such on-the-fly transactions could be compromised. Second, besides data recovery, the RDB's binlog also serves for other functions, such as instance replication and synchronization, which cannot be handled by solely keeping WAL. As indicated by the end-to-end philosophy [32], it is a more sensible choice to keep binlog rather than WAL to address the double-logging problem.

4 Critical Challenges

Our main idea is to remove WAL while still retaining data reliability upon failures. However, it is non-trivial to achieve this goal. We must address three critical challenges.

• *Unwarranted data persistence*. In an LSM-tree based RDB, the upper RDB layer translates each transaction into multiple KV items and submits to the lower storage engine layer, which receives the KV items and makes them persistent. It is assumed that the KV items are persisted once the RDB receives completion. However, if the WAL was eliminated, such an assumption could not hold anymore. In other words, the transaction commit flag in the binlog can no longer be reliably viewed as the safe point for data persistence, since the upper RDB layer cannot be certain whether the transactions prior to this point have been truly made persistent or not.

• *Partial persistence*. In a storage engine with multiple LSM-trees, an SQL transaction is translated into a batch of KV items, which are often distributed to multiple LSM-trees, a.k.a. *Column Families* (CFs) in RocksDB. Once the memory buffer (MemTable) of a CF is filled up, it is flushed to the storage in the form of an SSTable. Since the sizes and arrival rates of the KV items accommodated in different LSM-trees may vary, such memory buffer flushes can happen at distinct frequencies and are completely uncoordinated across the LSM-trees. This could lead to a situation that at a point of time when system failure happens, an SQL transaction may be partially persistent. In other words, some KV items of the transaction have already been flushed to storage but some others are not yet (still in the volatile memory buffers).

• *Lost track of LSN*. LSM-tree based RDBs use a *Log Sequence Number* (LSN) [36] for concurrency control and meeting the ACID requirements [24]. Each KV is allocated with an LSN, which is essentially a globally unique sequence number. The items of a KV batch, which corresponds to an SQL transaction, are guaranteed to receive a sequence of consecutive LSNs. With WAL, we can guarantee that each recovered KV item still carries the originally assigned LSN.

However, if the WAL was eliminated, we would lose track of these LSNs. Even replaying the binlog cannot regenerate the lost LSNs, and if we did so, the LSNs of KV items persisted on storage would become incomplete and out of order.

In the next section, we will present our design to handle these challenging issues. We develop a set of effective schemes for safely removing WAL while still fully retaining the guarantees for data persistence.

5 Design

In this paper, we present a highly efficient solution to address the double-logging problem in LSM-tree based RDBs. We desire to achieve three important goals in our design.

• *Aim #1: Effectiveness and efficiency*. Our solution should effectively and efficiently address the double-logging problem. We need to achieve not only low performance overhead during normal runs but also fast data recovery when failure happens.

• *Aim #2: Data persistence and correctness*. Removing the redundant logging should not come at the cost of weakening the promise for data persistence and correctness. Data reliability should remain identical to the existing system.

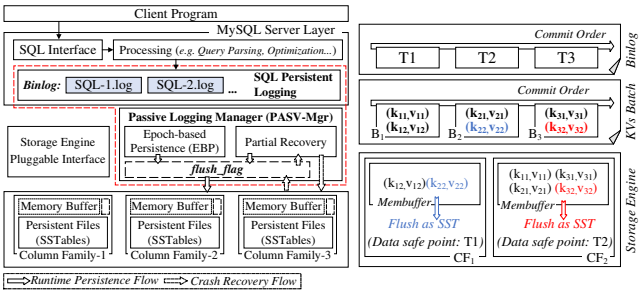
• *Aim #3: Minimal and non-intrusive changes*. A merit of the current LSM-tree based RDB design is its modularity. The RDB layer and the storage engine layer are relatively independent. We should avoid introducing complicated, intrusive changes and retain the current system's modular structure.

By following the above-said three design goals, we propose a *Passive Data Persistence Scheme* (PASV) to address the double-logging problem in LSM-tree based RDBs. We have implemented a full-featured, open-source prototype [21] based on Facebook's MyRocks. It is worth noting that the design rationale presented in this paper can also be applied to other LSM-tree based RDBs with similar double-logging problems. Although the detailed implementation may vary, our prototype provides guidance for eliminating the redundant logging for optimized performance while still achieving fast and reliable data recovery upon system failures. In the following, we will first introduce the overall design and then describe each component one by one.

5.1 Overview

Figure 3(a) illustrates the architecture of PASV for LSM-tree based RDBs. To minimize changes to the existing system design, we keep the original two-layer structure to the maximum, and only remove the Write-Ahead Log (WAL) in the lower storage engine layer and solely rely on the binlog in the RDB layer for data recovery.

We introduce three new components for the goal of eliminating redundant logging but still achieving reliable and efficient data recovery upon failures. In particular, (1) *Passive Logging Manager* (PASV-Mgr) coordinates the data logging and recovery operations between the two layers, and a special



(a) Architecture of PASV (b) Memory buffer flush operations

Figure 3: PASV architecture and data-flow.

Flush Flag is used to deliver the data recovery-related transaction information between the two layers, (2) *Epoch-based Persistence* (EBP) module determines the nearest global safe point for reliably and completely recovering the lost data upon failures, and (3) *Partial Recovery* process identifies the minimal number of KV items to be recovered for each column family, which enables fast and efficient data recovery. We introduce each component in details below.

5.2 Passive Data Persistence

In order to address the double-logging problem, our key idea is to completely remove the WAL in the lower storage engine layer and rely on the binlog in the upper RDB layer for data recovery. This is for two reasons.

First, the binlog contains a complete set of the original SQL transactions committed to the database, which makes it possible for us to recover all the database data in their original format. Second, all the KV items received at the lower layer can be reconstructed from the original transactions, even if the underlying storage engine cannot guarantee the data persistence. However, safely recovering all transaction data in their original order is non-trivial.

Challenges. The main difficulty stems from the uncoordinated flushes of the memory buffers of the underlying LSM-trees. As mentioned previously, modern LSM-tree based storage engine, such as RocksDB, maintains multiple LSM-tree structures for parallelizing I/Os and maximizing the achievable performance. Each LSM-tree, a.k.a. *Column Family* (CF), maintains an individual and independent memory buffer (MemTable) to temporarily hold incoming KV items. When the memory buffer reaches the size limit, it is flushed to disk or SSD for persistent storage. Without the WAL, upon a system failure, the KV items in the volatile memory buffer would be lost and unrecoverable. The problem is that the flush operations of memory buffers of different column families are completely independent and uncoordinated, meaning that memory buffer flushes may happen at different frequencies depending on the sizes and arrival rates of the incoming KV items, which often vary significantly across column families and dynamically change over time. As a result, the KV items translated from one SQL transaction may be persisted on storage at different time points. When a failure happens, we may have a *partially* persisted transaction and the transactions may not

be fully persisted in their original serial order as their commit flags in the binlog.

Figure 3(b) illustrates such an example, in which we have two column families, CF_1 and CF_2, and three transactions, T_1, T_2, and T_3. If CF_2 flushes before CF_1, transaction T_3 would be fully persisted on storage, while T_1 and T_2 still have partial data, (K_{12}, V_{12}) and (K_{22}, V_{22}), in the volatile memory buffer of CF_1. If a failure happens, the two transactions (T_1 and T_2) would become incomplete on storage, and we can find that the transactions are not fully persisted in their original commit order in the binlog. In order to ensure complete data recovery, we would have to replay the entire binlog from the beginning, since we cannot determine which transactions are safely and completely persisted on storage.

An active approach. A simple solution to the above-said issue is to directly insert an *Active Flush Point* after each or a number of transactions to explicitly invoke the underlying storage engine layer to flush the memory buffers of all the LSM-trees at the same time, arbitrarily creating a synchronization point. Although this "active" approach guarantees that all transactions before the active flush point are persisted safely, it has several limitations. (1) The transactions are essentially serialized, which foils the effort of creating parallelism. (2) Frequent flushes would in effect invalidate the memory buffers, causing many small and synchronous I/Os to storage. (3) Most importantly, this approach impairs the effort in the current design for modularity. It forces the RDB layer to directly control the memory buffer operations at the lower storage engine layer, which we desire to avoid.

A passive approach. To avoid intrusive changes to the existing two-layer structure, we develop a "passive" approach to handle uncoordinated flushes in a more elegant way. Here is how it works. When the storage engine flushes a memory buffer, a special KV item, called *Flush Flag*, is inserted into the memory buffer and flushed together with other KVs to the storage. The purpose is to place a "marker" in the persistent storage to indicate the progress of the latest flush operation.

A flush flag is a special-purpose KV item. Its key is a randomly chosen 128-bit magic number, which indicates that this KV item contains a flush flag rather than user data. Each column family has a unique key for its flush flag. The value contains a vector of four metrics $< CF, TSN, LSN_{first}, LSN_{last} >$. CF is the column family (LSM-tree) whose memory buffer is being flushed; TSN is the *Transaction Sequence Number* of the last transaction whose KV items are inserted in the memory buffer of the column family; LSN_{first} and LSN_{last} are the LSNs of the first KV and the last persisted KV of the transaction, respectively. To retrieve the latest flush flag, we simply query the LSM-tree using the corresponding key, which is just like retrieving any regular KV item. In this way, we can use a flush flag to keep track of the latest transaction and its KV items that are persisted in storage during a flush operation, from which we can derive the safe point for data persistence during recovery.

This approach is safe due to the *serial property* of transaction processing in LSM-tree based RDBs. In LSM-tree based RDBs, it is guaranteed that the KV items are processed in a serial manner: (1) During the transaction commit process, all transaction records are persisted to the binlog in serial; (2) The SQL transactions are parsed in the RDB layer and translated into KV batches in the storage engine layer in serial; (3) The KV items that are translated from a transaction are inserted into the LSM-trees' memory buffers in serial. Hence we can ensure that all KV items logically prior to the last KV item of the last transaction in a column family would never be persisted to storage later than it. It is worth noting that this serial property is not unique to MyRocks. Other databases also adopt the serial design. For example, Amazon Aurora [34], a novel OLTP-oriented RDB, is known to "model the database as a redo log stream" and "exploit the fact that the log advances as an ordered sequence of changes".

Based on this serial property, we can conclude that for a column family CF_i in the LSM-tree storage engine, if the retrieved flush flag contains transaction TXN_p, the KV items of transaction TXN_{p-1} and transactions prior to it in CF_i must have already been persisted. Thus, transaction TXN_{p-1} can be regarded as the *Data Safe Point* of column family CF_i. Comparatively, transaction TXN_p may be partially persisted (some KV items of the same batch may not arrive in the memory buffer yet). Hence we call transaction TXN_p the *Data Persistence Point* of column family CF_i, indicating the current position of persisting data.

In the example of Figure 3(b), if the memory buffers of CF_1 and CF_2 are flushed, their data safe points are transaction T_1 and T_2, while their data persistence points are T_2 and T_3, respectively. We also see in this example that due to the different sizes and arrival rates of the involved KV items, the column families may make unequal "progresses" in terms of persisting data for transactions (T_2 vs. T_3 in this example). We will discuss how to address this problem in the next section.

This passive approach brings several important advantages. First, we can minimize intrusive changes to the existing modular design. The RDB layer does not need to explicitly invoke memory buffer flush operations in the storage engine layer. Instead, the flush flag is naturally persisted in storage together with other KV items when the memory buffer is flushed. Second, the memory buffer flushes still follow the original logic. We do not need to prematurely flush a memory buffer that is not full yet, which maximally retains the benefits of parallelism and memory buffering. Third, the memory buffer management also remains nearly unchanged, not incurring extra performance overhead. The flush flag is very small, meaning that the spatial overhead is also minimal.

5.3 Epoch-based Persistence

The storage engine has multiple column families (CFs), each of which is an individual LSM-tree with a volatile memory

buffer. As mentioned above, different CFs may make unequal progresses in terms of persisting KV items of transactions. Thus, we need to determine the latest transaction whose KV items have been *fully* persisted on storage across all CFs.

Inspired by Epoch-based Reclamation [37–40], we propose an *Epoch-based Persistence* (EBP) to identify the global safe point for data persistence. The basic idea is to use *Local Epoch* to separately manage each CF's data safe point, and use *Global Epoch* to identify the global data safe point, which determines where we should start in binlog for recovery.

Local epoch. Flush flag maintains the last transaction and the last KV item being flushed to persistent storage, which is the *Local Data Persistence Point* as described in the previous section. The passive persistence manager, PASV-Mgr, tracks the progress of persisting data made by each column family by maintaining a tuple $< CF, TXN >$ for each column family. For a column family CF_i, we record the corresponding local data persistence point TXN_p, which is the transaction recorded in its flush flag. It indicates that all KV items of transactions prior to TXN_p in column family CF_i must have already been persisted safely on storage. Note that a "locally safe" transaction may not be safe in a "global" viewpoint, since some KV items of the transaction may not be persisted yet in another column family. A local epoch is the transactions between two consecutive persistence points in the binlog.

Global epoch. Based on the local epoch, the data persistence status for each column family (CF) is known by the system. That is, the system is aware of the local data persistence points for all CFs. Each time when a new local epoch is created, we can derive the *Global Data Persistence Point* by comparing the local data persistence points. The smallest TXN, or the earliest local data persistence point, is the global persistence point in the sequence of transactions. For a given global persistence point, all the transactions committed in the binlog prior to it must have already been safely and completely persisted on storage. If system crash happens, only the transactions starting from this global data persistence point (including itself) need to be examined and replayed.

Figure 4 illustrates an example, in which we can see that the local epochs indicate that column family CF_1 has made the most significant progress by flushing the KV items of all transactions prior to TXN_t, while CF_2 is the slowest one, which only flushes until transaction TXN_n. In terms of the global epoch, it is clear that the current global data persistence point is at transaction TXN_n, meaning that all transactions before TXN_n must have been completely persisted. Upon data recovery, we only need to replay transactions starting from transaction TXN_n and thereafter.

5.4 Partial Recovery

The epoch-based persistence policy enables us to quickly determine the latest transaction that is made completely persistent on storage. Upon a system failure, we need to recover data by replaying the transactions after that.

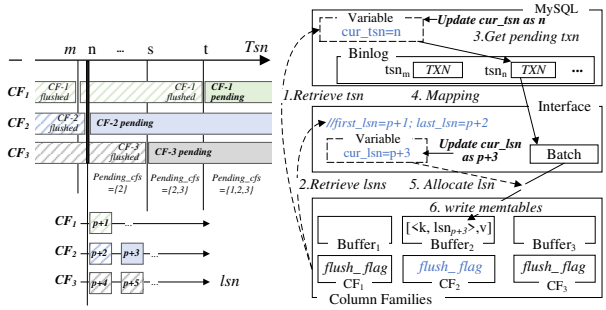


Figure 4: Partial recovery and LSN allocation.

A simple method is to replay all the KV operations in all column families. However, as illustrated in Figure 4, since column families make different progresses, such a conservative approach would be very inefficient and wasteful. For example, there is no need to reinsert the KV items of transaction TXN_n in column family CF_1, because the KV items of this transaction have already been flushed to storage. Thus, we develop *Partial Recovery* for column families whose local data persistence point differs from the global data persistence point. In other words, we selectively skip the transactions and KV items that have already been persisted.

Partial recovery consists of two steps. (1) We first retrieve the latest flush flag from each CF to determine the global data persistence point and the local data persistence points, using which we can determine the range of transactions that we need to replay for each CF. (2) Then we perform the replaying operations by scanning through the binlog and translating each SQL transaction to KV items and submit to the corresponding CFs. Due to the partial recovery policy, the KV items are only dispatched to column families that need to replay the operations. For example, for CF_1, transactions before TXN_t can be skipped.

In this way, we can avoid replaying the entire binlog from the beginning and only need to perform partial recovery for each column family as needed. In each column family, the KV items that have been persisted before the local data persistence point are skipped. This not only ensures that we can safely recover all the data but also avoid unnecessary replaying operations, which reduces the involved I/O overhead and accelerates data recovery.

5.5 Reconstructing LSNs

Another challenge in data recovery is how to reconstruct the original logical sequence number (LSN) for each involved KV item. As mentioned in Section 4, each KV item is attached with a unique LSN for version control. After removing the WAL, unfortunately, the LSN information is lost.

If the LSNs were not reproduced correctly, the data would not be recovered with correct version information and stale data could be returned for a query. LSN represents the internal order for KV items in a KV batch (corresponding to a transaction in the RDB layer). The loss of LSNs may lead to erroneous data updates. For example, assuming a KV batch contains two update operations to the same key. In the origi-

nal system, the two KV items are attached with two unique LSNs. Because LSN is a monotonically increasing number, the KV item with the larger LSN must contain the latest data. However, if we could not recover the LSNs correctly, stale data may be returned, which is unacceptable.

In the current LSM-tree based RDB design, the storage engine layer maintains a global LSN counter, which is incremented by one each time when it is attached to an inserted KV item. Thus, the KV items of a KV batch must have a sequence of consecutive LSNs. As long as we know the original LSN of the first KV item in the batch, we can recover the entire sequence of LSNs of all generated KV items due to the serial property (see Section 5.2). The flush flag contains the last transaction and the LSNs of the first KV and the last persisted KV of the transaction. When replaying a transaction, we simply re-translate the transaction and assign the LSNs one by one. As we know the range of LSNs that are originally assigned, we can derive all the related LSNs for the KV items that need to be recovered.

5.6 Put It All Together

In Figure 4, we show an illustrative example for the replaying process. As shown in the figure, when performing recovery, we first retrieve the flush flags of all the column families. Then we determine the global data persistence point from which we start replaying the transactions and also the local data persistence points from which we perform partial recovery.

In this example, the flush flag of CF_2 contains the local data persistence point TXN_n, the batch's first LSN is $p+1$ and the last persisted LSN is $p+2$. Accordingly, we update the system's current TSN for data recovery to TXN_n, which marks the transaction that we should start replaying from, and set the start LSN to $p+1$. Since the last persisted LSN is $p+2$, the LSN of the next to-be-replayed KV item's LSN should be $p+3$. Then we retrieve the transaction from the binlog and invoke the translation process to regenerate the corresponding KV batch. Note that the first KV item's LSN is $p+1$, but the first KV item that needs be recovered is $p+3$, so we can skip the first two items. During KV replaying, each time when a KV item is allocated with an LSN, we increment the LSN by one, and so on so forth.

6 Evaluation

We have implemented a fully functional prototype of PASV based on Facebook's MyRocks [8], which is a popular LSM-tree based RDB. PASV involves light changes to the existing system (only about 500 lines of C/C++ code), which are mainly in the components for binlog management, persisting data in RocksDB, and transaction-to-KV translation, etc.

In this section, we denote the stock MyRocks as "MyRocks" and our prototype as "PASV". Both MyRocks and PASV use the ROW format in the binlog to log the SQL statements in the MySQL server layer. The WAL of MyRocks is set to the default size. The other parameters for MyRocks and PASV are configured using the default setting from the stock MyRocks. Our experiments are conducted on a workstation equipped with an Intel i7-8700 3.2GHz processor, 32GB memory, and a 500GB SSD. We use Ubuntu 18.04 LTS with Linux Kernel 4.15 and Ext4 file system.

Table 2: Database schema and mapping to column families.

Table Name	Attributes	Format	Primary Key	Secondary Key
LINKTABLE	id1 id2 link_type visibility data time version	bigint(20) bigint(20) bigint(20) tinyint(3) varchar(255) bigint(20) int(11)	(link_type, id1, id2) comment **cf_linkPK;** **CF_ID=2;**	(id1, link_type, visibility, time, id2, version, data) comment **cf_linkSK;** **CF_ID=3;**
COUNTTABLE	id link_type count time version	bigint(20) bigint(20) int(10) bigint(20) bigint(20)	(id, link_type) comment **cf_countPK;** **CF_ID=4;**	
NODETABLE	id type version time data	bigint(20) int(10) bigint(20) int(10) mediumtext	(id) comment **cf_nodePK;** **CF_ID=5;**	

Our workload simulates a typical application scenario supporting Facebook's social network engine, i.e., User Data Base (UDB) [2]. In Facebook, the social graph data includes many object types, such as graph nodes and links, etc. This workload creates a model to simulate the critical pattern of UDB with three major tables, LINKTABLE, NODETABLE and COUNTTABLE, for nodes, relationships and metadata, such as counts of link type for a node, etc. The schema of these three tables are summarized in Table 2. We use LinkBench [1], an open source benchmark tool that simulates UDB-like requests (e.g., SQL INSERT, UPDATE, SELECT, etc.), to generate workloads for evaluating MyRocks and PASV.

Since the data mapping to column families may affect performance, we have extended LinkBench to map the translated KVs into different column families. As shown in Table 2, in the storage engine layer for both MyRocks and PASV, we maintain five column families (CF_ID=1–5). We manually allocate the primary key and secondary key of LINKTABLE, the primary key of COUNTTABLE, and the primary key of NODETABLE to CF2-CF5, respectively. The system column family, cf_system (CF-1), is initialized to store system metadata, such as table schema.

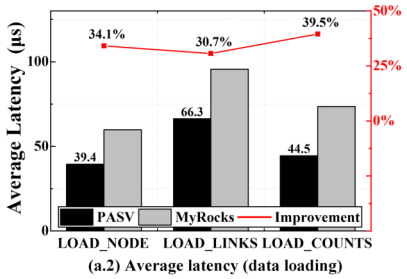
6.1 Overall Performance

We evaluate the performance of PASV and MyRocks for both data loading process and query running process. The data loading process populates the database and only involves write operations. During the query running process, the workloads are mixed with insert, update, and query operations. We use three main metrics reported by LinkBench, namely the total time, throughput, and average latency to compare the performance of PASV and MyRocks.

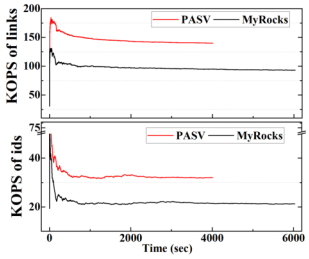
Data loading process. We use 10 loaders to initialize 100 million nodes (*ids*) and their corresponding associations (*links*) in the three tables for both PASV and MyRocks. The

	MyRocks	PASV	Improvement
Total Loading Time (Seconds)	6,027.3	4,019.9	33.3%
Throughput (KOPS)	72.6	108.8	49.9%
Total Binlog Size (GB)	54.4	54.4	
Storage Engine's IO (GB)	206.5	117.9	42.9%

(a.1) General performance (data loading)



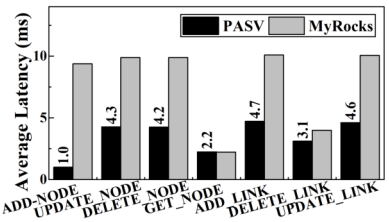
(a.2) Average latency (data loading)



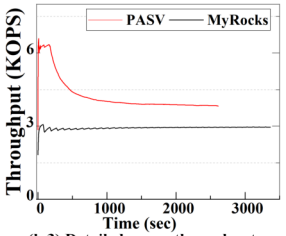
(a.3) Detailed loading throughput

	MyRocks	PASV	Improvement
Total Execution Time (Seconds)	3,371	2,614	22.5%
Throughput (KOPS)	2.97	3.82	28.6%

(b.1) General performance (query running)



(b.2) Average latency (query running)



(b.3) Detailed query throughput

Figure 5: Performance of PASV and MyRocks during (a) data loading process and (b) query running process.

data loading process is write intensive. About 432 million write operations (insert and update) with 100 GB data are issued to the database.

Figure 5(a.1) shows the total loading time and overall throughput. The total loading time of PASV is 4,019.9 seconds, which is 33.3% less than MyRocks (6,027.3 seconds). The throughput for MyRocks and PASV are 72.6 and 108.8 KOPS, respectively, and PASV outperforms MyRocks by 49.9%. Figure 5(a.3) shows the run-time throughput of different phases for PASV and MyRocks. The top sub-figure shows the run-time throughput for loading links, and the bottom sub-figure shows that of loading ids. We can see that since the memory buffers are empty initially, both PASV and MyRocks achieve a high throughput at the beginning (0–200 seconds) of the loading process. After that, the throughput has a sharp decline and finally settles at around 150 and 100 KOPS for loading links with PASV and MyRocks, respectively. For loading ids, the throughput for PASV is around 31 KOPS and 21 KOPS for MyRocks. PASV clearly outperforms MyRocks in both total loading time and throughput. For latency, as shown in Figure 5(a.2), PASV achieves much better performance than MyRocks as well. We divide the system latency into three parts, LOAD_NODE, LOAD_LINKS, and LOAD_COUNTS, as reported by LinkBench. As we can see in the figure, the average latencies of PASV are much lower than MyRocks, which are 34.1%, 30.7%, and 39.5% for the three parts, respectively.

Figure 5(a.1) also shows the I/O cost incurred in the MySQL server layer (caused by writing the *binlog*) and the LSM-tree based storage engine layer. PASV significantly reduces the amount of I/Os in the storage engine layer, compared to MyRocks. This is due to the removal of redundant writes to WAL. Specifically, the total volume of I/Os in the storage engine layer of PASV is 117.9 GB, which is 42.9% less than MyRocks. For the I/Os in the MySQL server layer, both PASV and MyRocks incur 54.4 GB I/Os during the entire loading process, which indicates that PASV does not cause extra overhead for maintaining the binlog.

In summary, by removing WAL in RocksDB, which resides in the critical path, PASV shows strong performance and storage I/O advantages over the stock MyRocks.

Query running process. During the query running process, we invoke 10 clients to perform a mixed set of operations. Each client issues 1 million read and write requests. Note that different from the data loading process, which only involves inserts and updates, the query running process consists of a variety of operations, such as deletes and retrieve queries (e.g., select from primary/secondary key, range scan, etc.) The evaluation results are as follows.

As shown in Figure 5(b.1), the total execution times of the query running process for PASV and MyRocks are 2,614 and 3,371 seconds, respectively, meaning that PASV is 22.5% faster than MyRocks. Figure 5(b.3) further shows the run-time throughput comparison between the two schemes. We can see that during the running process, MyRocks has a small performance fluctuation at the beginning (0–1,000 seconds), and it finally achieves a stable throughput around 3 KOPS. Benefiting from the removal of WAL, PASV significantly outperforms MyRocks from the beginning to 500s, and its throughput gradually settles at around 3.7 KOPS. The overall throughput of PASV is 28.6% higher than MyRocks.

We have also studied the average latencies of different types of requests. As shown in Figure 5(b.2), for the write-intensive requests, such as {ADD, UPDATE, DELETE}_NODE and those for links, PASV outperforms MyRocks across the board. PASV achieves the highest latency reduction in ADD_NODE phase, which is 89.3% lower than MyRocks. Compared to MyRocks, the average latency reduction of PASV for write-intensive requests ranges from 22% to 89.3%. For read-intensive requests (GET_NODE), PASV has slight latency overhead, which is around 0.7% higher than MyRocks. This

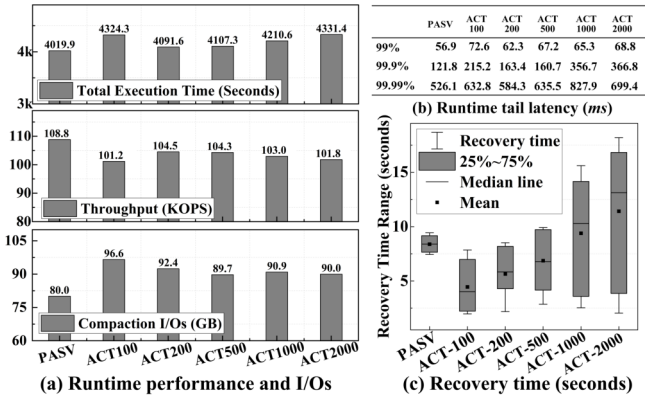


Figure 6: Performance/recovery of PASV and ACTs.

subtle performance difference is mainly due to the relatively lower read hit-ratio of the L0 SSTables in PASV. Since in PASV, the storage engine has no WAL, the I/O pressure is shifted from flush operations to L0 compaction, which makes PASV tend to have more SSTables in L0 than MyRocks. Because the key ranges among the SSTables of L0 may have overlaps, PASV needs to retrieve potentially more SSTables when searching KVs, which takes more time to complete.

6.2 Comparison with Active Flush

PASV adopts a passive approach to flush memory buffers when they reach the size limit (64 MB by default). Active flush, in contrast, synchronizes data persistence by enforcing all memory buffers to periodically flush at the same time, which would incur heavy overhead and weaken the effect of memory buffers. In this section, we compare the two methods.

We implement the active flush method (ACT) based on the `flushAll()` interface in RocksDB, which flushes all memory buffers of the LSM-trees compulsively. We record the transaction commit time during the running process. Once the number of committed transactions reaches a predefined threshold, `flushAll()` is invoked. Since `flushAll()` is a synchronized operation, lock is required during this process and all the commit threads are blocked.

We evaluate ACT by configuring different thresholds and compare their performance with PASV. We have implemented five settings for ACTs. ACT-{100, 200, 500, 1000, 2000} refer to the configurations that invoke the active buffer flush process every corresponding number of transactions, respectively. In order to show the performance during normal runs and data recovery upon failures, a write-intensive workload is created by using 10 loaders issuing 100 million node insertions and corresponding link updates.

Performance. Figure 6(a) shows the overall execution time and throughput of PASV and ACTs (ACT-100 to ACT-2000). We can see that PASV outperforms all the others, taking only 4,019.9 seconds of execution time and achieving a throughput of 108.8 KOPS. Unlike ACTs, which enforce all the LSM-trees' memory buffers to flush at a synchronized time point, PASV does not need to arbitrarily lock and flush the memory buffers periodically. This not only helps retain the efficacy of memory buffers (no premature buffer flushes) but also avoids interfering the foreground requests with unnecessary, costly I/Os. Figure 6(b) shows the effect on the observed tail latencies. The $99th$ to $99.99th$ percentile tail latencies of PASV are much lower than the others, meaning that PASV introduces less interference to foreground requests.

Compaction. Another side effect of active flushes is the more frequently happening compaction operations in the underlying LSM-trees. Figure 6(a) compares the amount of compaction I/Os generated by PASV and ACTs. PASV is much more efficient: The total amount of compaction I/Os is around 80 GB, which is 10.8%–17.2% lower than the ACTs. With frequent flushes, the memory buffers, even not being completely filled up, have to be persisted to storage, which pushes small SSTables into the underlying LSM-trees. As a result, the compaction operations tend to be triggered more often, which in turn amplifies the I/Os.

Data recovery. We have also studied the data recovery performance. We randomly select ten insertion points in the selected time window (1,000–2,000 seconds) to artificially create a simulated "failure", which triggers the recovery process. We collect the time of completing recovery after each failure and show the aggregated results for each configuration in Figure 6(c). As we increase the interval of flushing memory buffers from 100 transactions (ACT-100) to 2,000 transactions (ACT-2000), the recovery time generally increases. This is because the less frequently active flush happens, the more data needs to be restored. PASV achieves a recovery time in the mid-range. Compared to ACTs, PASV shows a much smaller variance in the data recovery time, since our epoch-based approach and partial recovery can identify the last data persistence point and minimize the amount of data for recovery. In contrast, ACTs have to recover data completely since last flush, and the time taken to complete recovery depends on when the failure happens between two consecutive flushes and thus varies significantly.

6.3 Evaluation with TPC-C

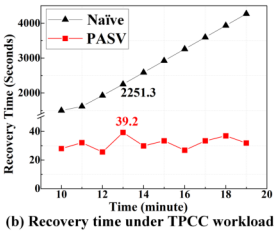
In this section, We perform experimental evaluations using the TPC-C [22] benchmark to compare the performance and potential recovery time of the stock MyRocks, the naïve approach, which simply disables WAL, and PASV.

For better understanding the recovery efficiency of PASV under a more skewed workload, we configure the TPC-C benchmark with 100 warehouses and 10 tables, which are allocated to 10 column families with uneven flush speeds. The data size is around 90 GB in the InnoDB-based MySQL.

Figure 7(a) shows that PASV outperforms MyRocks by 26.4%, 35.9%, and 23.7% in total execution time, throughput (KQPS), and storage I/O amount, respectively. The naïve approach simply disables WAL with no other operations, which represents the possibly achievable performance during normal runs. PASV achieves nearly identical performance to the

	PASV	Naïve	MyRocks
Total Execution Time (Seconds)	2,658.4	2,651	3,612
Throughput (KQPS)	35.6	35.7	26.2
Storage Engine's IO (GB)	204.2	204.2	267.6

(a) General performance under TPCC workload



(b) Recovery time under TPCC workload

Figure 7: Performance/recovery under TPC-C workload.

naïve approach. In fact, PASV only adds a very small KV item (several bytes) during memory buffer flush. The incurred overhead is minimal.

To compare recovery time, we first perform a 10-minute data loading phase for both PASV and the naïve approach, and then simulate a "failure" by artificially turning off the system at different time points (10-19 minutes after the loading phase is completed). As shown in Figure 7(b), although the column families receive KVs at skewed speeds (10x difference), the recovery time of PASV remains low (25-40 seconds), which is far less than that of the naïve approach (at least 1,497 seconds). As the test phase runs longer, the recovery time of the naïve approach increases almost linearly. Compared to PASV, the naïve approach suffers from a time-consuming recovery process, since it has to replay all the transactions in the binlog from the beginning, and the larger the binlog is, the longer it takes for recovery.

6.4 Data Recovery

In this section, we first perform the correctness analysis of the data recovery process in PASV and then we study and compare its recovery performance with MyRocks.

Recovery correctness analysis. By eliminating a redundant logging structure, PASV provides significant performance improvement and still guarantees data persistence, just like the original LSM-tree based RDB. To achieve this goal, upon a system failure, we must ensure that (1) all volatile data stored in the memory buffers before the crash should be recovered completely, and (2) the KVs managed by the storage engine should be consistent with the RDB's view.

The original design provides the above-said data persistence guarantees through a two-phase recovery using both binlog and WAL. In PASV, we must achieve the same goal with binlog only. When a failure happens, PASV first retrieves the local data persistence point (DPP) of each LSM-tree and determines the global DPP from the storage engine layer. Then, it scans all pending transactions from the binlog to determine the KVs that should be replayed. We only need to replay the KVs that have been committed at the RDB layer but not yet been persisted at the storage engine layer.

To analyze the recovery correctness, we divide the data persistence process into three phases based on two key operations `flush(Binlog)` and `flush(MemTable_x)` in Figure 8. The time period $[T_a, T_{a+5}]$ includes all the possible crash points that may happen during the whole process for an LSM-tree.

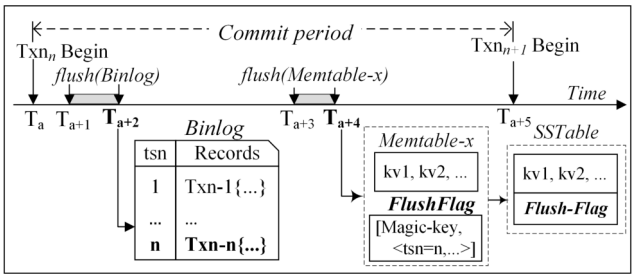


Figure 8: An illustration of transaction commit process.

• *Period #1: Uncommitted to binlog.* During time period $[T_a, T_{a+2})$, the incoming transaction TXN_n is parsed into SQL requests and then being persisted to the binlog. Before T_{a+2}, TXN_n is neither persisted to *binlog* nor the storage engine. If system crash happened during this period, PASV would retrieve the latest flush flag of the LSM-tree from the system. In this situation, the DPP must be a transaction whose sequence number is less than n (TXN_{n-1} or earlier). Thus, during recovery, the partial recovery process would compare current DPP with the global one and only recover the data after the local DPP of this LSM-tree. Since TXN_n is not committed to binlog, it will not be recovered, which is consistent with the status of the RDB layer before the system crash.

• *Period #2: Committed to binlog but not flushed.* During time period $[T_{a+2}, T_{a+4})$, the transaction TXN_n is persisted to the binlog, meaning that all KV items from TXN_n need to be recovered if crash happens. Suppose at time T_{a+3}, the memory buffer of this LSM-tree reaches the size limit, the flush flag (FF_p) containing the current committed transaction TXN_n, the corresponding LSNs and a magic key, needs to be also flushed to disk as the last KV item in the buffer. At time T_{a+4}, the flush operation ends. If a system crash happened during T_{a+2} to T_{a+4}, the flush flag FF_p could be not completely persisted, meaning that we must retrieve the last flush flag FF_{p-1}, which has already been written to the storage in the last round of memory buffer flush. Thus, the last safely persisted transaction should be prior to transaction TXN_n, which is TXN_{n-1} or earlier. Partial recovery can recover starting from there. Since TXN_n is already committed to binlog, all the KVs translated from it will be completely replayed, which is also consistent with the status before system crash.

• *Period #3: Committed and flushed.* After T_{a+4}, the FF_p, which records its current DPP TXN_n and the related LSNs, is persisted to the disk, meaning that for this LSM-tree, the current data persistence status has been persisted to disk. When a failure happens during this phase, PASV will retrieve the latest persisted flush flag, which is FF_p containing TXN_n and the corresponding LSNs, and replay uncompleted KV items from TXN_n after the latest DPP. In this case, the DPP is TXN_n. Partial recovery will only recover KV items of TXN_n that appear after T_{a+4} for this LSM-tree. Hence, it is also consistent with the status before system crash.

In general, PASV maintains at least one local DPP for each LSM-tree indicating the restart point. According to the latest

Table 3: Recovery performance of PASV and MyRocks.

	PASV		MyRocks	
	Average	*Range*	*Average*	*Range*
Recovery Time (Seconds)	7.9	0.11	8.3	0.18
Logging Volumes (GB)	1.3	0.26	3.7	0.48
Recovery I/Os (GB)	0.9	0.04	1.9	0.09

flush flag of each LSM-tree, all possible recovery situations can be solved based on the time periods of the transaction commit when crash happens.

Recovery performance analysis. To evaluate the recovery performance, we create a scenario that both PASV and MyRocks have 0.8 GB data stored in memory buffers. Table 3 shows the evaluation results of repeating the same test for 5 times. For the recovery time, PASV recovers the buffered data within 7.9 seconds on average, which is 4.8% less than MyRocks. Replaying SQL statements incurs more complexities and time cost than replaying KVs directly. When replaying SQL statements, more translation and transaction control are involved, which diminishes the performance gains from less disk I/Os and explains the relatively small improvement.

For space usage, PASV frees disk space needed for WAL, and it only needs to maintain the binlog for crash recovery. The binlog accounts for about 1.3 GB, which allows us to recover all in-memory data in storage engine's buffers and all pending transactions maintained in transaction buffer. In contrast, MyRocks needs to maintain about 3.7 GB data for both binlog and WAL, which is 2.8 times of PASV.

Furthermore, benefiting from the efficient partial recovery, PASV also incurs less disk I/Os (0.9 GB on average) for data recovery, compared to MyRocks (1.9 GB). MyRocks with multiple LSM-trees has to retrieve all the KV items in the WAL to recover each LSM-tree, which leads to severe I/O amplification. Since MyRocks is unaware of the relationship between KV items in WAL and each LSM-tree, for each LSM-tree, MyRocks needs to check all KV items in the WAL and replay the KVs that are related to the LSM-tree. In contrast, benefiting from the epoch-based data persistent policy and partial recovery, PASV can skip the unrelated KV items. Hence, for replaying 0.8 GB data, PASV only involves 0.9 GB disk I/Os, which shows its high efficiency.

7 Related Work

In recent years, storage systems have become increasingly more diverse to meet the new requirements of various emerging applications. A lot of efforts have been particularly made on optimizing data storage performance and reliability.

LSM-tree based RDBs. To satisfy the demand for performance and space efficiency, some RDBs (e.g., distributed RDBs [5], HTAP RDBs [28]) have begun to adopt LSM-tree-based storage engines [5, 27–30, 41]. Specifically, MyRocks [8], Spanner [5, 27], CockroachDB [30] as well as TiDB [28], are relational database systems that support MySQL-style protocol and provide full-featured transactional guarantees. They all adopt LSM-tree-based KV store [12, 42] as storage engines. Our work particularly focuses on solving the double-logging problem in LSM-tree based RDBs.

Journaling of journal. The journaling-of-journal (JoJ) problem is widely existing in modern storage systems, which often results in write amplification and time-costing synchronous I/Os when persisting data to file systems and databases [43–47]. A similar problem is the log-on-log problem [20], which appears when running a log-structured file system on a log-structured flash FTL. These problems are different from the double-logging problem addressed in this paper. For double-logging, the two logs are redundant and stored separately, thus one unnecessary log can be eliminated safely. In contrast, the log-on-log problem happens in a distinct environment where each of the two layers of logs is a must-have needed for different semantics and functionalities, meaning that we cannot easily remove one of them in the way as how we handle the double-logging problem.

NVM-assisted logging. Prior works also propose to adopt byte-addressable Non-volatile Memory (NVM) devices to optimize the logging performance [9, 25, 48–56]. For example, NoveLSM [54] proposes to replace the LSM-tree's memory component with persistent NVM devices. It exploits I/O parallelism by searching multiple levels concurrently to reduce lookup latency. MatrixKV [55] is another hybrid design that combines NVM with DRAM for better performance, in which the WAL is implemented in NVM to prevent data loss from system failures. These prior works cannot handle the double-logging problem. They either conservatively keep redundant logging mechanisms or simply attempt to accelerate logging using a faster device. Our work takes a different strategy and aims to fundamentally address the problem by completely removing the redundant WAL in the storage engine layer.

8 Conclusion

LSM-tree based storage engine is becoming increasingly popular in modern relational databases. A unique and critical issue is the double-logging problem, which incurs high and unnecessary overhead. In this paper, we have systematically studied this challenging problem and proposed a set of mechanisms to optimize the system design for achieving high performance and reliability. Experimental results show that our solution can effectively improve the system performance and accelerate data recovery at low cost.

Acknowledgments

We thank our shepherd, Amy Tai, and the anonymous reviewers for their constructive feedback and insightful comments. This work is supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 15224918), Direct Grant for Research, The Chinese University of Hong Kong (Project No. 4055151), the National Science Foundation for Young Scientists of China (Grant No.61902218), and the National Natural Science Foundation of China (Grant No.92064008).

References

[1] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A Database Benchmark based on the Facebook Social Graph. In *ACM International Conference on Management of Data (SIGMOD)*, 2013.

[2] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[3] MySQL Database Service. https://www.mysql.com.

[4] Chin-Hsien Wu, Tei-Wei Kuo, and Li-Ping Chang. An Efficient B-tree Layer Implementation for Flash-memory Storage Systems. *ACM Transactions on Embedded Computing Systems*, 6(3):19, 2007.

[5] David F. Bacon, Nathan Bales, Nicolas Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a SQL System. In *ACM International Conference on Management of Data (SIGMOD)*, 2017.

[6] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing Space Amplification in RocksDB. In *International Conference on Innovative Data Systems Research (CIDR)*, 2017.

[7] 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.

[8] Yoshinori Matsunobu. InnoDB to MyRocks Migration in Main MySQL Database at Facebook. Technical report, USENIX Association, 2017.

[9] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *ACM International Conference on Management of Data (SIGMOD)*, 2020.

[10] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[11] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[12] RocksDB: A Persistent Key-Value Store for Fast Storage Environments. https://github.com/facebook/rocksdb.

[13] 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 *USENIX Annual Technical Conference (ATC)*, 2017.

[14] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[15] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. Xengine: A Fast and Scalable XACML Policy Evaluation Engine. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2008.

[16] MongoDB: A General Purpose, Document-based, Distributed Database. https://www.mongodb.com.

[17] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal Navigable Key-Value Store. In *ACM Conference on Management of Data (SIGMOD)*, 2017.

[18] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[19] Introduction to InnoDB. https://dev.mysql.com/doc/refman/8.0/en/innodb-introduction.html.

[20] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack Your Log On My Log. In *Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.

[21] PASV-based MyRocks. https://github.com/ericaloha/MyRocks-PASV.

[22] TPC-C Benchmark for MySQL. https://github.com/Percona-Lab/sysbench-tpcc.

[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 *ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[24] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[25] 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 *USENIX Conference on File and Storage Technologies (FAST)*, 2021.

[26] Group Commit for the Binary Log. https://mariadb.com/kb/en/group-commit-for-the-binary-log.

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

[28] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-based HTAP Database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[29] YugabyteDB. https://github.com/yugabyte/yugabyte-db.

[30] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *ACM International Conference on Management of Data (SIGMOD)*, 2020.

[31] KVs Batch in MyRocks. https://github.com/facebook/rocksdb/wiki/Basic-Operations.

[32] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[33] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[34] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *ACM International Conference on Management of Data (SIGMOD)*, 2017.

[35] Theo Haerder and Andreas Reuter. Principles of Transaction-oriented Database Recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

[36] Tianzheng Wang and Ryan Johnson. Scalable Logging through Emerging Non-volatile Memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, 2014.

[37] Keir Fraser. *Practical Lock-freedom*. PhD thesis, University of Cambridge, UK, 2004.

[38] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based Memory Reclamation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[39] Aleksandar Prokopec. Cache-tries: Concurrent Lock-free Hash Tries with Constant-time Operations. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[40] Jeehoon Kang and Jaehwang Jung. A Marriage of Pointer- and Epoch-based Reclamation. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[41] Andrew Pavlo and Matthew Aslett. What's Really New with NewSQL? *SIGMOD Rec.*, 45(2):45–55, 2016.

[42] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[43] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O Stack Optimization for Smartphones. In *USENIX Annual Technical Conference (ATC)*, 2013.

[44] Kai Shen, Stan Park, and Meng Zhu. Journaling of Journal is (Almost) Free. In *USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[45] Daejun Park and Dongkun Shin. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *USENIX Annual Technical Conference (ATC)*, 2017.

[46] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In *USENIX Annual Technical Conference (ATC)*, 2015.

[47] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[48] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. Better I/O through Byte-addressable, Persistent Memory. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[49] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. High Performance Database Logging using Storage Class Memory. In *IEEE International Conference on Data Engineering (ICDE)*, 2011.

[50] Shen Gao, Jianliang Xu, Bingsheng He, Byron Choi, and Haibo Hu. PCMLogging: Reducing Transaction Logging Overhead with PCM. In *ACM Conference on Information and Knowledge Management (CIKM)*, 2011.

[51] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[52] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. *Proceedings of the VLDB Endowment*, 8(4):389–400, 2014.

[53] Tianzheng Wang and Ryan Johnson. Scalable Logging through Emerging Non-Volatile Memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, 2014.

[54] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *USENIX Annual Technical Conference (ATC)*, 2018.

[55] 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 *USENIX Annual Technical Conference (ATC)*, 2020.

[56] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

Improving the Reliability of Next Generation SSDs using WOM-v Codes

Shehbaz Jaffer[1,2], Kaveh Mahdaviani[1], and Bianca Schroeder[1]

[1]*University of Toronto*, [2]*Google*

Abstract

High density Solid State Drives, such as QLC drives, offer increased storage capacity, but a magnitude lower Program and Erase (P/E) cycles, limiting their endurance and hence usability. We present the design and implementation of non-binary, Voltage-Based Write-Once-Memory (WOM-v) Codes to improve the lifetime of QLC drives. First, we develop a FEMU based simulator test-bed to evaluate the gains of WOM-v codes on real world workloads. Second, we propose and implement two optimizations, an efficient garbage collection mechanism and an encoding optimization to drastically improve WOM-v code endurance without compromising performance. A careful evaluation, including microbenchmarks and trace-driven evaluation, demonstrates that WOM-v codes can reduce Erase cycles for QLC drives by 4.4x-11.1x for real world workloads with minimal performance overheads resulting in improved QLC SSD lifetime.

1 Introduction

Flash-based Solid State Drives (SSDs) offer a faster alternative to Hard Disk drives (HDDs), but have a major limitation: unlike HDDs, where previously written data is over-writable, a flash cell needs to be erased before it can be programmed, and each erase operation causes wear-out that reduces a cell's lifetime. Older generations of flash were based on single-level cells (SLC), which store only a single-bit in a cell and can typically tolerate several thousand program and erase cycles before wearing out. However, to keep up with the increasing demand for storage capacity, more bits need to be stored in a cell. Such SSDs are called multi-bit cell SSDs. Recent work [26] shows that with each additional bit stored in one SSD cell, the number of erase cycles that the SSD can endure reduces by an order of magnitude. Figure 1 illustrates the problem based on recent projections [5]. Flash based on Multi-level Cells (MLC) and Triple Level Cells (TLC), which are common nowadays, can tolerate a significantly smaller number of P/E cycles. Recently, QLC drives have started being deployed in datacenters. Even more worrisome is a look into the future with PLC drives, which might see P/E cycle limits drop to tens or a few hundred. To make high-density SSD drives usable beyond archival applications, it is paramount to reduce the number of times the storage media is erased.

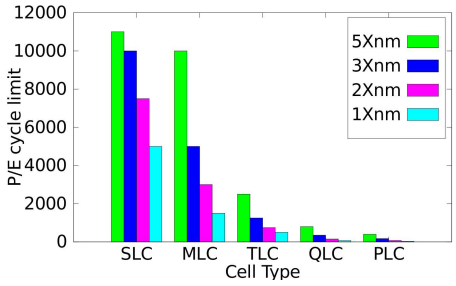


Fig 1:*Reduced P/E cycles with increased storage density [5]*

In our recently published workshop paper [12] we have shown great promise in improving endurance of multi-bit cell SSDs using WOM-v codes. WOM-v codes use a single lookup table to provide a low-performance overhead encoding scheme during a write and a read operation. While there exist other codes that allow additional overwrites before erase and are more space optimal, such codes traditionally involve multiple iterations before encoding and decoding is completed and are therefore performance inefficient. Further, WOM-v codes provide a family of codes that can be adapted to the amount of space overheads the underlying storage media can tolerate. In [12], we present theoretical back-of-the-envelope calculations based on a simplistic cell level model. Our calculations show that upto 500% additional writes can be done on QLC drives using WOM-v codes [12].

Although the theoretical results show significant improvement, it is not clear whether these improvements can be achieved in practice. First, in the real world, SSDs have several restrictions while overwriting data. SSD contains multiple erase blocks containing multiple pages. Writes to erase blocks is done at page granularity, where pages can be written to in sequential order from the first to the last page. Only a limited number of erase blocks can be programmed at a time. Unless

all the pages within the target erase block are invalidated, the SSD must first relocate all valid pages from target erase block to another erase block. Once the valid pages are relocated, the entire erase block is available to be reprogrammed. This "copy before overwrite" step, also called garbage collection, considerably increases the overall writes done to the device, which counters the gains we get from WOM-v codes. The evaluation in [12] falls short as it ignores the garbage collection workflow in SSDs.

Second, modern SSDs employ parallelism for higher performance. Multiple erase blocks are arranged in groups called *Erase Units (EUs)*. Only one EU is active at a time. Incoming data is first buffered and subsequently sharded across all erase blocks in the active EU. Any performance based metric computation may not capture such nuances involved in writing data to a shared buffer that requires preventing race conditions using locks, nor the gains due to striping the data across different parallel units. The evaluation in [12] fails to do any performance evaluation, and hence needs a real world flash emulator to evaluate the impact of WOM-v code on application performance.

Finally, real world workloads vary in their storage access patterns. The access pattern determines the amount of garbage collection in the device. A thorough analysis of WOM-v codes over multiple real world workloads is required to assess the practical gains of WOM-v codes introduced by [12].

The contribution of our paper is to demonstrate that WOM-v codes have real gains in practice. First, we present the first paper to our knowledge that provides a detailed design, implementation and evaluation of Non-Binary WOM codes on next generation dense SSD (QLC) drives to improve SSD lifetimes. Second, we show that there is a difference in theory and practice in the amount of SSD Erase Cycle reduction that can be achieved using WOM-v codes. With a system implementation and evaluation, we see more realistic gains than those provided by a theoretical evaluation. Moreover, counter intuitive to theoretical assumptions, we show that higher order WOM-v codes do not provide Erase Cycle reduction if the workload generates high write amplification. For high write amplification workloads, we provide two novel optimizations- GC-OPT and NR-Mode - which significantly reduce erase operations while retaining high performance. Third, our simulator is open-sourced and can be used as a test-bed to evaluate future WOM code designs on next generation SSDs. Finally, we show how WOM-v codes improve the lifetime of QLC flash by 4.4x-11.1x with negligible performance overheads.

2 Background

Traditional WOM codes were first proposed in the 1980s for media such as punch cards, where data is written bit-wise and each written bit can only be changed in one direction, e.g. from 0 to 1 [24]. More recently, WOM codes have attracted attention since their model of changing a written bit in only one direction matches the characteristics of a (single-level) flash cell. Prior work uses this observation by applying WOM codes to increase the lifetime of flash [32]. We refer to these WOM codes, which assume bits can only be changed from 0 to 1 as *binary-WOM* codes.

In our workshop paper [12] we observe that these binary WOM codes are a poor fit for new generations of flash cells, such as MLC or QLC, where more than one bit is encoded in a single cell. The real constraint for a flash cell is that the voltage level of a cell can always be increased (up to some maximum level V_{max}), but not decreased, independently of what the encoded bit values are. Adhering to a binary model creates unnecessary constraints that limit the power of a codes and also does not match device characteristics.

To quantify the limitations of a binary WOM codes for a QLC drive, we write a search program to compute the maximum number of generations writable using binary WOM(2,4) encoding scheme. We find that we are unable to write more than 2 generations of data using WOM(2,4) coding scheme. Hence, beyond two overwrites on a QLC drive, an erase operation will be required. Moreover, the code will have a 2x space amplification since 2 bits of data are encoded into 4 bits of codeword. Hence there is no net gain in using Binary WOM codes for QLC drives. Furthermore, if Binary WOM codes are implemented in a real SSD, the additional writes due to garbage collection will perform worse than not using WOM code at all. Hence in the remaining paper, we compare non-binary WOM-v(k,N) codes with non-encoded (NO-WOM) configuration.

Instead [12] proposes a new family of WOM codes for QLC flash, referred to as *voltage-based* WOM codes (WOM-v codes), that are based on the voltage level constraint and achieve a higher number of overwrites between erases.

For a detailed description of WOM-v codes for QLC flash, we refer the reader to [12]. However, for convenience we provide a high-level summary of how these codes work in the remainder of this section.

2.1 Introduction to WOM-v Codes

A WOM-v code has two parameters x and y and a WOM-v(x,y) code encodes x bits of data into a code word of y bits. In the case of QLC flash, y is equal to 4 and the 2^4 code words correspond to the 16 voltage values a QLC flash cell can have. Figure 2 shows an illustration of the three specific WOM-v codes for QLC flash [12] presents: WOM-v(3,4), WOM-v(2, 4) and WOM-v(1,4).

For a simplified explanation of WOM-v codes consider the WOM-v(3,4) code on the left in Figure 2. The column labelled “CODE” shows the mapping of code words to a cell’s voltage levels V0 to V15, where V0 is the lowest and V15 is the highest voltage level. The left column labelled “DATA” shows how the 3-bit data words are mapped to the 16 voltage levels.

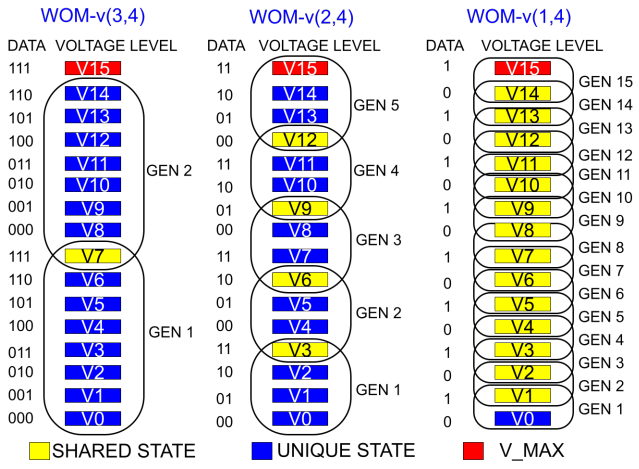


Figure 2: *Voltage based codes for encoding 3 2 and 1 data bit(s) into 16 Voltage Levels. Each oval represents a generation. Each generation contains a set of Voltage Levels mapped to a set of all possible data bit sequences for a coding scheme.*

Note that each 3-bit data word appears twice in the "DATA" column, once mapped to the bottom half of voltage levels and once mapped to the top half of voltage levels. We refer to the bottom half of voltage levels as generation 1 (GEN1) and the top half as generation 2 (GEN2). Having for each data word a corresponding mapping in each generation ensures that we can write to a freshly erased cell as many times as we have generations without erasing. To see how consider again the example of the WOM-v(3,4) code: If we ensure that the first write to a cell (after an erase) maps the data word to the bottom half of the voltage levels (GEN1), then we can map the second write of any data word to the top half of voltage levels (GEN2). Note that writing to a higher generation only requires increasing a cells voltage, which can be done without an erase. For example, sequentially writing the two data words 101 and 001 to a freshly erased cell would involve programming the cell first to the voltage level corresponding to V5 and V9 respectively. We define each iteration of write to the underlying device as one *write cycle*.

2.2 Optimizations in WOM-v Codes

While the above describes the basic idea behind WOM-v codes, our workshop paper [12], describes 3 optimizations that further improve the efficiency of WOM-v codes:

1. Same-generation transitions: Sometimes it will be possible to write a data word without increasing voltage levels into the next generation. Consider the example of sequentially writing the two data words 001 and 101 to a freshly erased WOM-v(3,4) cell. Writing 001 raises the voltage to the level corresponding to V1 in GEN1. Now observe that the second data word, 101, has a mapping within GEN 1 (V5) that corresponds to a higher voltage level than the current voltage level (V1). In this case we do not need to transition to GEN2, but can perform the second write while staying at a GEN1 voltage level. In this case we will be able to write to the cell more often than the number of generations.

2. Code word sharing: The reader might have noticed that voltage level V7 in WOM-v(3,4) cell is shared between GEN1 and GEN2. The data word 111 maps to V7 in both GEN1 and GEN2. This sharing allows us to squeeze in more generations. The savings become more obvious when considering WOM-v(2,4), where sharing allows us to create 5 generations instead of otherwise only 4.

3. Using ECC to increase page writes between erases: As writes happen at the granularity of pages (not individual cells) we can no longer overwrite a page once any one of its cell has reached its maximum generation. To continue writing to a page with a few cells in the max generation without erasing [12] suggests to make use of pre-existing device error correcting code (ECC). Since flash media are naturally prone to bit errors, in particular with increasing age, any flash-based SSD incorporates ECC. The idea is that if there is only a small number of cells in a page that have reached the max generation and therefore cannot be written to and we had a way to mark these cells as *invalid* while writing to the rest of the page, a later read could rely on the SSD's existing ECC to determine the value of these cells.

An obvious question is whether using existing ECC to reconstruct the content of cells that are marked *invalid* (because they had reached the max generation and could not be written to) will affect reliability or performance of the drive. We make the case that if done right neither reliability nor performance will be affected. The first key observation is that bit error rates of SSDs grow with age/wear-out and that the ECC is provisioned to be able to handle the high bit error rates toward the end of the drive's lifetime. That means that for the majority of a drive's lifetime the ECC is over-provisioned - it is stronger than what is required to ensure drive reliability. Therefore during the first years of a drive's operation, when bit error rates are lower than what the ECC is designed for, the ECC can handle the correction of a certain number of invalid cells without any impact on reliability. To control the impact on reliability we set a threshold called *ECC_threshold* on the fraction of cells per page that are allowed to be marked as invalid before an erase operation is required. If the number of cells in a page that are in their maximum generation reaches this ECC_threshold, further overwrites are not allowed and the page needs to be erased first. In practice, this threshold could be chosen dynamically based on age or observed bit error rates of the SSD.

The second key observation with respect to reliability is that recovering the value of invalid cells is easier than recovering from random bit errors that the ECC is designed to handle. Since the location of the invalid cells is known (in contrast to the unknown random location of bit errors) the ECC only needs to perform erasure *correction* rather than detection and correction. Since correcting x erased bits requires half the

redundancy as detecting and correcting x erroneous bits [10], it is less likely for the ECC to fail to reconstruct invalid cells.

The discussion above should also make it clear that our use of the existing ECC should not or only minimally affect read latency or throughput. The reduction in performance due to correcting reads of V_{max} cells is the same as that of correcting a corrupted bit. Finally, the use of ECC to enhance WOM-v code performance is optional, WOM-v code can work with lower endurance and maintain similar performance as No-WOM configuration if the underlying device ECC is not used.

2.3 Flash Friendliness of WOM-v(k,N) codes

Besides reducing the number of required erase operations, WOM-v codes also have the added benefit that their writes are more flash friendly as they only involve voltage level increases within a short range of voltage levels. For instance, in a standard QLC drive with 16 distinct voltage levels, the voltage increment for a NO-WOM configuration could be anywhere between V0 to V15. However using a WOM-v scheme, the voltage will only monotonically increment by a factor of one generation (eg. the next 4 voltage levels in WOM-v(2,4) coding scheme). In general, for a WOM-v(k,N) coding scheme, the next write will only increase the voltage level from 0 to the next $2^k - 1$ levels as compared to No-WOM configuration where the voltage may increase anywhere between 0 to 2^N levels. The implications of this lower rate of increment in voltage level during a write operation for WOM-v(k,N) code is that less amount of charge will be injected to each cell as compared to NO-WOM configuration. This mode of programming a cell in WOM-v encoding scheme is more flash friendly as it induces less program disturb errors in both programmed cell and neighbouring cells as compared to NO-WOM configuration. Further, gradual voltage increment may also simplify SSD circuitry as the number of possible transitions on each state is significantly reduced in WOM-v configuration as compared to the NO-WOM configuration.

In this work, we focus our attention on reducing the number of erase operations in exchange for more number of writes to improve SSD endurance. We acknowledge that there are other factors, including write operations, and temperature that would impact the endurance of SSD drives. However, such factors have much less impact on the endurance of SSD drives as compared to the erase operation [11].

3 System Implementation

This section presents the system implementation that we use to evaluate the real world potential of WOM-v codes. With a systems implementation of the WOM-v code, we are able to measure the practical reduction in the number of Erase Units (EUs) erased in SSDs. We are also able to measure the impact of input data contents, workload patterns and performance overheads of WOM-v codes that cannot be accurately measured in theory. This is because SSDs are programmed at the granularity of pages and not cells. Further, SSDs are erased at the granularity of EUs before additional data can be written on them. Ideally, we would want to implement WOM-v code on a real world hardware by manipulating the device Flash Translation Layer(FTL). However, we have the following challenges: 1) SSD FTL is a proprietary closed sourced software that is not available for change. 2) An evaluation done on one hardware device configuration may not be conclusive and applicable to future SSD generations.

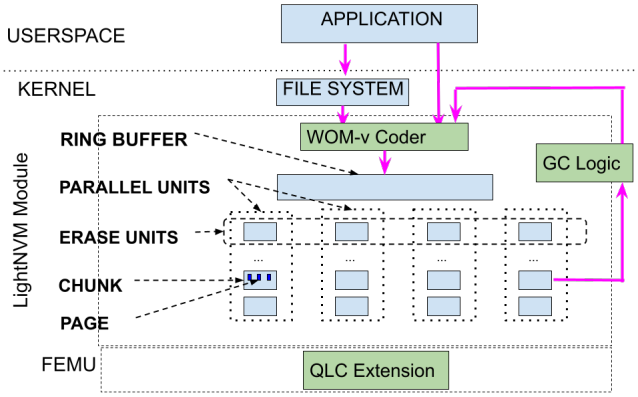


Figure 3: *LightNVM Architecture with our contributions in green. The LightNVM and FEMU modules together emulate a host-managed-SSD device. Our WOM-v encoder logic is inside the LightNVM module. We change the garbage collection workflow to perform erase operations based on WOM-v configuration type. We add QLC-flash support to FEMU.*

To address these issues, we implement WOM-v code in the Linux LightNVM Open-Channel SSD Subsystem module [9], which allows making changes in the device FTL. We add `445` LOC in the LightNVM module and `220` LOC in FEMU. In order to emulate a QLC device, we extend FEMU, a widely used Flash Emulator [17] for MLC devices, to emulate a QLC device. Our extension for QLC support is already merged into the mainline FEMU repository [6]. The WOM-v implementation requires no changes in the application or file system running on top of the device. The computation overheads are negligible as encoding and decoding involve simple table lookups. Although the WOM-v coding scheme is built within the LightNVM module, in reality, some of the key functionality might be implemented using special purpose hardware. We leave more efficient hardware designs of encode and decode operations as part of future work.

3.1 LightNVM Architecture

LightNVM is a Linux module that exposes the underlying architecture of a real or an emulated NVMe SSD to the host. This helps us to make modifications in the way we write

to and read from the device. LightNVM also enables us to control the garbage collection scheme and when an erase operation should be performed on the underlying device. The internal architecture of LightNVM is as shown in Figure 3. The two main data structures of a LightNVM module are 1) a shared ring buffer and 2) Parallel Units.

3.1.1 Ring Buffer

The ring buffer is a circular buffer where data is placed before being written to the underlying device. The device may be accessed either directly by the application or through the file system as shown in Figure 3. Once the ring buffer is full or the user requests a sync operation, the data copied to the ring buffer is striped across different Parallel Units (see 3.1.2) in a round robin order. The ring buffer is a shared resource between two threads, the *user-write thread* that copies the incoming application/file-system data to the ring-buffer and the *gc thread* that copies valid pages from the underlying device to the ring-buffer during garbage collection. The ring buffer and the device configuration remains unchanged across NO-WOM and WOM-v(k,N) configurations.

3.1.2 Parallel Units

Figure 3 shows a device with 4 Parallel Units. A Parallel Unit(PU) is an independent unit of storage on the device. Each Parallel Unit is divided into multiple erase blocks or *chunks*. Each *chunk* contains a linear array of *pages* that can be sequentially programmed from the first page to the last page of the chunk. A group of same-sized chunks, one chunk from each Parallel Unit, forms an *erase unit (EU)*. Pages within a chunk are sequentially programmed. Pages across chunks within an EU are programmed in parallel. As a result, all chunks in an EU get filled at the same time. Furthermore, all chunks in an EU can only be erased together and hence are garbage collected at the same time in the default setup. At any time, a single EU is opened for application writes. The EU is closed once all pages in the erase-unit have been programmed.

We can issue 3 types of operations to each parallel unit - 1) page read 2) page write and 3) chunk erase which is issued in parallel to all chunks within an EU. Reads on the emulated device are 10 times faster than writes, and erase operations are 10 times slower than writes.

All operations are performed sequentially within a Parallel Unit. Two operations on different Parallel Units can be performed in parallel. The number of parallel units on an emulated LightNVM module is configurable. We use the default 4 Parallel Units for all our experiments.

It is important to note that our implementation of WOM-v codes does not make any changes to how parallelism across Parallel Units or sequentiality within parallel units works. In 4.2.3 we show that the performance gains provided by LightNVM parallelism are retained even after using WOM-v(k,N) codes.

3.1.3 Write and Read Operation

All page writes are staged on the ring buffer. If the data available in the ring buffer is small and a *sync* command is issued by the user, the data to be written is appropriately padded for alignment and striped across Parallel Units. At any give time a single EU, called the *active EU*, is open for writes. Equal number of pages are simultaneously written to all chunks of an active EU, until the last page of all chunks have been programmed. Once the *active EU* has been filled, the EU is closed and a new EU is made *active* and opened for future writes.

All read operations are sent to the device as a block I/O (*bio*) request. The LightNVM module first translates the logical block address (LBA) of the requested page from the *bio* structure to the device Physical Page Address (PPA) using the Logical-to-Physical (L2P) Map. The page contents are then copied from the device PPA to the *bio* request and returned back to the user.

3.1.4 Garbage Collection

LightNVM employs garbage collection to reclaim space occupied by invalidated pages. A page gets either explicitly invalidated by a TRIM command, or implicitly because the logical page stored in it gets overwritten by the application. To free SSD pages occupied by such invalidated pages, a closed EU is opened in the *gc-mode* by the *gc-thread* for garbage collection. In the garbage collection phase, all valid pages from a *gc-erase-unit* are copied to the ring buffer. The entire EU is then erased and closed. This EU is returned back to the free pool of erase-units available for the *user-write* thread to be opened for future writes.

LightNVM follows a greedy approach to select an EU to be garbage collected. An EU with the maximum number of invalidated pages is chosen first. LightNVM reserves an over provisioned space of 11% in order to not run out of space while performing garbage collection. Additionally, LightNVM has no wear-leveling mechanism and delegates wear-leveling to lower level drive FTL.

3.2 WOM-v Implementation

To incorporate WOM-v codes in the LightNVM code, first, all writes to the device need to be encoded. Second, all reads issued to the device need to decode previously written data. Third, the default garbage collection logic needs to be modified. Instead of erasing all the EUs during garbage collection, an erase should now be selectively done based on the state of the pages within an EU. Fourth, for our experiments, the underlying device emulator needs to support next generation

SSD devices with QLC or denser flash medium. Finally, we implement two optimizations, which while they do not change the design of WOM codes, help to improve the performance and reduce the overheads of WOM coding. We first present the baseline implementation without these improvements in the next subsection and then present the optimizations in subsection 3.2.2

3.2.1 Baseline Implementation

We add the following components to the LightNVM module (highlighted in green in Figure 3): 1) encode and decode logic 2) WOM-v aware garbage collection logic and 3) QLC support for FEMU. Our framework is extendable to emulate future SSDs and future coding schemes.

Write Operation

An application or a file system can submit a write request to LightNVM. All writes are encoded before being written to the drive. By default, a WOM coding scheme first reads the previously written data on the media. This data is encoded and overwritten on the physical page, maintaining the voltage based constraint of the underlying media. Since this default methodology causes increased read amplification, we present a simple mechanism to avoid such reads altogether for WOM-v codes using the No-Reads configuration as discussed in 3.2.2.

During a write operation, the ring buffer creates a mapping between the logical block address (LBA) of pages staged in the ring buffer to the destination physical page address (PPA) of the pages on the device before writing the pages to the device. We intercept all writes at this stage and apply the following transformation: First, we read preexisting encoded data in the PPA of all pages being written. (In 3.2.2 we describe a No-Reads configuration that obviates the need for this read operation). Next we encode incoming pages using the preexisting data. The encoding scheme is straightforward and involves a simple lookup in the static WOM-v(k,N) encode table shown in Figure 2. Finally, we write the new encoded pages to the device PPA on the drive.

For a WOM-v(1,4) coding scheme, each page incoming write is encoded and stored in 4 x 4KB physical of a QLC page on the device. Similarly, for a WOM-v(2,4) coding scheme, each 4KB incoming write is encoded and stored in 2 x 4KB physical pages on the device. To reduce the performance overheads of additional page writes, we increase the logical page size at which an application page is sent to the device after an encode operation to 16KB and 8KB for the WOM-v(1,4) and WOM-v(2,4) configuration respectively. We maintain logical page locality among all encoded pages. i.e. all pages belonging to the original logical page are encoded into consecutive logical LBAs. We describe the importance of logical page locality during reads in the next sub-section.

Read Operation

In the read workflow, the original read block I/O (*bio*) request from the application is first translated into the corresponding consecutive LBA address *bio* requests. The consecutive pages correspond to a single page due to logical page locality. Next, all encoded pages that were read are decoded in the read return path. The decoded data is copied to the originally submitted *bio* request structure and can be read by the application with no modifications. The decoding scheme is straightforward and involves a simple lookup in the static WOM-v(k,N) encode table shown in Figure 2.

Garbage Collection

Like the standard LightNVM garbage collection mechanisms, our implementation chooses the erase unit with the smallest number of valid pages for garbage collection and copies out all valid pages to the ring buffer. However, while the standard scheme would now erase the erase unit, our modified scheme will erase the erase unit only if any of its pages (valid or invalid) have reached the *ECC_threshold* on the number of cells in the maximum generation (recall Section 2.2).

We set the *ECC_threshold* to 3% in our experiments, i.e. we erase an erase-unit that has any page with more than 3% cells in *GEN_MAX*. This threshold is chosen based on the theoretical evaluation in [12] combined with the fact that current devices have reported 7% ECC in each page [23]. We predict higher ECC being reserved for QLC and future generation drives.

Cell and Page Layout

In a standard SSD based on N-level cells each cell can be programmed to take on one of 2^N different voltage levels. Our work assumes the same type of cells as a standard SSD with the same number of voltage levels and mechanisms for performing voltage increments. However, we change how these cells and voltage levels are used to store data. Besides the encoding of data as described in Section 2, we also change how cells are assigned to pages. Conventionally, for a N-level cell drive, 1 cell stores N-bits of data, where each bit is located on N different pages. For example, in QLC flash, 1 cell stores 4 bits of information, and each bit is located in 4 different pages of the drive. The drawback of this approach is that the 4 pages have to be programmed in a certain order. For WOM-v codes, we propose that instead of mapping each cell to 4 bits in 4 different pages, map 4 bits of information representing 1 cell in a single page. This mapping helps us program a single cell of the page to a voltage value of our choice. Using this technique, for a WOM-v(2,4) code, we will be able to encode 1 4KB logical page into 8KB data and store this data in 2K cells of the flash media. Similarly, for a WOM-v(1,4) code, 1 logical page of 4KB will get encoded as 16KB of data and be written to 4K cells of the flash media. We also increase

the logical page size to 2x and 4x the size of original logical page size for WOM-v(2,4) and WOM-v(1,4) configurations respectively. The implication of the above approach is that we no longer have to maintain a specific sequence or order of page programming. Instead, each cell stored on a single page can be programmed to a specific voltage range determined by the generation of the cell in the write cycle.

A WOM-v(k,N) scheme can be naturally extended to any N-level cell by changing the value of N to the number of bits each cell of the device. The value of k determines the space-endurance tradeoff - a lower value of k will give higher endurance but also consume more physical space.

3.2.2 WOM-v Optimizations

We identify two optimizations to the baseline implementation of WOM-v(k,N) codes. First, we present `GC_OPT Mode`, a novel methodology for garbage collection in WOM-v(k,N) configuration that improves SSD endurance considerably by delaying valid page rewrites during garbage collection. Second, we present `NR Mode` WOM coding scheme, a technique to perform encode operations and overwrite an invalidated page without reading the previously existing contents of the page which completely eliminates read amplification during writes.

GC_OPT Mode

The goal of this optimization is to reduce the write-amplification caused by copying out valid pages from an EU during garbage collection. The key observation is that with WOM-v codes in most cases the invalid pages in an EU can be over-written again, without first erasing them. Recall from Section 3.2.1 that we make use of this fact and only perform an erase when a page in an EU has reached its ECC_threshold. That means that in those cases where no erase is required for an EU, we can leave the valid pages in place (without copying them out), *as long as we skip writing to valid pages* when writing to this EU in the future. (The remaining, non-valid pages within a chunk will be overwritten in the same specific order as before in order to minimize cell-to-cell interference, and writes are parallelized across the parallel units of an EU, as before.)

No-Read (NR) Mode

The WOM-v codes we introduce share a source of potentially major performance overheads with other WOM codes previously proposed: to write to a cell we need to know the cell's contents in order to encode the data to be written. Therefore each write necessitates a prior read. In this subsection we make two observations about WOM-v(k,N) codes that allow us to eliminate this read-before-write: (1) The only reason we need to read a cell's contents before writing to it, is to determine which generation to use for the write. (2) If we remove the same generation transition optimization (recall Section 2) and instead in each write cycle move to the next generation, we can store the most recently used generation for each page with a chunk's metadata. When opening an EU for writing, this metadata can be loaded into memory and all writes to pages in the EU are done using the generation information in the metadata (obviating the need for reading the page). We refer to the above method for eliminating the read-before-write as *NR (No-Read) mode*.

We note that NR mode has two potential downsides: First, same generation transitions are no longer possible, which might reduce the gains achieved with WOM-v codes. We discuss the tradeoffs between improved performance gains and reduced endurance gains using NR mode in detail in Section 4.2.3. Second, NR-mode require additional storage and memory by storing in a chunk's metadata the generation for each page. However, this added data is on the order of a few bits for each page, which seems negligible (2 bits per page) as compared to pre-existing in-memory metadata such as the logical to physical map (32-64 bits per page).

3.2.3 Adding QLC Support to FEMU

We use FEMU [17] to emulate the underlying SSD media. FEMU emulates the SSD in main-memory and adds predictable I/O latency to each I/O request to mimic a real Open Channel SSD device. In order to read or write a page, a specific number of reference voltages need to be applied to access the page. The number of reference voltages applied increases with the increase in flash density [14].

The main challenge in using FEMU for new generation SSD device emulation is that the existing FEMU emulator only supports an MLC SSD with 2 page levels. A page can either be an Upper or a Lower page with write latency of 850μs and 2300μs and the corresponding read latency of 48μs and 64μs respectively. FEMU also adds a constant NAND read, write and erase latency of 40μs, 200μs and 2ms to each read, write and erase I/O request respectively.

We modify the default MLC configuration of FEMU's page layout in each chunk. Instead of having alternating upper and lower pages with varying latency in each chunk, we extend FEMU for QLC emulation: Each chunk in a QLC device has alternating Lower(L), Center-Lower(CL), Centre-Upper(CU), Upper(U) pages. A write latency of 850μs, 2300μs, 3750μs and 5200μs, and read latency of 48μs, 64μs, 80μs and 96μs is applied to L,CL,CU and U pages respectively based on the number of reference voltages [14] required to read a specific page type.

3.2.4 Testbed for Future SSDs and Coding Schemes

Our WOM-v simulator is generic and can be used as a testbed for denser SSD or higher order coding schemes. In order to add a new WOM-v(k,N) coding scheme, first, the user has

to provide a simple lookup table mapping each data word to a voltage level similar to WOM-v tables shown in Figure 2. Second, the user optionally sets an *ECC_threshold* value. Finally, the user specifies the latency of additional page levels for next generation SSDs in FEMU.

Our emulator can also be used standalone without any coding scheme. We have open-sourced our generic N-LC Simulator for more advanced coding and next generation SSD research [7].

4 Evaluation

In order to evaluate the gains of WOM-v(k,N) codes on QLC drives, it is important to evaluate the endurance gains and performance tradeoffs associated with WOM-v(k,N) codes. In particular, WOM-v(k,N) codes can potentially reduce the number of EUs by enabling overwrites between erases. However, WOM-v(k,N) also introduces space amplification during writes, which increase EUs and affect performance. Further, both read and write workflows in WOM-v(k,N) codes introduce read amplification for the device.

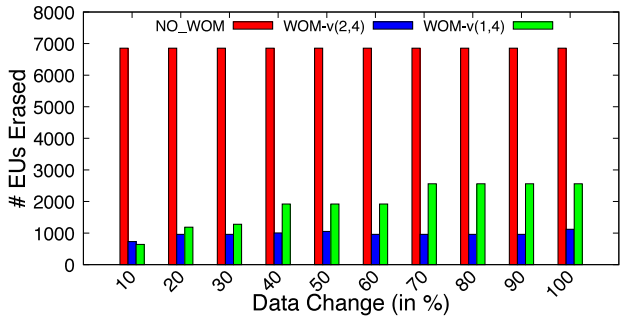
The goal of this section is to use our implementation to evaluate the EC reduction and the impact on performance for both micro-benchmarks and real world workload traces.

Since WOM-v(k,N) requires more space than NO_WOM configuration, there are two ways to compare them. The first option is to keep the size of the user facing logical address space constant, and increase the physical space allocated to WOM-v(k,N) configuration by a factor of N/k. For a fair comparison of EC reduction per hardware chip, we have to divide the resultant number of ECs encountered by NO_WOM code by N/k to account for additional space provided to the WOM-v(k,N) scheme.

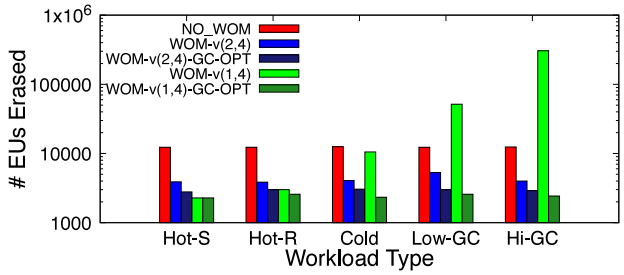
A second more practical approach to evaluate SSD hardware is to keep the physical capacity of the hardware constant, and reduce the logical block address range by a factor of N/k for WOM-v(k,N) coding scheme. For example, we use the same 4 GB physical space in NO_WOM and WOM-v(k,N) configurations. For NO_WOM, the entire 4 GB logical address space is exposed to the application, while for all modes of WOM-v(2,4) codes (i.e, WOM-v(2,4), WOM-v(2,4)-GC-OPT and WOM-v(2,4)-NR) the range of logical address space on the same 4 GB drive is reduced to a 2GB logical space, and similarly for all modes of WOM-v(1,4) codes, the same drive is exposed as a 1 GB logical space. We assume the workload running on such a drive to have a logical space of 1 GB.

Under the same workload this setup leads to more empty space for NO_WOM configurations which acts as extra over provisioned space to reduce garbage collection overheads. However, we show that for the same physical device WOM-v(k,N) is able to successfully reduce the number of ECs in all our experiments with minor performance overheads.

In the rest of this section, we compare the default SSD configuration (NO_WOM) with the baseline implementation



(a) *Effect of rate of data change on reduction in the number of EUs erased.*



(b) *Effect of access pattern on reduction in the number of EUs erased.*

Figure 4: *Endurance measurement for microbenchmarks*

of WOM-v(k,N) codes, garbage collection optimized (WOM-v(k,N)-GC_OPT) and performance optimized no-reads mode (WOM-v(k,N)-NR) implementations. We did not implement binary WOM(k,N) code in our emulator as the overall gains will be even lower than those obtained in No-WOM configuration (recall Section 2).

4.1 Micro Benchmarks

We use micro-benchmarks to study the impact of specific access patterns and data-block contents, on WOM-v(k,N) codes. Each micro-benchmark writes 25 GB data to 1 GB physical emulated drive containing 4 Parallel Units and 160 EUs.

4.1.1 Effect of change in data buffer contents

The gain of the baseline WOM-v(k,N) codes depend on the data contents of the block that is overwritten to the existing data, due to the possibility of same generation transitions in WOM-v codes.

In this micro-benchmark, we fill the drive sequentially so no garbage collection is invoked. Once the drive is full, we flip a fraction of all bits in the data buffer, and fill the entire drive again with this modified data. We repeat this multiple times (Figure 4a X-Axis) and compare the number of EUs erased as the rate of change in data buffer contents increases.

Figure 4a shows the total number of EUs erased for varying

data buffer contents during writes. We note that for all types of data buffer contents, WOM-v codes reduce the number of EUs erased compared to the NO_WOM configuration. However, the rate at which a cell reaches the maximum voltage level will be slower when there is smaller rate of data change between subsequent writes. For workloads with a higher amount of data change, the maximum voltage level will be reached faster, and so the EU erase gains will be lower. Unlike WOM-v codes, the NO_WOM configurations' EUs erased remain constantly at the higher end irrespective of the data buffer contents. The number of EUs erased in the NO_WOM configuration also remain constant as we run the same sequential pattern of write workload with different data contents.

4.1.2 Effect of access Pattern

In order to measure the impact of access patterns, we create micro-benchmarks that invalidate previously written pages in a specific order to cause varying degrees of device write amplification: *Hot-S* keeps updating the same data sequentially, *Hot-R* updates the same data in a random order, *Cold* only updates a fraction of pages, *Low-GC* and *High-GC* generate medium and high amount of Garbage collection, respectively.

Figure 4b shows the number of EUs erased for different workload patterns. Across all benchmarks, we observe that WOM-v(1,2) codes significantly reduce the number of EUs erased. However, with higher number of overwrites due to GC, we see diminishing gains for WOM-v(1,4) code. Finally, GC_OPT mode is able to alleviate the problems incurred and maintain low write amplification overhead.

Hot-S keeps all pages "hot" i.e. uniformly accessed over the course of the benchmark. During garbage collection no pages from the previously written EU are garbage collected and hence we do not have any write amplification. We observe that *Hot-R* also does not create any garbage collection. Hence the pattern of write between two workloads does not impact the gains by WOM-v codes if the garbage collection thread writes minimal or no additional pages to the drive.

For the *Cold* benchmark, we observe a slight increase in the total number of EUs erased for the WOM-v(2,4) configuration and an order of magnitude increase in EU erases for the WOM-v(1,4) configuration as compared to *Hot-S* and *Hot-R* configurations. This is due to localized writes on only a subset of the device. We also observe that WOM-v(1,4)-GC-OPT and WOM-v(2,4)-GC-OPT continue to maintain lower write amplification for this benchmark as there is almost always a candidate EU with at-least a single programmable page, and hence it significantly reduces the number of valid pages from hot EUs that are relocated.

Low-GC creates a moderate amount of write amplification in the NO_WOM configuration. The WOM-v(2,4) configuration continues to outperform NO_WOM. But the WOM-v(1,4) configuration starts performing poorly as compared to the NO_WOM configuration. This is because additional writes generated space amplification in the WOM-v(1,4) configuration. Even with Low-GC, since there are continuous page invalidations, both WOM-v(2,4)-GC-OPT and WOM-v(1,4)-GC-OPT continue to reprogram an EU without relocating valid pages in a significant portions of reprogram operations, which keeps the write amplification relatively low.

High-GC benchmarks cause severe write amplification due to high influx of valid pages recycled during garbage collection. This causes WOM-v(1,4) to perform two orders of magnitude worse than NO_WOM. However, even in High-GC mode, WOM-v(1,4) GC_OPT continues to find EUs that have intermediate pages available for reprogramming for a significant fraction of reprogram operations, and hence the write amplification remains consistent over the course of the workload run.

Source	# Traces	Medium	Year
Alibaba [18]	814	SSD	2020
RocksDB/YCSB Trace [25, 30]	1	SSD	2020
Microsoft Cambridge [22]	11	HDD	2008
Microsoft Production [15]	9	HDD	2008
FIU [16]	7	HDD	2010

Table 1: *Historical HDD and recent SSD based block traces*

4.1.3 Conclusions from micro benchmarks

We conclude that across all data write patterns, WOM-v codes are highly effective in reducing the number of EUs erased. For workloads where similar or incremental data is overwritten on the device, huge gains are possible.

WOM-v(2,4) codes are highly robust to different kinds of workload patterns. For workloads that exhibit increased garbage collection, WOM-v(k,N)-GC-OPT codes continue to maintain near constant EUs erased even for artificial, extremely-high garbage collection workload.

4.2 Real World Evaluation

4.2.1 Setup

Trace Selection: Table 1 shows a summary of 844 real world traces from 5 different sources. We shortlist and present 10 write-based traces and 6-read-based traces representing each source.The write-based workloads have higher number of writes than reads and help us to measure the endurance improvement (Section 4.2.2) and write performance tradeoffs (Section 4.2.3).The read-based workloads help us better understand the impact of WOM-v codes on read performance (Section 4.2.3).

The selected traces have at-least 1 million and at most 50 million page writes. We also select traces with varying number of unique block accesses and varying number of updates per unique block. We choose enterprise workloads (except [30]) instead of synthetic benchmarks for our evaluation for two reasons. First, the enterprise workloads exhibit significantly different workload characteristics than the TPC and

FIO benchmarks, that are specifically designed to stress the system under test. Second, enterprise workloads show variation in usage over time, for example due to diurnal patterns.

There are a few limitations of using real-world workload traces. First, there are no publicly available block traces where TRIM information is available. We consider a block to only be invalidated if it is overwritten. Hence our estimate of gains using WOM-v codes are rather conservative. When considering the addition of pages invalidated with the TRIM command, there will be more opportunity for reprogramming physical pages that contain TRIM'ed logical pages in WOM-v codes. Moreover, we will have lower write amplification induced by garbage collection that will favour WOM-v codes.

Second, block-traces do not have any information about the buffer contents. We fill each block with random data. This does not impact NO_WOM configuration, but impacts the WOM-v configurations and higher gains could be possible if the data is more uniform causing less state change. The Microsoft Cambridge (MC-stg, MC-rsch), Florida International University (FIU-online,FIU-websearch) and Microsoft Production (MP-backend, MP-authentication) are older traces collected on HDDs. The Alibaba workloads (Alibaba-316, Alibaba-746, Alibaba-4) and RocksDB traces represent workloads that are aware of the underlying SSD device.

The traces require pre-processing for various reasons. The real world workloads are captured on different sized SSDs and HDDs. Further, traces only capture a subset of the logical block numbers of the entire drive. Also, our FEMU emulator models the underlying SSD storage in main memory. so the emulated drive size is much smaller than the original SSD or HDD on which the trace was captured.

To standardize these traces, we pre-process each real-world trace and reduce it to a format that can be run on a constant sized 16GB FEMU Based LightNVM emulator. During trace reduction, we ensure that the access pattern of each page in the original and reduced trace remains the same. We reduce the I/O size proportionately, if required and ensure the distribution of page update frequencies across the trace remains the same.

FEMU+LightNVM Drive Setup: When configuring the QLC drives for trace-based FEMU+LightNVM experiments, for each trace we set up a drive with the same *physical* capacity (same number of cells) for all NO_WOM and WOM-v(k,N) configurations. We size the capacity of the drive such that the NO_WOM drive would be half full (half of its capacity is used), which is common for real-world drives [8, 19, 20]. For each workload, we compute the number of unique blocks accessed, and set up a QLC drive that is twice the size of the total number of unique blocks accessed. We note that the logical address space for all WOM-v(2,4) setups is half the logical address space of NO_WOM. Similarly, the logical address space for WOM-v(1,4) setups is one-fourth the logical address space of NO_WOM.

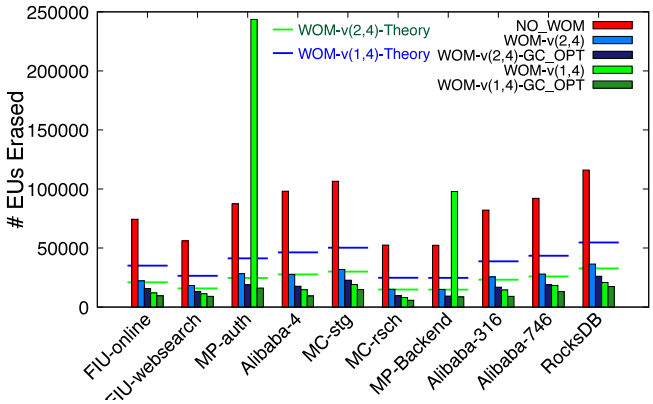


Figure 5: *Reduction in the number of Erase Units (EUs) erased using different configurations of WOM-v codes.*

4.2.2 Reduction in number of erase operations

This section uses our experimental testbed to evaluate the improvements in drive lifetime that different WOM-v codes achieve across a variety of workloads. In particular, we are interested in comparing the number of Erase Unit (EU) erases required to run the different workloads on a standard NO_WOM QLC drive and WOM-v enabled QLC drives, as these EU erases directly affect drive lifetime.

We first consider the question of whether a baseline WOM-v implementation, without any of the optimizations we propose in Section 3.2.2, can reduce the EU erases compared to a NO_WOM QLC drive. Toward this end, we observe in Figure 5, that the WOM-v(2,4) code reduces the total number of EU erased consistently by a factor of at least 3 (between 68% to 71% reduction) across all traces. Results for the WOM-v(1,4) code are a bit more mixed: In most cases WOM-v(1,4) further reduces the number of EU erasures, providing an additional factor of two reduction on top of WOM-v(2,4). Two notable exceptions are the MP-auth and MP-Backend traces, where the WOM-v(1,4) code erases more EUs than the NO_WOM configuration. This shows that there are certain workloads where the reduction in erase operations that WOM-v codes offer through overwrites between erases does not make up for the write amplification that comes with a higher rate code. (Recall that in a WOM-v(1,4) code each data bit gets encoded in 4 coded bits). The two MP workloads are more skewed in their popularity distribution, which results in higher garbage collection cost, and the added write amplification in WOM-v(1,4) codes gets amplified.

The poor performance of the WOM-v(1,4) codes for GC-intensive workloads motivates us to study the impact of the GC_OPT setting that we introduced in Section 3.2.2 to reduce write amplification during WOM-garbage collection. Figure 5 shows that enabling GC_OPT does indeed reduce the number of erases. With the GC_OPT setting both WOM-v(2,4)-

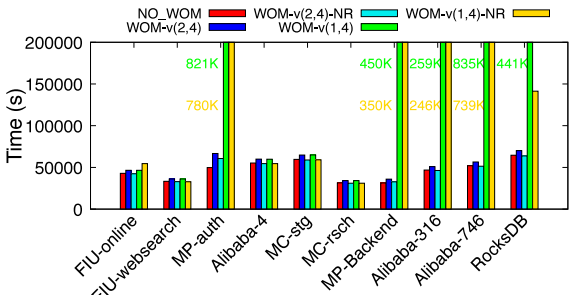


Figure 6: *No Read Mode Performance*

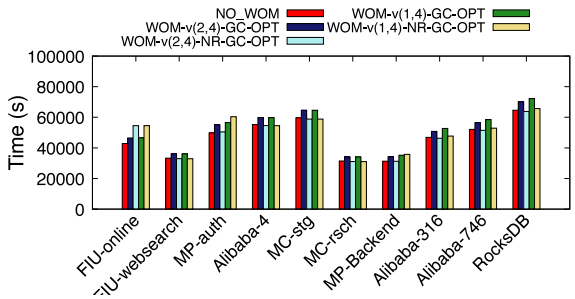


Figure 7: *No Read Mode with GC_OPT performance*

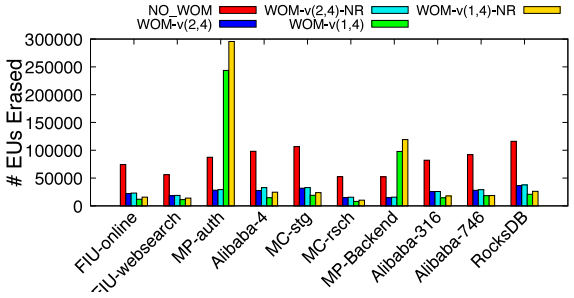


Figure 8: *No Read Mode endurance*

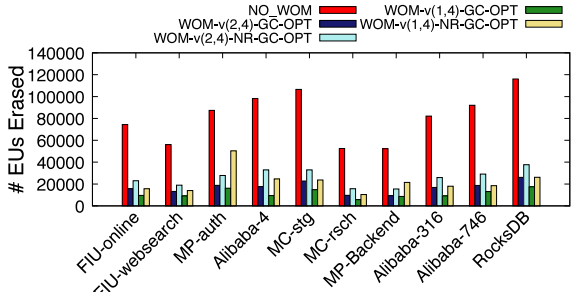


Figure 9: *No Read Mode with GC_OPT Mode endurance*

Figure 6 and 7 show performance improvement using NR Mode for WOM-v(k,N) baseline and WOM-v(k,N) GC_OPT modes, respectively, and Figure 8 and 9 compare endurance gains.

GC_OPT and WOM-v(1,4)-GC_OPT improve significantly over NO_WOM across *all workloads*, including the two MP workloads where the baseline WOM-v(1,4) did not improve over NO_WOM. The number of erases is reduced by 77-83% for WOM-v(2,4)-GC-OPT and 82-91% for WOM-v(1,4)-GC-OPT compared to NO_WOM. This translates to $4.4-11.1\times$ reduction in erase operations for real-world workloads.

It is interesting to compare the results from our experimental evaluation to the simplistic back-of-the envelope results in [12], which estimates reductions in the number of erase operations by $5\times$ for WOM(1,4) and of $3.5\times$ for WOM(2,4). While we observe results in this range for some workloads, we are able to draw more differentiated conclusions. Improvements vary by workload and in particular GC-intensive workloads see significant improvements only if care is taken to optimize the GC process.

We also compared our results with theoretical estimates based on an analytical model presented by Yaakobi et al. in [29]. The authors in [29] derived an analytical model to analyze the gain that a WOM coding scheme can achieve in terms of reducing the expected number of erase operations, while considering the trade-off between the write amplification induced by the coding rate, including its effect on the garbage collection, and the number of reprogram operations the WOM code can perform between two erase operations. We use the number of write cycles and the coding rate in our different WOM-v code settings and the fullness of the drive based on our traces to determine the erase factor derived in [29] equation (3) and then convert the erase factor to the number of Erase Units erased. The corresponding lines are labelled WOM-v(k,N)-Theory in Figure 5. We observe that the experimental results of WOM-v codes achieve considerably larger reductions in erases than what was predicted by the analytical model. The reason is most likely that the analysis in [29] relies on several simplifying assumptions in order to be tractable, including for example an assumption that page invalidations happen uniformly at random. These observations underline the importance of trace-driven experimental evaluation of the benefits of WOM codes in SSD drives.

In conclusion, we demonstrate that WOM-v codes can greatly reduce the number of erase operations across a wide range of workloads. For higher rate codes, such as WOM-v(1,4) codes, and workloads that result in high garbage collection it is important to employ GC_OPT mode to limit write amplification during garbage collection.

4.2.3 Performance Evaluation

This section evaluates the impact of WOM-v(k,N) coding on a drive's performance. Section 3.2.2 already discussed how the read-before-write requirement of standard WOM codes can impact write performance and presented the WOM-v_NR optimization to eliminate this requirement (potentially at the cost of reduced endurance gains). A second potential source

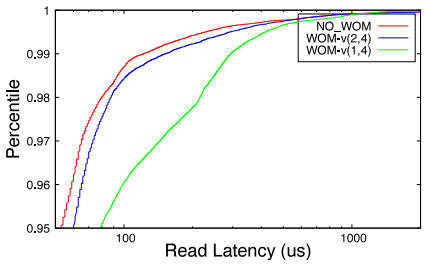


Figure 10: *MSR-Cambridge Web1*

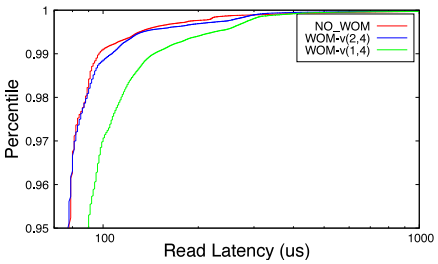


Figure 11: *MSR-Production DAPL*

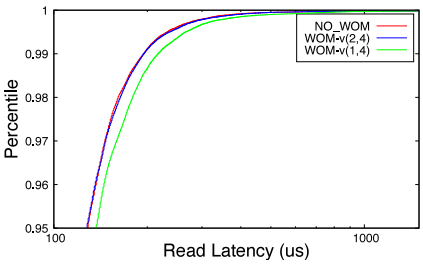


Figure 12: *Alibaba 3*

Figure 10, 11 and 12 show read tail latency of MSR-Cambridge Web1, MSR-Production Display Ads Payload and Alibaba-Server 3 workload with read:write ratios of 11:9, 4:3 and 5:6 respectively.

of performance impact is the generally higher write workload generated by WOM-v(k,N) as each write adds additional (N/k) data to be written. The goal of this section is to quantify the performance impacts of WOM-v(k,N) coding across different workloads.

Average performance / throughput Figure 6 compares the cumulative time to run our ten workloads on a NO_WOM drive versus WOM-v(k,N) drives with and without NR mode enabled. A larger cumulative time means lower system throughput and higher average request latency.

We observe that for the WOM-v(2,4) configuration the performance overheads are quite small, in the 3-8% range, compared to the NO_WOM configuration. Adding the NR mode optimization eliminates those overheads and leads to performance comparable to a NO_WOM configuration. At the same time we observe in Figure 8 that enabling NR mode for the WOM-v(2,4) configuration leads to only minimal reduction in the endurance gains compared to plain WOM-v(2,4).

We also evaluate whether adding the GC_OPT optimization, either in isolation, or in combination with NR mode will reduce performance impacts, in particular for the WOM-v(1,4) scheme, which has high overheads in the baseline implementation. The results are shown in Figure 7.

We find that enabling both GC_OPT and NR mode greatly reduces performance overheads for the WOM-v(1,4) scheme, bringing performance within 0-8% of the NO_WOM configuration. At the same time we observe in Figure 9 that when enabling GC_OPT and NR mode we achieve large endurance gains over a NO_WOM drive.

Read Tail Latency Finally, while the above discussion focused on average performance, we also considered tail latency, which is particularly important for read requests. Figure 10, 11 and 12 show the tail latency of read requests across three traces that we chose because they were read intensive. For all three cases, we observe that the 95'th percentile tail latency is 0.6-7% for NO_WOM and WOM-v(k,N) baseline encoding schemes, and that there is no large tail latency introduced due to the use of WOM-v(k,N) code.

4.3 Comparison with MLC Drives

Given that increases in drive endurance from WOM codes come with space overheads, an interesting question is how the WOM-v endurance/space tradeoff compares to simply using MLC technology. MLC cells have higher PE cycle limits than QLC drives, but are less space efficient. In particular, if we reduce a QLC drive to an MLC drive with the same number of physical cells, we would lose 50% of logical capacity (2 bits per cell instead of 4), which is identical to the logical capacity loss when applying WOM-v(2,4) codes to a QLC drive. In exchange for the 50% capacity loss, the MLC drive's endurance will increase because the PE cycle limit of a cell increases from 3K for a QLC drive to 10K for an MLC drive [1–4], and the WOM-QLC drive's endurance will increase compared to QLC due to overwrites between erases.

The goal of this section is to compare the endurance of an MLC drive to that of a WOM-v(2,4) QLC drive with the same logical capacity and the same number of physical cells. Endurance is the amount of user data that can be written before the PE cycle limit is reached. We can use our results from Section 4.2.2 to estimate endurance for a WOM-v(2,4) QLC drive for a given workload, based on the number of erases observed for the experiment with the corresponding trace and the amount of user data written in the trace. We also ran experiments for all workloads on an MLC drive with the same physical capacity and recorded those numbers.

Figure 13 compares the ratio of the endurance of an WOM-QLC drive and the endurance of an MLC drive for different workloads based on the methodology described above. We observe that for the same write pattern, the endurance of the WOM-QLC drive exceeds the endurance of the MLC drive in all scenarios (ratio larger than 1). This is the case even for the workloads with higher garbage collection where improvements from WOM codes were lower than for other workloads. On average the improvement in endurance is a factor of 3.5x for GC_OPT mode and 2.4x for GC_OPT-NR Mode, with negligible performance overheads as compared

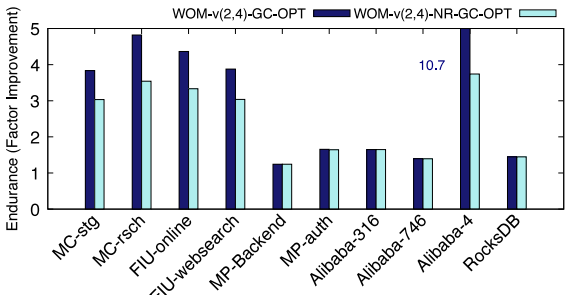


Figure 13: *The ratio of write endurance of WOM-v based QLC drives in GC_OPT and NR-GC_OPT mode and the write endurance of an MLC drive with the same logical capacity and the same number of physical cells. For all workloads, WOM-v(2,4) enabled QLC drives provide better endurance than MLC drives.*

to a NO-WOM QLC drive using the NR Optimization. In summary, we conclude that WOM-v codes can significantly improve drive endurance compared to standard MLC as well as QLC drives.

5 Related Work

The closest to this work is our own earlier workshop paper [12]. In [12], we introduced the idea of WOM-v codes and used a preliminary back of the envelope analysis to demonstrate their potential in theory. The goal of this paper is to study whether and how we can bring the advantages of WOM-v codes into practice where real world factors such as workload access patterns, device realities (write amplification, parallelism, etc) as well as its performance implications are taken into account. We provide a full implementation of WOM-v codes based on FEMU and LightNVM, including a discussion of implementation challenges and some optimizations we made, and an evaluation using micro-benchmarks as well as a trace-driven evaluation. We demonstrate that higher order WOM-v(k,N) baseline codes may lead to drastically higher endurance and performance overheads for real world, high garbage collection workloads. To alleviate this issue, we propose GC_OPT, a novel garbage collection mechanism that continues to provide endurance and performance gains for WOM-v(k,N) codes.

Other prior work [27, 28, 31] studies the use of traditional bit-based WOM codes for MLC and TLC flash. However, those studies focus on binary WOM coding scheme that cannot scale on higher density drives such as QLC drives. Further, in Section 2, we note that binary WOM codes do not provide gains for QLC drives. Our focus is a voltage-based coding scheme designed for high-density SSD drives.

Margaglia et. al. [21] present hardware limitations in MLC drives that reduce the gains achievable by WOM codes in practice. In particular, they observe that certain bit transitions are not feasible within the flash hardware. Such restrictions could appear for QLC drives as well. However, WOM-v(k,N) codes would avoid these restrictions as they operate directly based on voltage levels. Furthermore, WOM-v(k,N) codes present a family of codes - if a few state transition restrictions were imposed by the underlying hardware, one can use a lower code ratio WOM-v(k,N) code that is more suitable to the flash hardware restrictions. In [13], the authors show how erase operations are detrimental to NAND-flash. They recommend differential levels of programming and erase to normalize the amount of charge issued to erase a NAND flash chip. [13] is complementary to our work and both can be used together to improve overall flash lifetime.

6 Summary and Implications

Below we summarize our findings and their implications.

- WOM-v codes reduce the number of erase operations by 4.4-11.1x for QLC SSDs. This directly improves the lifetime of flash media and reduces the overhead of purchasing, replacing and restoring data from an older drive.
- Even for workloads that exhibit high-write amplification, WOM-v codes present a novel opportunity to delay garbage collection in GC_OPT mode until the entire erase unit is no longer programmable. Such optimization is possible *only* in WOM-v codes where data can be overwritten without erasing previous data on a SSD.
- Even for performance-critical workflows deployed on high-density SSDs, drive endurance can be enhanced to a significant degree using WOM-v codes without compromising on performance using the No-Reads optimization.
- The space amplification of WOM-v codes can be a cause of concern for SSDs that exhibit high space utilization. For such drives, we recommend routinely transitioning between WOM-v codes with different code rates based on the drive fullness or the space requirements of the workload. We leave evaluation of such dynamically transitioning coding schemes based on space-utilization as part of future work.
- While this paper demonstrates the effective use of WOM-v codes and discusses the implementation, evaluation, and optimization of WOM-v codes in the context of QLC SSDs, WOM-v codes can be easily extended to higher density, future generation of SSDs such as PLC SSDs and more sophisticated coding schemes.

Acknowledgements

We thank our USENIX FAST '22 reviewers and our shepherd Rob Ross for their detailed feedback and valuable suggestions. We thank Charles Xu for his help with trace selection and processing. This work was supported by an NSERC Discovery Grant and an NSERC Canada Research Chair.

References

[1] 4 QLC workloads and why they're a good fit for QLC NAND flash. https://www.techtarget.com/searchstorage/tip/4-QLC-workloads-and\-why-theyre-a-good-fit-for-QLC-NAND-flash. Accessed: 2022-01-25.

[2] QLC NAND – what can we expect from the technology? https://www.architecting.it/blog/qlc-nand/. Accessed: 2022-01-25.

[3] SLC vs MLC vs TLC vs QLC. https://memkor.com/slc-vs-mlc-vs-tlc%2Fqlc. Accessed: 2022-01-25.

[4] TLC vs QLC SSDs: The Ultimate Guide. https://storagereviews.net/tlc-vs-qlc-ssds/. Accessed: 2022-01-25.

[5] Western digital and toshiba talk up penta-level cell flash. https://blocksandfiles.com/2019/08/07/penta-level-cell-flash/. Accessed: 2020-03-24.

[6] FEMU TLC and QLC NAND support. https://github.com/ucare-uchicago/FEMU/pull/47, 2021.

[7] WOM-v Source Code. https://github.com/uoftsystems/womv, 2022.

[8] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. In *5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. USENIX Association.

[9] Matias Bjørling, Javier González, and Philippe Bonnet. Lightnvm: The linux open-channel {SSD} subsystem. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 359–374, 2017.

[10] Thomas M. Cover and Joy A. Thomas. Elements of information theory, 2nd edition. Wiley, 2006.

[11] Peter Desnoyers. What systems researchers need to know about nand flash. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'13, page 6, USA, 2013. USENIX Association.

[12] Shehbaz Jaffer, Kaveh Mahdaviani, and Bianca Schroeder. Rethinking WOM codes to enhance the lifetime in new SSD generations. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020.

[13] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 61–74, Santa Clara, CA, February 2014. USENIX Association.

[14] Arpith K and K. Gopinath. Need for a deeper cross-layer optimization for dense NAND SSD to improve read performance of big data applications: A case for melded pages. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020.

[15] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *2008 IEEE International Symposium on Workload Characterization*, pages 119–128. IEEE, 2008.

[16] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

[17] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 83–90, Oakland, CA, February 2018. USENIX Association.

[18] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. An In-Depth Analysis of Cloud Block Storage Workloads in Large-Scale Production. In *IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020*, pages 37–47. IEEE, 2020.

[19] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. A study of SSD reliability in large scale enterprise storage deployments. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 137–149, Santa Clara, CA, February 2020. USENIX Association.

[20] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. Operational characteristics of SSDs in enterprise storage systems: A Large-Scale field study. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, Santa Clara, CA, February 2022. USENIX Association.

[21] Fabio Margaglia, Gala Yadgar, Eitan Yaakobi, Yue Li, Assaf Schuster, and Andre Brinkmann. The devil is in the details: Implementing flash page reuse with WOM codes. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 95–109, Santa Clara, CA, February 2016. USENIX Association.

[22] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):1–23, 2008.

[23] Yuqian Pan, Haichun Zhang, Mingyang Gong, and Zhenglin Liu. Process-variation effects on 3d tlc flash reliability: Characterization and mitigation scheme. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 329–334, 2020.

[24] Ronald L Rivest and Adi Shamir. How to reuse a "write-once memory. *Information and control*, 55(1-3):1–19, 1982.

[25] SNIA IOTTA Trace Repository. YCSB RocksDB SSD traces. http://iotta.snia.org/traces/28568, September 2020.

[26] Amy Tai, Andrew Kryczka, Shobhit O Kanaujia, Kyle Jamieson, Michael J Freedman, and Asaf Cidon. Who's afraid of uncorrectable bit errors? online recovery of flash errors with distributed redundancy. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 977–992, 2019.

[27] Eitan Yaakobi, Laura Grupp, Paul H Siegel, Steven Swanson, and Jack K Wolf. Characterization and error-correcting codes for tlc flash memories. In *2012 International Conference on Computing, Networking and Communications (ICNC)*, pages 486–491. IEEE.

[28] Eitan Yaakobi, Jing Ma, Laura Grupp, Paul H Siegel, Steven Swanson, and Jack K Wolf. Error characterization and coding schemes for flash memories. In *2010 IEEE Globecom Workshops*, pages 1856–1860. IEEE, 2010.

[29] Eitan Yaakobi, Alexander Yucovich, Gal Maor, and Gala Yadgar. When do wom codes improve the erasure factor in flash memories? In *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*, 2015.

[30] Gala Yadgar, Moshe Gabel, Shehbaz Jaffer, and Bianca Schroeder. Ssd-based workload characteristics and their performance implications. *ACM Trans. Storage*, 17(1), jan 2021.

[31] Gala Yadgar, Eitan Yaakobi, Fabio Margaglia, Yue Li, Alexander Yucovich, Nachum Bundak, Lior Gilon, Nir Yakovi, Assaf Schuster, and André Brinkmann. An analysis of flash page reuse with wom codes. *ACM Transactions on Storage (TOS)*, 14(1):1–39, 2018.

[32] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. Write once, get 50% free: Saving SSD erase costs using WOM codes. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 257–271, 2015.

GuardedErase: Extending SSD Lifetimes by Protecting Weak Wordlines

Duwon Hong*
Seoul National University

Myungsuk Kim*
Kyungpook National University

Geonhee Cho
Seoul National University

Dusol Lee
Seoul National University

Jihong Kim
Seoul National University

Abstract

3D NAND flash memory enables the continuous growth in the flash capacity by vertically stacking wordlines (WLs). However, as the number of WLs in a flash block increases, 3D NAND flash memory exhibits strong process variability among different WLs, which makes it difficult for an SSD to fully utilize the maximum endurance of flash blocks, thus reducing the SSD lifetime. In this paper, we propose a new system-level block erase scheme, called GuardedErase, for extending the lifetime of a 3D flash block. The key feature of GuardedErase is that when a block is erased, a WL of the block can be selectively erased by one of two erase modes, the low-stress erase mode or the normal erase mode. When a WL is erased by the low-stress erase mode, the lifetime of the WL can be significantly extended although it may not store data. By supporting two erase modes at the WL level, GuardedErase enables an FTL to exploit the new endurance-capacity trade-off relationship at the SSD level. We have implemented the GuardedErase-aware FTL, called longFTL, which extends the SSD lifetime with a negligible impact on the overall I/O performance. Experimental results using various workloads show that longFTL can improve the SSD lifetime on average by 21% over an existing FTL with little degradation on the SSD performance.

1 Introduction

3D NAND flash memory has been a key enabling technology for sustaining a continuous capacity increase in flash storage systems [1, 2]. Since the capacity of 3D NAND flash memory depends on the number of vertical wordlines (WLs) in a flash chip, as the capacity of 3D NAND flash memory increases, the number of WLs tends to increase as well. Since the size of a flash block also depends on the number of vertical WLs, one of the key characteristics of 3D NAND flash memory over 2D NAND flash memory is that the block size gets bigger as the capacity of a flash chip increases. Table 1 shows how the capacity of a single block has been changed in recent 3D NAND flash chips. For example, the block size has increased by 5.3 times in 4 years. Since the page size does not change as fast as the block size, larger blocks typically contain more WL[1].

Table 1: Changes in the page size, the number of pages per block, and the block size over time in 3D flash memory.

Year	Total Capacity (Gb)	Block size (KB)	Page size (KB)	No. of pages per block
2014 [3]	128	3072	8	384
2015 [4]	128	6144	16	384
2016 [5]	256	9216	16	576
2017 [6]	512	12288	16	768
2018 [7]	1024	16384	16	1024

Another unique characteristic of 3D NAND flash memory is that there exists strong process variability among different WLs of a block [8, 9]. Process variability occurs because of the manufacturing process of 3D NAND flash memory. Since 3D NAND flash memory is manufactured using a vertically successive etching process from the topmost WL to the bottom WL, the cell structure of WLs may significantly vary along the z-axis, leading to significant WL-to-WL variations. As the number of WLs in a block increases, process variability among different WLs increases as well. Figure 1 illustrates that there exist significant variations in the maximum number of P/E cycles among different WLs. For example, the maximum number of P/E cycles of the weakest WL (i.e., WL *w*) is 46.3% smaller than that of the strongest WL (i.e., WL *b*).

A large reliability variation among different WLs makes it difficult to fully utilize all WLs, for example, up to the maximum endurance of each WL. Since the reliability of weak WLs deteriorates faster than that of strong WLs, the lifetime of a block is decided by the reliability characteristics of the weakest WL in the block. When the weakest WL reaches its maximum bit error rate (BER) value, the whole block becomes a *bad* block. This type of coarse-grained bad block management (BBM) is commonly used in SSDs [10]. Although the block is classified as bad under coarse-grained

*The first two authors contributed equally to this research.

[1]Since a single WL can contain multiple pages, the exact number of pages varies depending on the type of flash memory.

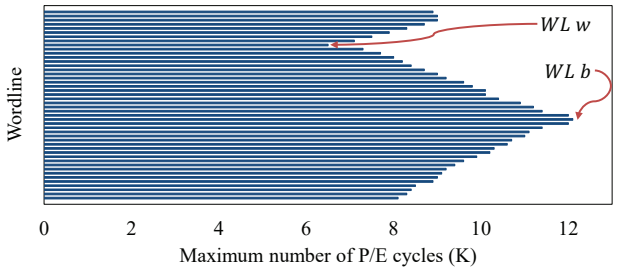


Figure 1: Per-WL variations on the maximum number of P/E cycles.

BBM, all the WLs in the block except for the weakest WL are still usable, thus significantly underutilizing WLs in the block. For example, in Figure 1, when WL *w* reaches its maximum P/E cycles (i.e., about 6,500 P/E cycles), the whole block becomes bad although WL *b* can reliably store data at least for 5,600 more P/E cycles. In order to overcome the shortcomings of coarse-grained BBM, we can manage bad *pages* instead of bad *blocks* so that the good WLs of a bad block can continue to be used. This type of fine-grained scheme is called bad page management (BPM) [11]. Although simple and somewhat effective in extending the SSD lifetime, BPM techniques, which inevitably reduce the total SSD capacity at the end of its lifetime, significantly degrade I/O performance.

A more intelligent solution to mitigate a reliability imbalance between the weakest WL and the rest of WLs would be to protect the weakest WL from degrading its reliability too early. For example, the program relief technique, which was proposed for 2D NAND flash memory [12], tries to reduce the wear stress of weak WLs. When the program relief technique is used, weak WLs are skipped from program operations or are programmed using a less-stressful program method (e.g., SLC programs instead of MLC/TLC programs). However, since erase operations, not program operations, are the main source of NAND flash wear-out [13, 14], the program relief technique is quite limited in extending SSD lifetimes. In our 3D flash characterization study (see Section 3), we confirmed that most flash wear-out comes from an erase operation, not from a program operation in 3D NAND flash memory. In order to effectively erase a large 3D flash block, a higher erase voltage is used for a longer period of time during an erase operation, thus significantly increasing the impact of erase operations on 3D flash over 2D flash wear-out. Our experimental observation strongly suggests a need for a new erase-centric flash stress-relief technique. In this paper, we propose such a stress-relief technique that focuses on erase operations.

As an effective stress-relief solution for 3D NAND flash memory, we propose a new block erase scheme, called GuardedErase (or gErase), that delays weak WLs from reaching their maximum endurance level so that the lifetime of a block can be extended. In order to extend the lifetime of weak WLs, GuardedErase employs two erase modes, normal erase mode and low-stress erase mode, at the WL level. When a WL is erased by the low-stress erase mode, the WL experiences reduced wear stress from a block erase operation, thus effectively increasing its maximum number of P/E cycles. For example, assume that a WL reaches its endurance limit with 10 normal erase operations. In this case, if the low-stress erase mode were used for the remaining erase operations, the WL will take more than 10 P/E cycles before the WL becomes bad. If the wear stress of the low-stress erase mode is just 50% of that of the normal erase mode, it takes 20 more P/E cycles before the WL becomes bad.

On the other hand, when the WL is erased by the low-stress erase mode, the WL is not fully erased so that it cannot reliably store data. That is, when the WL is erased by the low-stress erase mode, the effective capacity of the block is temporarily reduced by the capacity of the WL. Therefore, the key technical challenge of applying GuardedErase at the SSD level is how to efficiently extend the block lifetime using the low-stress erase mode while not degrading I/O performance from the reduced block capacity. More formally, assume that the lifetime L_S of an SSD S is given by the total amount of written data in terabytes, TBW_S, which can be expressed by the following equation:

$$TBW_S = \frac{C_S \times MAX_{P/E}}{WAF}, \quad (1)$$

where C_S is the capacity of the SSD S, $MAX_{P/E}$ is the maximum number of P/E cycles, and WAF is a total write amplification factor of an FTL including garbage collection (GC) and wear leveling. When GuardedErase is used carelessly, it will increase both $MAX_{P/E}$ and WAF, thus limiting its effectiveness in increasing TBW_S.

The low-stress erase mode of GuardedErase focuses on reducing an erase voltage applied to a WL because the erase voltage is the dominant factor affecting the flash wear-out status. In our prototype implementation, we exploited a custom flash test command [15] that allows to change various NAND flash parameters including the erase voltage level of each WL. Using the low-stress erase mode, we performed an extensive characterization study for understanding the reliability impact of the low-stress erase mode. We observed that the wear-stress of a WL under the low-stress erase mode (as implemented in the prototype) was about 35% of that under the normal erase mode. Based on the per-WL low-stress erase mode, we built a NAND endurance model for gErase-enabled flash blocks. Our NAND endurance model supports nine different low-stress block erase modes that differ in their endurance improvement ratios and block capacity reduction factors.

Exploiting the endurance-capacity trade-off of the proposed NAND endurance model, we have implemented the gErase-aware FTL, called longFTL, which dynamically changes the number of WLs erased by the low-stress erase mode while minimizing the I/O performance degradation from the SSD capacity reduction from the low-stress erase mode. LongFTL includes several gErase-specific FTL modules (such as the

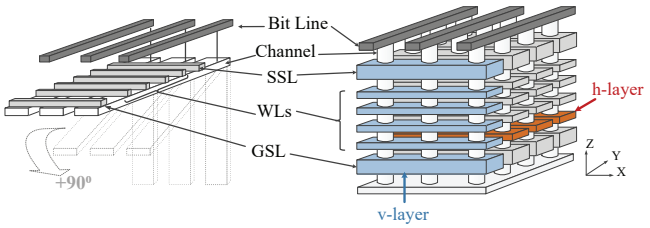


Figure 2: Differences between 2D NAND flash and 3D NAND flash [9].

weak WL monitor and the gErase mode selector) to achieve two conflicting goals, extending SSD lifetimes and maintaining SSD performance. We evaluated the effectiveness of longFTL with the MQSim simulation environment [16] with a custom extension for GuardedErase. Experimental results using various I/O traces show that longFTL can improve the SSD lifetime an average by 21% over an existing gErase-unaware FTL with an average 3% decrease in the overall I/O performance.

The rest of this paper is organized as follows. We review an overview of 3D NAND flash memory and explain how reliability is managed in 3D NAND flash in Section 2. In Section 3, we present key reliability characteristics of 3D NAND flash blocks. The proposed GuardedErase scheme is described in Section 4 and longFTL is covered in Section 5. Sections 6 and 7 describe our evaluation results and related work, respectively. We conclude in Section 8 with a summary and future work.

2 Background

2.1 Overview of 3D NAND Flash Memory

3D NAND flash memory [17] enables the continuous growth in the flash capacity by vertically stacking the memory cell to overcome various technical challenges in scaling 2D NAND flash memory. By exploiting the vertical dimension for higher capacity (instead of focusing on finer process technologies in 2D flash memory), 3D NAND flash memory has been successful in increasing its capacity by 50% annually while avoiding reliability degradation [18].

Figure 2 shows the organizational difference in a flash block[2] between 2D and 3D NAND flash memory. In this example, the 2D NAND flash memory has a matrix structure in which four WLs and three bitlines (BLs) intersect at 90 degrees. On the contrary, the 3D NAND flash memory has a *cube-like* structure. The 3D NAND flash block consists of four vertical layers (v-layers) in the y-axis where each v-layer has four vertically stacked WLs that are separated by select-line (SSL) transistors. As shown in Figure 2, when the 2D NAND flash block is rotated by 90° in a counterclockwise direction using the x-axis as an axis of rotation (i.e., if the WLs are set vertically), it corresponds to a single v-layer. Similarly, the 3D NAND flash block can be described to have four horizontal layers (h-layers) which are stacked along the z-axis, and each horizontal layer consists of four WLs.

In order to increase the capacity of a 3D flash chip, the most effective approach is to increase the number of h-layers in 3D NAND flash memory (i.e., stacking more h-layers along the z-axis). As the number of h-layers increases, the block size increases as well (as summarized in Table 1).

2.2 Reliability Management in NAND Flash Memory

In order to reliably store data in flash cells, various reliability requirements should be satisfied. For example, flash blocks should not be used more than their limited maximum P/E cycles. This is because the NAND cell characteristics in the flash block deteriorate as the number of P/E cycles increases. The main cause of wear-out of a flash block is electrical stress on the tunnel oxide during the program and erase operations. As program/erase operations are repeatedly performed on the block, the amount of charge trapped in the tunnel oxide layer increases. The trapped charges make it difficult for the threshold voltage level (i.e., state) of the erased NAND cells to be located within their intended voltage interval, thus significantly affecting the data retention characteristics of NAND cells in the block. [19]. Therefore, as the P/E cycle increases, the BER characteristics of data stored in the flash block deteriorate.

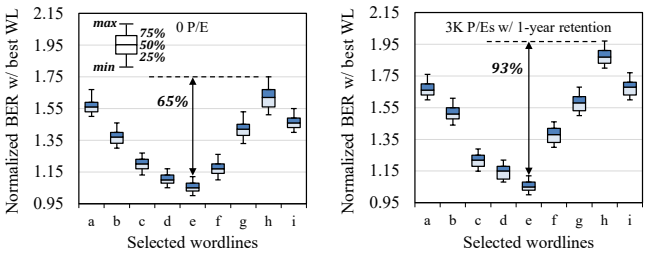
To prevent the number of error bits from surpassing the correction capacity of an error correction code (ECC) engine, NAND manufacturers set the maximum number of P/E cycles allowed for a block, which is called as NAND endurance. If the block continues to be used over the maximum number of P/E cycles, the BER of the block will exceed the ECC capability, which results in data loss (i.e., a read error). In addition to errors due to NAND wear-out, various errors (such as program errors and erase errors) can occur due to process defects during a NAND flash manufacturing procedure [20].

Although NAND operations can fail for different reasons, failed operations are managed in the block granularity within an FTL. For example, if a read operation to a page in a block B fails, the FTL identifies the block B as a bad block and replaces B with a reserved block. After the BBM module of the FTL moves all the valid data from the bad block B to the reserved block, the bad block B is no longer used.

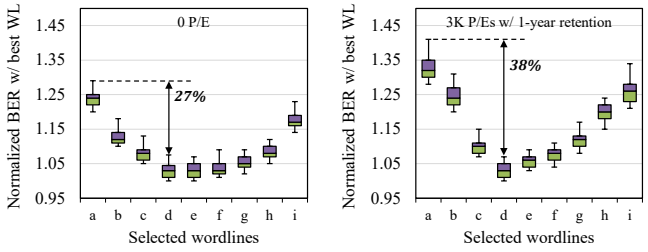
3 Reliability Characterization of 3D Blocks

In this section, we explain two key observations on 3D NAND flash characteristics which motivated the proposed GuardedErase scheme.

[2] NAND flash memory consists of multiple blocks. Each block has multiple WLs (e.g., 128 - 256 WLs) and each WL consists of a group of flash cells (e.g., 8K - 16K cells).



(a) Per-WL BER of 3D flash. (b) Per-WL BER of 3D flash.



(c) Per-WL BER of 2D flash. (d) Per-WL BER of 2D flash.

Figure 3: Per-WL BER variations in 3D NAND flash and 2D NAND flash.

3.1 Large Reliability Variations Among WLs

Ideally, we would expect all flash cells in a block (or chip) to have identical characteristics. However, in practice, significant electrical/physical characteristic variations between flash cells are unavoidable due to many unexpected process fluctuations in a complex NAND flash manufacturing system. For example, in 2D NAND flash memory, it is known that the reliability of WLs varies depending on the physical location of a WL within a flash block. Furthermore, it is widely accepted that such process variability would be much stronger in 3D NAND flash memory because of its unique manufacturing process. To quantify the reliability variations between WLs, we examined the inter-WL BER variations using 160 real 3D TLC NAND flash chips[3]

Figure 3(a) shows significant inter-WL BER variations exist within a tested block even when the blocks experience no program/erase cycles. All BER values were normalized over that of the most reliable WL. (For simplicity, we show the evaluation result of the selected 9 WLs.) The BER of the worst WL (WL h, which is near the top layer) is about 60% higher than that of the best WL (WL e, which is around the middle layer). When flash blocks get aged (from a large number of P/E cycles), as shown in Figure 3(b), the BER of the worst WL may be twice as large as that of the best WL. We

[3]Our flash chips are fabricated by 3D charge-trap technology (which is known as *SMArT* [21] or *TCAT* [22]). All the commercial 3D NAND flash memories have similar structures and cell types, and it is commonly believed that different 3D TLC flash chips share key device-level characteristics. Therefore, our characterization results on reliability variability agree with general trends with other 3D NAND flash chips.

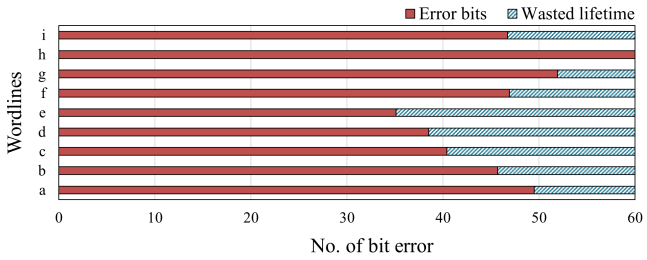


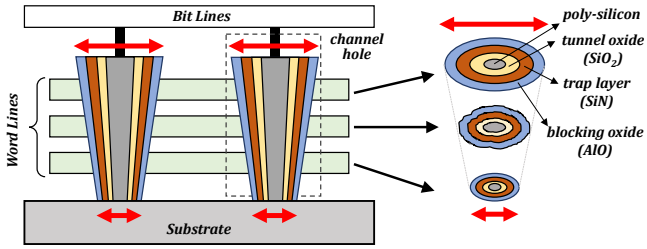
Figure 4: Inter-WL BER variations when the worst WL becomes bad.

also examined the inter-WL reliability variations of 1x-nm 2D TLC NAND flash memory. Unlike 3D NAND flash memory, as shown in Figures 3(c) and 3(d), the BER difference between the worst WL and best WL is much smaller.

In order to understand the impact of large reliability variations among WLs on the NAND block lifetime, we examined per-WL BER variations of different WLs when the number of bit errors from the worst WL exceeds the maximum ECC correction capacity. Figure 4 shows that when the number of bit errors from the worst WL, WL h, exceeds the ECC correction capacity, 60 bits per 1-KB sector, the rest of the WLs in the block are still fairly reliable. If we decide that the block is bad at this time, a significant amount of the block lifetime is wasted. For example, the BER of the best WL, WL d, is just about 60% of that of the worst WL, WL h. Therefore, to fully utilize the lifetime of 3D NAND flash memory, it is important to manage the reliability of a flash block at the individual WL level instead of the conventional block level because there exist large reliability variations among WLs in a flash block.

The root cause of large reliability variations is related to a unique manufacturing process to form the vertical architecture of 3D NAND flash memory. Figure 5(a) shows a detailed organization of a vertical layer in 3D NAND flash memory using a cross-sectional view along the y-z plane and a top-down view (of three cross sections along the x-y plane). The channel holes are formed at the early stage of 3D NAND flash manufacturing by an etching process [23].

Under an ideal manufacturing process, all the channel holes would have the same geometrical structure regardless of their physical locations to achieve the homogeneous reliability



(a) A cross-sectional view. (b) A top-down view.

Figure 5: Inter-WL variations in 3D NAND flash memory.

characteristics among flash cells. However, the cylindrical channel hole cannot avoid suffering from severe structural variations depending on its vertical (z-directional) location. For example, as shown in Figure 5(b), the diameter of the channel holes varies significantly over the height of an h-layer. Furthermore, the shape of channel holes is also affected depending on vertical positions. Different channel hole diameters as well as their shapes can cause large variations in the characteristics of flash cells. Therefore, even when the same program/erase voltage is applied to flash cells, they experience different electrical stress depending on the diameter or shape of a channel hole, resulting in different reliability characteristics along vertical locations.

3.2 Erase Stress on Flash Reliability

Before we design a WL-level stress mitigation technique for 3D NAND flash memory, in order to quantitatively understand the main source of flash stress, we performed a comprehensive characterization study using real 160 3D TLC NAND flash chips with 48 horizontal layers where each layer consists of 4 WLs. Since both a program operation and an erase operation incur a significant amount of wear stress on flash cells, the main goal of our study was to measure the relative impact of these operations on the reliability characteristics of 3D flash cells. We tested a total of 3,686,400 WLs (11,059,200 pages) to obtain statistically significant experimental results. Following a standard evaluation metric commonly used in NAND flash reliability studies, we measured changes in RBER (Raw Bit-Error Rate) after each measurement scenario.

To quantify the impact of the program and erase operations on the NAND endurance stress, we designed three measurement scenarios. In each scenario, one of three operation sequences is repeated until the lifetime limit of the block. Table 2 summarizes three operation sequences with their member operations. Note that all three sequences include dummy operations so that unexpected factors do not affect the accuracy of measurement results.[4]

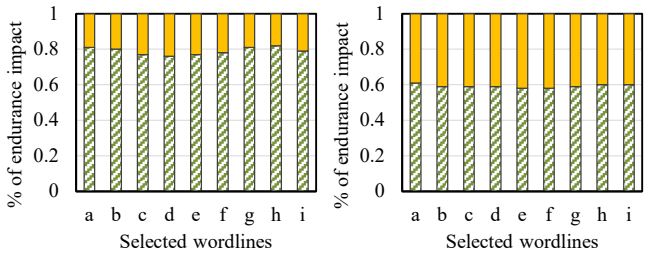
Figures 6(a) and 6(b) show how erase and write operations affect the lifetime of a flash block in 2D and 3D NAND flash memory, respectively. As expected, the endurance impact of an erase operation was significantly larger than that of a program operation in 3D NAND flash memory. The endurance stress of erase operations was responsible for almost 80% of the total stress of flash cells in 3D NAND flash memory. Note

[4] As explained in Section 2.2, the flash cell's wear-out is mainly caused by trapped charges in the tunnel oxide layer during program and erase operations. If the flash cell is sufficiently erased, there is little charge remaining that can be transferred from the SiN layer to the substrate. Therefore, for example, repeating only erase operations in `Erase-only Sequence` cannot accurately measure the endurance effect of the erase operation. As shown in Table 2, dummy program operations are performed before erase operations. Dummy erase operations were added to the sequence because they were included in `Modified Base Sequence` so that the effect of erase operation can be effectively measured by comparing BER values of two sequences.

Table 2: A summary of three operation sequences.

Sequence type	Operations order
`Modified Base Sequence`	P → dummy P → E → dummy E
`Erase-only Sequence`	dummy P → E → dummy E
`Program-only Sequence`	dummy E → P → dummy P

P : program, E : erase



(a) 3D NAND flash. (b) 2D NAND flash.

Figure 6: Endurance impact of erase and program operations.

that the impact of erase operations on the flash endurance is about 33% higher in 3D NAND flash memory over 2D NAND flash memory. The high impact of erase operations over program operations was observed similarly regardless of WL locations. Our measurement study clearly indicates that a low-stress mechanism should be focused on erase operations, not program operations.

3.3 Erase Stress Reduction

As described in Section 2.2, electrical stress during P/E cycles could have a detrimental effect on the tunnel oxide layer of flash cells. As the amount of damage to the tunnel oxide layer increases, flash cells eventually get worn out. As shown in Section 3.2, lowering the erase stress is essential in reducing the amount of damage to the tunnel oxide layer. Since the amount of damage to the tunnel oxide layer increases exponentially as the erase voltage increases [24], the proposed GuardedErase scheme offers a low-stress erase mode by lowering erase voltage.

4 GuardedErase: Mechanism and Modes

4.1 Basic Idea

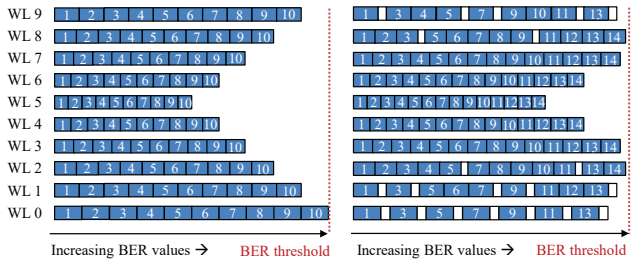
Since an erase operation is a major source of flash wear-out, it is important to reduce the erase stress on flash cells when they are erased if the flash cells can be used for more P/E cycles. In order to reduce the erase stress when necessary, GuardedErase supports the low-stress erase mode as well as the normal erase mode. The key motivation behind GuardedErase is that when a WL is erased by a lower erase voltage, the lifetime of the WL increases [14]. GuardedErase protects weak WLs from

experiencing high erase stress by lowering their erase voltage, thus extending the maximum number of block erasures for a block (i.e., the lifetime of a flash block).

Consider an example 3D flash block with 10 WLs in Figure 7(a). Reflecting strong process variability among WLs, there are significant variations on BER values for the same number of P/E cycles. For example, when the normal block erase operation is used, WL 0, which is the weakest WL, reaches its maximum BER value (i.e., the BER threshold value of the block) after 10 P/E cycles. On the other hand, after 10 P/E cycles, WL 5, which is the strongest WL, barely reached 50% of its maximum BER value. However, since WL 0 reached its maximum BER value, the block is no longer usable, thus becoming a bad block. If we employ a fine-grained BPM policy, the block can continue to be used with nine WLs after 10 P/E cycles. However, since the effective block capacity is reduced by 10% (which, in turn, may introduce a severe WAF increase), the BPM policy is rather limited in increasing the total amount of written data to the block while incurring no I/O performance degradation. In the example block in Figure 7(a), even if an SSD can tolerate up to 20% of the average block capacity reduction (thanks to its OP space), only 9 more WLs can be written to the block before WL 1 becomes unusable after one more block erasure.

Figure 7(b) illustrates how the lifetime of the example block can be extended using the proposed GuardedErase scheme. When BER values of weak WLs (such as WL 0, WL 1 and WL 9) are higher than those of other WLs, weak WLs are erased using the low-stress erase mode that reduces wear by approximately 1/3 as will be shown in Section 4.2. For example, after the 1st erase cycle, the BER value of WL 0 is more than double that of WL 5. In order to protect WL 0, the low-stress erase mode, which is indicated by a white box in Figure 7(b), is used for WL 0 in the 2nd erase cycle. WL 0 is erased six more times using the low-stress erase mode under similar conditions. Figure 7(b) shows that the lifetime of WL 1 and WL 9 is extended to the 14-th erase cycle with five applications of the low-stress erase mode while that of WL 2 and WL 8 is extended to the 14-th erase cycle with two applications of the low-stress erase mode.

When weak WLs are erased using the low-stress erase mode, they cannot store data for the following program cycle (i.e., it cannot reliably store data). In Figure 7(b), WL 0 is not used to store data 7 times out of 14 block erasures. (White boxes indicate the unstable WL state.) Although the total amount of written data to WL 0 is significantly reduced, the lower erase voltage prolongs the lifetime of WL 0, which is the weakest WL of the block. By delaying WL 0 to reach its BER threshold, we can increase the lifetime of the block to 14 block erasures, thus more data are written to the block. In Figure 7(b), the total amount of written data (in the number of WLs) increases from 100 WLs (in Figure 7(a)) to 119 WLs. That is, by using the lower erase voltage for weak WLs, we were able to increase the total amount of data written to a



(a) Using the normal erase mode. (b) Using the gErase mode.

Figure 7: Per-WL BER changes under the normal block erase and the gErase block erase.

block by 19%. This, in turn, extends the SSD lifetime in terms of the total amount of written bytes.

Note that the GuardedErase scheme outperforms the BPM scheme in the total number of WLs written to the block by writing 10 more WLs. Furthermore, unlike the BPM scheme where the effective block capacity is monotonically non-increasing over P/E cycles, the GuardedErase scheme can dynamically control the effective block capacity up to the maximum block capacity depending on which erase mode is used. If the low-stress erase mode can be adaptively applied by exploiting the future write demand and intensity characteristics, the GuardedErase scheme can efficiently extend the SSD lifetime without an I/O performance degradation. In order to achieve the full potential of the GuardedErase scheme, therefore, we need to design an efficient mechanism for supporting the low-stress erase mode and devise an intelligent management policy of applying the low-stress erase mode to proper WLs at the right time.

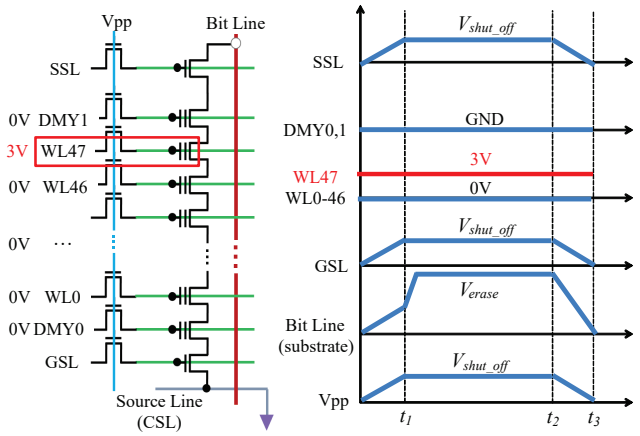
4.2 Per-WL Low-Stress Erase Mode

4.2.1 Implementation

There are two implementation options for the low-stress erase mode. One is to reduce the erase time (i.e., erase time scaling [14]) and the other is to reduce the erase voltage (i.e., erase voltage scaling [14]). However, since it is not easy to control the erase time at the WL granularity, we employed a scheme that lowers the erase voltage. In order to reduce the erase voltage for a specific WL, we exploited a test-mode command [15] for 3D NAND flash memory that allows to vary the driving voltage setting at the WL granularity. Although we cannot directly apply a lower erase voltage to a specific WL, this test-mode command can be used to effectively lower the erase voltage for the target WL[5].

Figure 8(a) illustrates how to reduce the erase stress on WL 47 using the proposed method. Since the voltage difference between the control gate voltage and the voltage applied to

[5]It has been known that all NAND manufactures have their own hidden test-mode commands to modify the internal operating voltage.



(a) Differential voltage driving to control gates.

(b) Voltage changes over time for a block erase.

Figure 8: An implementation of the low-stress erase mode.

the substrate acts as the effective erase voltage, when a higher voltage is applied to the control gate of a WL, its erase voltage is effectively reduced, thus reducing the erase stress on the WL. In Figure 8(a), 3V, not the usual 0V, is applied to the control gate of WL 47, thus reducing the effective erase voltage of WL 47 by 3V. Figure 8(b) shows how various voltages are driven when the low-stress erase mode is used for WL 47 while the rest of WLs are erased using the normal erase mode. When the nominal erase voltage V_{erase} is applied to the bit line (or substrate) from time t_1 to t_2, the NAND flash cells of WL 47 experience a smaller electrical potential difference by 3V over the flash cells of the other WLs. Considering that V_{erase} is approximately $17V$ and the erase stress is exponentially proportional to an electrical potential difference, a significant amount of erase stress is reduced to the NAND flash cells of WL 47 during an erase operation.[6]

4.2.2 Stress Mitigation Effect

In order to understand the endurance impact of the low-stress erase mode on WLs, we performed a comprehensive process characterization study using state-of-the-art 3D TLC NAND flash chips. For endurance comparisons, we formed two block groups whose member blocks were evenly selected from different physical locations using 30 NAND flash chips. Two block groups were erased using different erase modes, one group using the normal erase mode only and the other group using the normal erase mode and the low-stress erase mode half and half,[7] respectively. For each WL of a tested block,

[6] When the low-stress erase mode is applied to a WL, we increase the verify voltage setting for the WL so that the erase operation can take the same number of erase loops as in the normal erase mode.

[7] We experimented with different combinations of two erase modes (e.g., 25% : 75%). However, we could not find any difference in the experimental conclusions.

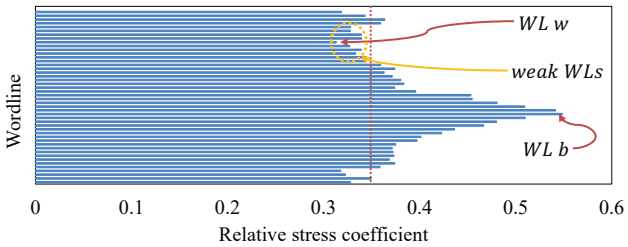


Figure 9: Per-WL relative stress coefficients.

we collected the maximum number of P/E cycles that keeps the BER value of the WL below the BER threshold value.

Based on the measured maximum P/E cycles per WL, we computed the relative stress coefficient S_k of WL_K that indicates how much the erase stress is mitigated when the low-stress erase mode is used. S_k is a per-WL quantity that indicates how much WL_K wears out when it is erased using the low-stress erase mode over the normal erase mode. For example, if the maximum number of P/E cycles of WL_P was 10K when the normal erase mode was used, but it was increased to 20K when the low-stress erase mode is used, the relative stress coefficient S_p is 0.5. Figure 9 summarizes our measured relative stress coefficients for different WLs in a block.

Although there are significant differences in S_k values depending on WLs, we observed that the relative stress coefficients of weak WLs (for which most of the low-stress erase mode are applied) are quite similar: all the coefficients belong to an interval [0.31, 0.34] with the mean of 0.33. Although the low-stress erase mode may be applied to a few strong WLs with larger relative stress coefficients in GuardedErase, when we evaluate the stress mitigation effect of the low-stress erase mode, we assume that all the WLs have the same relative stress coefficient, 0.35, for a simple analysis.[8]

4.3 Per-Block Erase Modes

In order for an FTL to effectively exploit the endurance-capacity trade-off of the low-stress erase mode, we support nine block erase modes, gE(1), ..., gE(9). Table 3 summarizes the proposed nine block erase modes with varying numbers of protected WLs and different erase relief ratios[9]. The higher n in gE(n), the more WLs are erased using the low-stress erase mode with a higher erase relief ratio. Therefore, when a block is erased with a higher gErase mode, the maximum number of P/E cycles of the block is increased. For example, when a block is erased with gE(9) only, the maximum number of P/E

[8] In GuardedErase, a fixed number N of WLs are selected for applying the low-stress erase mode. Although N is fixed, selected N WLs change over time because N WLs with the worst BER values are protected when a block is erased. Since the relative stress coefficient of selected WLs is mostly less than 0.35 with few stronger WLs, our assumption of 0.35 is rather conservative in understanding real stress mitigation levels.

[9] The erase relief ratio of gE(n) is defined as a ratio of applying the gE(n) block erase mode over the normal block erase operation.

Table 3: A summary of nine gErase modes.

	gErase mode(n) == gE(n)								
	gE(1)	gE(2)	gE(3)	gE(4)	gE(5)	gE(6)	gE(7)	gE(8)	gE(9)
No. of protected WLs	8	12	16	20	24	24	24	28	32
Erase relief ratio	25%	33%	38%	40%	42%	50%	50%	50%	50%
Block capacity reduction	1.04%	2.08%	3.13%	4.17%	5.21%	6.25%	7.29%	8.33%	9.38%
Norm. Max P/E cycles	1.19	1.26	1.30	1.33	1.37%	1.39	1.41	1.43	1.45

cycles of the block is increased by 45% whereas when the block is erased with gE(1), it is increased by 19% only. On the other hand, as shown in Table 3, the higher the gErase mode, the more WLs become unusable for the next program cycle, thus increasing WAF values. As described in Section 5, it is an important task for an FTL to choose a proper gErase mode when it erases a block.

In the remainder of Section 4.3, we provide a high-level description on how we designed nine block erase modes. (For additional details, see [25].) In Table 3, the percentage of block capacity reduction in gE(n) is given by (1.04 × n)%. Since our flash block has 192 WLs, when a block is erased by gE(n), $2n$ WLs of the block become unusable for the next program cycle[10]. When a target percentage of block capacity reduction, say 1.04% in gE(1), is given, there can be multiple options to achieve the target block capacity reduction ratio using the low-stress erase mode. Let n_{WL} and f denote the number of protected WLs and the erase relief ratio. A combination $(k \times n_{WL}, f/k)$ over any k achieves the same block capacity reduction ratio as when $(n_{WL} \times f/100)$ WLs are always protected. For example, (4 WLs, 50%), (8 WLs, 25%) and (12 WLs, 16.7%) can achieve the same 1.04% reduction percentage in a 192-WL blocks. From multiple candidate combinations, we prefer ones with a lower f because such a combination can achieve higher I/O efficiency by allowing more flexible applications of gErase modes by an FTL [25]. For example, we prefer (8 WLs, 25%) and (12 WLs, 16.7%) over (4 WLs, 50%). On the other hand, if f is too low (i.e., n_{WL} is too large), weak WLs may not be fully protected, thus limiting their endurance improvements [25]. For example, (12 WLs, 16.7%) cannot maximally extend the lifetime of the weakest WL, WL 0, because f should be larger than 18.3% if WL 0 could fully achieve its maximum expected endurance [25]. Therefore, as shown in Table 3, (8 WLs, 25%) was selected for gE(1).

5 Design and Implementation of LongFTL

5.1 Overview

Based on the proposed gErase modes of Table 3, we implemented a gErase-enabled FTL, called longFTL, which exploits the key trade-off relationship of gErase between the block endurance extension and the block capacity reduction. The main goal of longFTL is to significantly extend the block lifetime with a negligible performance degradation by intelligently choosing a gErase mode under varying I/O workloads. Figure 10 shows an overall organization of longFTL. LongFTL, which is based on a typical page-level mapping FTL, employs three gErase-specific modules, the weak WL detector, the WAF monitor, and the gErase mode selector. The weak WL detector dynamically identifies weak WLs in a block, the WAF monitor tracks WAF changes, and the gErase mode selector determines the optimal gErase mode for the current workload characteristics estimated by the WAF monitor. The extended flash block manager selectively applies the optimal gErase mode in producing free blocks during its free-block allocation step.

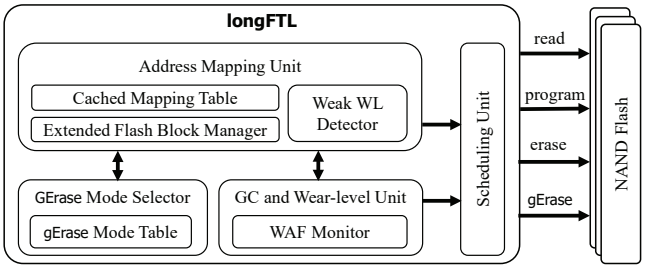


Figure 10: An organizational overview of longFTL.

5.2 Weak WL Detector

In order to protect weak WLs to extend the endurance of a block, it is required to identify which WLs are weak. [11] The weak WL detector (WLD) module is responsible for maintaining WLs according to their BER values so that when N weakest WLs are requested by the gErase mode selector, the WLs with the N largest BER values can be quickly identified. The WLD module employs $(N_{ecc}^{max} - 9)$ linked lists where N_{ecc}^{max} represents the maximum number of bit errors that can be corrected by an ECC module. A linked list L_k contains all the WLs with k bit errors. Figure 11 shows an example of BER-sorted linked lists used for detecting weak WLs.

The BER-sorted linked list is used temporarily when weak WLs are identified. (In the current implementation, we identify weak WLs once in 100 P/E cycles.) We maintain the identified weak WLs using a separate bitmap data structure. For each WL, we allocate a 1-bit flag that indicates the WL is weak or not. Using this bitmap information, longFTL recognizes which WLs should be erased by the low-stress erase mode. The memory footprint for supporting the bitmap of weak WLs is small. For example, if a NAND die consists of 2000 blocks and there are 192 WLs per block, the required memory is less than 48 KB, which is less than 0.3% of the memory requirement of a page-level mapping table.

[10] (2/192) × n ≃ 0.0104 × n.

[11] Since each WL may experience different erase modes (i.e., normal and 9 gErase modes), we maintain a 16-bit relief count for each WL that indicates the current wear status of the WL. However, for a simpler presentation, we assume that each WL is managed based on their BER value. The memory overhead of supporting per-WL relief count can be limited to about 2% of the logical-to-physical mapping table size.

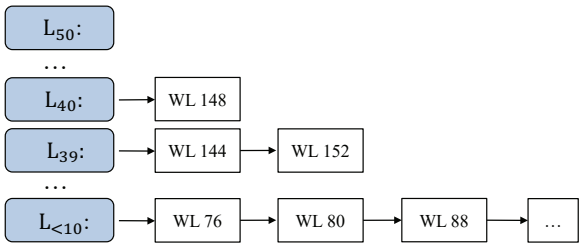


Figure 11: BER-sorted linked lists for detecting weak WLs.

To identify weak WLs of a block, we construct the BER-sorted linked list for the block after every 100 P/E cycles. Since we use the BER-sorted linked list only when weak WLs are identified, the BER-sorted linked list for the block is allocated temporarily from a heap memory and returned after use. That is, the memory requirement for one BER-sorted linked list is sufficient for a whole NAND die with 2000 blocks. Assuming that N_{ecc}^{max} is 50, the maximum memory requirement is less than 2 KB per NAND die when each block has 192 WLs. For example, assuming 4 bytes for a list pointer and 2 bytes for WL information, (50 - 8) headers $\times$ 4 bytes/header + 192 WLs $\times$ (4+2) bytes/WL = 1320 bytes.

Since weak WLs of a block are identified every 100 P/E cycles of the block, its time overhead is minimal. For the current implementation, we check the BER value of a WL by reading its worst page (e.g., MSB page). The total overhead of checking BER values of WLs is typically less than 0.1% of total I/O time in our evaluation. Furthermore, we can distribute the overhead of the BER checking step of each block to multiple P/E cycles (e.g., 91-st to 100-th P/E cycles) by overlapping the weak WL identification step among multiple blocks (e.g., 10 blocks). Since this requires ten BER lists at the same time, the memory requirement increases to about 13 KB (which is less than 0.1% of the page-level mapping table size). However, we only need to check BER values from one-tenth of the total WLs during one P/E cycle interval, thus further reducing the time overhead. Since the BER value of a WL changes slowly, the distributed scheme does not affect the accuracy of identifying weak WLs.

In addition to selecting N weakest WLs from a block, the WLD module ensures that gErase modes are evenly applied to all the blocks. If a large number of gErase modes are used for a few blocks only, the SSD lifetime may not be extended at all because the rest of blocks are erased using the normal erase mode only. We maintain a separate linked list of free blocks by the number of gErase operations. When we select a free block, we prefer blocks with fewer gErase counts[12].

5.3 WAF Monitor

The WAF monitor is responsible for tracking a WAF value of an SSD. Since a block capacity is reduced when the block is erased using a gErase mode, it is important to understand if the current effective SSD capacity is adequate to properly process the current I/O workload. If the effective SSD capacity were reduced too much from aggressive gErase mode applications by the gErase mode selector, the SSD may suffer a significant I/O performance degradation. On the other hand, if gErase modes were applied too conservatively, the effectiveness of GuardedErase is quite limited. In order to prevent the I/O performance fluctuations from frequent gErase-mode changes, we modify gErase modes only when the current I/O workload has been relatively stable so that transient I/O workload variations do not cause an SSD capacity reduction from mispredicted mode changes.

The main function of the WAF monitor is to observe WAF fluctuations, which we interpret as I/O workload changes, and decide if the current WAF value is *stable* enough for the gErase mode selector to make a proper mode decision. Although it is difficult to precisely define what a stable WAF value means, in the current implementation, we assume that a WAF value is stable if the last 9 measured WAF, as well as the current WAF, have been *steady* in their respective observation intervals. A WAF value is called *steady* for an observation interval $(t_s, t_e]$ if all the WAF values observed in $(t_s, t_e]$ are within $[0.98w, 1.02w]$ where w is the WAF value at t_s. Since we are interested in knowing long-term workload characteristics instead of fast changing short-term workload characteristics, a single observation interval is defined as the time it takes to perform a sufficient number (e.g., 5% of total blocks) of garbage collections (GC). The WAF monitor also tracks the change in WAF values (i.e., WAF history) so that the recent WAF trend can be considered by the gErase mode selector.

5.4 GErase Mode Selector

The gErase mode selector decides which erase mode would be used for the next block erase. Figure 12 describes the key steps in selecting the next gErase mode.

In deciding the next erase mode, the gErase mode selector considers two pieces of information from the WAF monitor: WAF stability and WAF history. The gErase mode selector with the current gErase mode gE(cur) invokes the procedure in Figure 12 whenever a new WAF value w is measured (i.e., every observation interval). If the WAF value is stable, we compute the expected TBW values (from Equation 1) for three neighboring gErase modes, gE(cur-1), gE(cur) and gE(cur+1), and select the gErase mode with the largest TBW value as the next gErase mode to be used. If not all three TBW values are computable (i.e., because of WAF history for gE(cur+1) is not available), we set the next gErase mode as gE(cur+1), so that we can explore the potential benefit of using a more aggressive gErase mode. Note that this explorational step is tried in a limited fashion only after an I/O workload signif-

[12] Since a free block may be selected under a different criterion (such as low P/E cycles), leveling gErase counts among different blocks is supported in a combined fashion with other selection criteria.

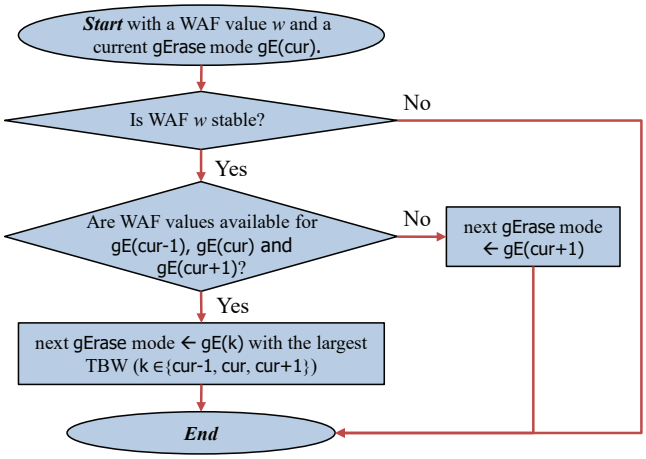


Figure 12: A procedure for selecting the next gErase mode.

icantly changes when most previous WAF values become unstable. Once a WAF value becomes unstable for gE(n), the TBW value for gE(n) cannot be computed until the WAF value get re-stabilized again.

Figure 13 illustrates how the gErase mode selector works over varying I/O workloads. The gErase mode is started with gE(0). When the WAF is saturated (❶), there is no WAF history for the adjacent gErase mode, so it is changed to gE(1) without a TBW comparison. When the WAF is saturated after the mode change (❷, ❸) a higher mode is used because TBW gets higher over the previous mode. However, the TBW of gE(3) is lower than that of gE(2), the mode changes back to gE(2). Since the newly calculated TBW of gE(2) is lower than the TBW of gE(1) calculated by WAF history (❺), it is changed back to gE(1). In the actual gE(1) execution, the TBW is rather low (❻), so the mode is changed back to gE(2). After that, the WAF remains stable, and the TBW is maximized at the mode, gE(2).

5.5 Extended Flash Block Manager

Once the next gErase mode is determined, the extended flash block manager decides whether to apply the gErase mode before generating free blocks that can be used for incoming allocation requests. In order to minimize the negative impact on the I/O performance from applying gErase modes, we take a conservative policy in employing gErase modes.

In the current implementation, the extended flash block manager uses gErased free blocks only for storing hot data from host requests. The main rational behind this policy is that the negative performance impact of the SSD capacity reduction from gErase modes would last longer if cold data (that are not likely to be updated) are allowed to be stored to gErased free blocks. Since a gErased block with cold data is less likely to be selected as a victim block of garbage collection, the unused WLs from the gErased block will take longer to be reclaimed for future usage. Therefore, the impact

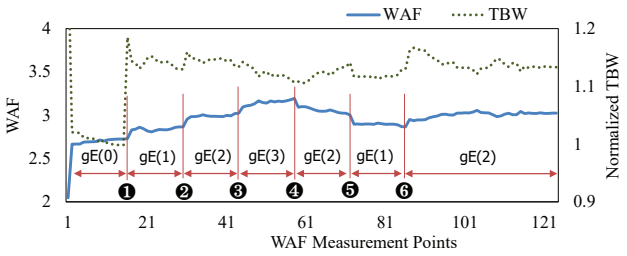


Figure 13: An example gErase mode selection.

of the SSD capacity reduction on WAF will last longer. Note that our current policy prevents gErased free blocks from storing valid data moved during a garbage collection process because they are implicitly cold data [26]. Since valid data moved during the garbage collection process are considered cold, we do not store them to gErased blocks, thus reducing I/O performance degradation.

6 Experimental Results

6.1 Experimental Settings

To evaluate the effectiveness of the proposed technique, we implemented longFTL as an extension on a well-known FTL simulator, MQSim [16]. For our evaluation, we configured the FTL simulator to support the 32-GB storage capacity for faster experiments. Our simulated storage system has four channels with one NAND flash chip per channel. In setting the key NAND flash configuration parameters, recent large-block flash chips [5, 6] were used as references. Each NAND flash chip has a 2-plane configuration and each plane has 1822 blocks. Each block consists of 192 WLs and each WL can store three 8-KB pages. The average page program time and the block erase time were set to 700 μs and 4000 μs, respectively. The overprovisioning ratio was set to 10% as commonly done in commercial SSDs. To effectively exploit the overprovisioned area, GC is invoked when the number of free blocks is less than 0.2% of the total number of blocks.

We compared longFTL with two existing schemes: Baseline, and BPM [11]. Baseline represents a standard page-level mapping FTL without any special scheme for optimizing large-block NAND flash chips. BPM is the same FTL as Baseline with the bad-page management scheme [11]. All evaluation results were normalized over those from Baseline.

We have carried out our experiments with various I/O traces with different I/O characteristics from MSR trace workloads [27] and synthetic I/O traces generated from Sysbench [28] and Filebench [29]. Table 4 summarizes key I/O characteristics of these workloads. Since longFTL improves its lifetime at the expense of temporarily reduced SSD capacity, we selected I/O traces (e.g., prxy0 and src10) with a large amount of written data so that we can better understand how longFTL works under reduced SSD capacity scenarios. When the total amount

Table 4: I/O characteristics of traces used for evaluations.

	proj0	prxy1	prxy0	src10	proj2	OLTP	fileserver	varmail
Read:Write	6:94	64:36	5:95	46:54	86:14	70:30	40:60	40:60
Total writes (GB)	144	725	54	302	169	639	249	312
WAF (without gErase)	1.08	1.16	1.24	2.66	3.55	1.89	2.49	2.98

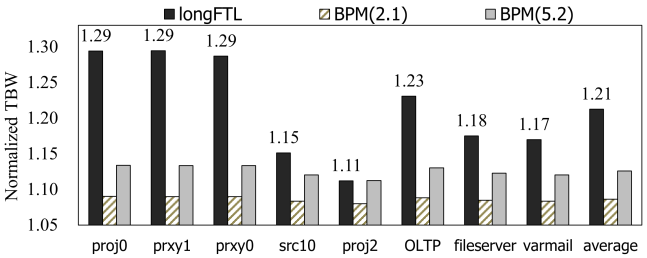


Figure 14: Comparisons of lifetime extensions.

of written data from an I/O trace was not sufficient, the same traces were repeated multiple times so that sufficient write requests can be generated. In Table 4, prxy0, proj0 and proj2 were such cases. Since the MSR traces were extracted from slow HDD-based storage systems, we accelerated the I/O intensity from a host machine by scaling down inter-request intervals by an appropriate factor (e.g., $\frac{1}{3000}$) so that fast SSD processing speed can be properly considered.

6.2 Lifetime Improvement

In order to understand how longFTL improves the SSD lifetime, we measured TBW values for eight I/O traces. Figure 14 shows normalized TBW values for the benchmark I/O traces over Baseline. For the BPM techniques, we used two versions, BPM(2.1) and BPM(5.2), where BPM(r) allows the block capacity to be reduced by r% of the block capacity. LongFTL achieved the highest average improvement ratio, 21%, on the lifetime extension in all eight traces, followed by BPM(5.2) and BPM(2.1). As expected, the lifetime improvement of both BPM(2.1) and BPM(5.2) was insensitive on I/O characteristics of each trace, achieving 9% and 13% improvement ratios mainly depending on the upper limit of the block capacity reduction.

Although longFTL outperformed BPM(2.1) and BPM(5.2) on all the benchmark traces, the efficiency of longFTL significantly varies over different traces. For example, longFTL improved the SSD lifetime by 29% in proj0, prxy1 and prxy0. On the other hand, longFTL can improve the SSD lifetime only by 11% for proj2. The main reason behind a rather large efficiency difference is because the impact on the WAF value of each trace is quite different when gErase modes are applied.

Figure 15 shows how the WAF values change in longFTL. In Figure 15, we included the most commonly used gErase mode for each I/O trace (which we call the dominant gErase mode). In general, the more the number of protected WLs, the higher the WAF value. However, as shown in Figure 15, the exact relationship is workload-dependent. For example, in proj0 and prxy1, where longFTL was the most effective in

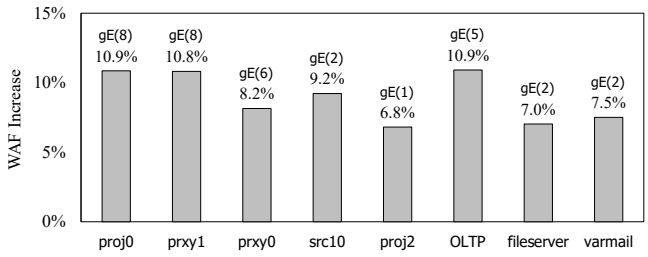


Figure 15: Comparisons of WAF changes under dominant gErase modes used.

increasing the SSD lifetime, the WAF value is only increased by 11% even when the dominant gErase mode protects 28 WLs with gE(8). On the contrary, in proj2, its WAF was very sensitive to the number of protected WLs. Thus we mostly used gE(1) with 8 protected WLs only. When 1.04% of the physical capacity was reduced by gE(1), the WAF value was increased by 7%. Therefore, no higher gE(n) mode was used for proj2, limiting the effect of longFTL on the SSD lifetime. This workload dependency is the main motivation for the WAF Monitor in longFTL. Furthermore, it emphasizes the importance of an FTL for the low-erase modes to be effectively utilized.

6.3 Performance Overhead

Although longFTL selectively decreases the effective SSD capacity from gErase modes, its negative impact on the I/O performance is very small because longFTL applies gErase operation mostly for hot data (as described in Section 5.5). Figure 16 compares normalized IOPS values of three techniques over the Baseline scheme. LongFTL experiences the average 3% IOPS degradation over Baseline. On the contrary, BPM(5.2) degrades its IOPS on average by 10%. Furthermore, BPM(2.1) achieves a similar IOPS level as longFTL but it is ineffective in improving the SSD lifetime. BPM(5.2) may not be employed in practice because it violates the common SSD performance requirement specification (such as the 10% upper limit on the SSD performance degradation at the end of the rated product lifetime [30]). BPM(5.2) can experience more than 20% performance degradation as in proj2.

6.4 Effectiveness of Block Relief Policy

As explained in Section 4.3, longFTL prefers limited applications of gErase modes. For example, when we build the gErase modes of Table 3, we selected ones with the lowest erase relief ratio. Furthermore, longFTL applied gErase modes only for storing host-requested data as described in Section 5.5. That is, no gErase mode is used when storing data during GC. In order to understand the effectiveness of the current conservative policy of longFTL, we compared this policy with a more aggressive policy where gErase modes were always used when erasing blocks.

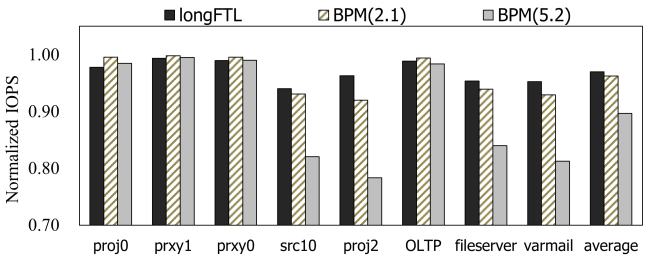


Figure 16: Comparisons of performance overhead.

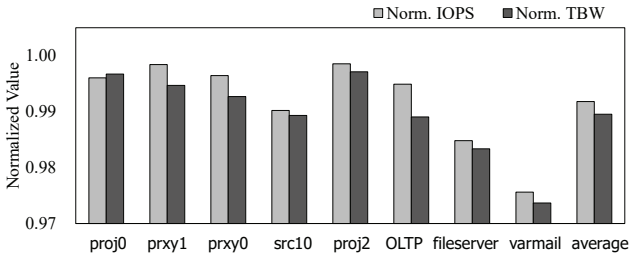


Figure 17: Performance and lifetime comparisons of an aggressive gErase application policy.

Figure 17 shows how the more aggressive policy works over the current one. All the values were normalized over those of longFTL. The evaluation result shows that TBW and IOPS can degrade up to 3% (as in varmail). In the aggressive policy, the TBW improvement ratio can be reduced by up to 15.2%[13] over the current conservative policy.

7 Related Work

Improving the SSD lifetime has been an active research topic because the endurance of modern flash memory is continuously reduced. To overcome the limited lifetime problem, several system-level techniques have been proposed by exploiting the physical characteristics of NAND flash memory [11–14].

Jeong *et al.* conducted a study to improve the NAND endurance by adjusting the erase voltage and erase time [13, 14]. It is similar to our gErase scheme in that it improves the block lifetime by reducing the stress voltage during erase operations. However, unlike the gErase scheme, their technique does not exploit the intra-block reliability variations (i.e., between pages within a single block). It only focuses on reducing the erase-caused stress at the block granularity.

Jimenez *et al.* focused on the intra-block reliability variations and conducted a study to improve NAND endurance through a program stress relief technique [12]. However, this approach is not effective in modern 3D NAND flash memory because the main cause of NAND flash wear-out is erase operations, not program operations. Our gErase scheme focuses on erase operations as a stress relief target. Furthermore, longFTL based on gErase handles a stress relief approach in a more complete fashion, considering the impact of gErase modes on WAF changes.

Debao *et al.* proposed a BPM technique that improves the lifetime by continuously using the remaining pages instead of classifying the entire block as a bad block when a read error occurs on one of pages in the block [11]. The BPM technique may be considered as a valid solution to tackle large reliability variations among pages in the block. However, it is difficult to apply the BPM technique without large I/O performance degradation. Unlike our gErase scheme, the BPM technique cannot recover bad pages for future write requests. Therefore, once the SSD capacity is reduced, the SSD capacity is permanently reduced so that future write-intensive requests cannot be properly serviced.

[13] (1 - (17.8/21.0)) × 100 ≃ 15.2%.

8 Conclusions

We have presented the GuardedErase scheme, a new block erase scheme for modern 3D NAND flash memory, that protects weak WLs from reaching their maximum endurance too early so that the lifetime of a block can be extended. The GuardedErase scheme exploits the trade-off relationship between the NAND endurance and the block capacity so that the SSD lifetime can be effectively extended with minimal impact on I/O performance.

Based on the per-WL low-stress erase mode that was devised from an extensive 3D NAND flash characterization study, we proposed nine different gErase modes and implemented a gErase-aware FTL, longFTL, which uses appropriate gErase modes under varying I/O workload characteristics. Our experimental results show that longFTL can improve the SSD lifetime by 21% on average with negligible degradation on the SSD performance.

The current version of longFTL can be further improved in several directions. For example, in the current version, the impact of a newly selected gE(n) on the future WAF change is measured *after* gE(n) is applied. If it could be predicted in advance with high accuracy, a better gE(n) could be selected earlier, thus further improving the SSD lifetime. It may be an interesting future direction to devise such a predictor using data-driven machine learning techniques.

9 Acknowledgments

We would like to thank Brad Settlemyer, our shepherd, and the anonymous reviewers for their valuable comments that greatly improved our paper. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics, Republic of Korea, under Project Number SRFC-IT2002-06, and by Samsung Electronics Co., Ltd. (IO201207-07809-01). The ICT at Seoul National University provided research facilities for this study. (Corresponding Author: Jihong Kim)

References

[1] Hyunsuk Kim, Su-Jin Ahn, Yu Gyun Shin, Kyupil Lee, and Eunseung Jung. Evolution of nand flash memory: From 2d to 3d as a storage market leader. In *Proceedings of IEEE International Memory Workshop (IMW)*, 2017.

[2] Mark LaPedus. "3d nand's vertical scaling race". https://semiengineering.com/3d-nands-vertical-scaling-race, 2020.

[3] Ki-Tae Park, Sangwan Nam, Daehan Kim, Pansuk Kwak, Doosub Lee, Yoon-He Choi, Myung-Hoon Choi, et al. Three-dimensional 128gb mlc vertical nand flash-memory with 24-wl stacked layers and 50mb/s high-speed programming. In *Proceedings of International Solid-State Circuits Conference*, 2014.

[4] Woopyo Jeong, Jae-woo Im, Doo-Hyun Kim, Sang-Wan Nam, Dong-Kyo Shim, Myung-Hoon Choi, Hyun-Jun Yoon, et al. A 128gb 3b/cell v-nand flash memory with 1gb/s i/o rate. In *Proceedings of International Solid-State Circuits Conference*, 2015.

[5] Dongku Kang, Woopyo Jeong, Chulbum Kim, Doo-Hyun Kim, Yong Sung Cho, Kyung-Tae Kang, Jinho Ryu, et al. 256gb 3b/cell v-nand flash memory with 48 stacked wl layers. In *Proceedings of International Solid-State Circuits Conference*, 2016.

[6] Ryuji Yamashita, Sagar Magia, Tsutomu Higuchi, Kazuhide Yoneya, Toshio Yamamura, Hiroyuki Mizukoshi, Shingo Zaitsu, et al. A 512gb 3b/cell flash memory on 64-word-linelayer bics technology. In *Proceedings of International Solid-State Circuits Conference*, 2017.

[7] Seungjae Lee, Chulbum Kim, Minsu Kim, Sung-min Joe, Joonsuc Jang, Seungbum Kim, Kangbin Lee, et al. A 1tb 4b/cell 64-stacked-wl 3d nand flash memory with 12mb/s program throughput. In *Proceedings of International Solid-State Circuits Conference*, 2018.

[8] Chun-Hsiung Hung, Meng-Fan Chang, Yih-Shan Yang, Yao-Jen Kuo, Tzu-Neng Lai, Shin-Jang Shen, Jo-Yu Hsu, , et al. Layer-aware program-and-read schemes for 3d stackable vertical-gate be-sonos nand flash against cross-layer process variations. *IEEE Journal of Solid-State Circuits*, 50(6):1491–1501, 2015.

[9] Youngseop Shim, Myungsuk Kim, Myoungjun Chun, Jisung Park, Yoona Kim, and Jihong Kim. Exploiting process similarity of 3d flash memory for high performance ssds. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, 2019.

[10] Micron. TN-29-59: Bad block management. https://www.micron.com/-/media/client/global/documents/products/technical-note/nand-flash/tn2959_bbm_in_nand_flash.pdf, 2011.

[11] Wei Debao, Qiao Liyan, Zhang Peng, and Peng Xiyuan. Bpm: A bad page management strategy for the lifetime extension of flash memory. In *Proceedings of International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2015.

[12] Xavier Jimenez, David Novo, and Paolo Ienne. Wear unleveling: Improving nand flash lifetime by balancing page endurance. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2014.

[13] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of nand flash-based storage systems using dynamic program and erase scaling. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2014.

[14] Jaeyong Jeong, Youngsun Song, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Dynamic erase voltage and time scaling for extending lifetime of nand flash-based ssds. *IEEE Transactions on Computers*, 66(4):616–630, 2016.

[15] Rino Micheloni, Luca Crippa, and Alessia Marelli. *Inside NAND flash memories*. Springer Science & Business Media, 2010.

[16] 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 USENIX Conference on File and Storage Technologies*, 2018.

[17] Jungdal Choi and Kwang Soo Seol. 3d approaches for non-volatile memory. In *Proceedings of Symposium on VLSI Technology-Digest of Technical Papers*, 2011.

[18] Youngwoo Park, Jaeduk Lee, Seong Soon Cho, Gyoyoung Jin, and Eunseung Jung. Scaling and reliability of nand flash devices. In *Proceedings of the IEEE International Reliability Physics Symposium*, 2014.

[19] Yixin Luo, Saugata Ghose, Yu Cai, Erich F Haratsch, and Onur Mutlu. Heatwatch: Improving 3d nand flash memory device reliability by exploiting self-recovery and temperature awareness. In *Prpc. International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[20] Samsung v-nand technology, white paper. https://studylib.net/doc/8282074/samsung-v-nand-technology, 2014.

[21] Eun-Seok Choi and Sung-Kye Park. Device considerations for high density and highly reliable 3d nand flash cell in near future. In *Proceedings of IEEE International Electron Devices Meeting (IEDM)*, 2012.

[22] Jaehoon Jang, Han-Soo Kim, Wonseok Cho, Hoosung Cho, Jinho Kim, Sun Il Shim, Jae-Hun Jeong, et al. Vertical cell array using tcat (terabit cell array transistor) technology for ultra high density nand flash memory. In *Proceedings of IEEE Symposium on VLSI Technology (VLSI)*, 2009.

[23] Jaehoon Jang, Han-Soo Kim, Wonseok Cho, Hoosung Cho, Jinho Kim, Sun Il Shim, Jae-Hun Jeong, et al. Vertical cell array using TCAT (Terabit Cell Array Transistor) technology for ultra high density NAND flash memory. In *Proceedings of the IEEE Symposium on VLSI Technology (VLSI)*, 2009.

[24] Klaus F Schuegraf and Chenming Hu. Effects of temperature and defects on breakdown lifetime of thin sio/sub 2/at very low voltages. In *Proceedings of IEEE International Reliability Physics Symposium (IRPS)*, 1994.

[25] Duwon Hong. *Optimizing the Reliability and Performance of Ultra-large SSDs*. PhD thesis, Dept. of Computer Science and Engineering, College of Engineering, Seoul National University, 2021.

[26] Benny Van Houdt. On the necessity of hot and cold data identification to reduce the write amplification in flash-based ssds. *Performance Evaluation*, 82:1–14, 2014.

[27] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: practical power management for enterprise storage. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2008.

[28] Sysbench. http://github.com/akopytov/sysbench.

[29] Filebench. http://filebench.sourceforge.net.

[30] Ulrich Hansen. The ssd endurance race: Who's got the write stuff? https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2012/20120821_TC11_Hansen.pdf, 2012.

Hardware/Software Co-Programmable Framework for Computational SSDs to Accelerate Deep Learning Service on Large-Scale Graphs

Miryeong Kwon Donghyun Gouk Sangwon Lee Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory
Korea Advanced Institute of Science and Technology (KAIST)
http://camelab.org

Abstract

Graph neural networks (GNNs) process large-scale graphs consisting of a hundred billion edges. In contrast to traditional deep learning, unique behaviors of the emerging GNNs are engaged with a large set of graphs and embedding data on storage, which exhibits complex and irregular preprocessing.

We propose a novel deep learning framework on large graphs, *HolisticGNN*, that provides an easy-to-use, near-storage inference infrastructure for fast, energy-efficient GNN processing. To achieve the best end-to-end latency and high energy efficiency, HolisticGNN allows users to implement various GNN algorithms and directly executes them where the actual data exist in a holistic manner. It also enables RPC over PCIe such that the users can simply program GNNs through a graph semantic library without any knowledge of the underlying hardware or storage configurations.

We fabricate HolisticGNN's hardware RTL and implement its software on an FPGA-based computational SSD (CSSD). Our empirical evaluations show that the inference time of HolisticGNN outperforms GNN inference services using high-performance modern GPUs by 7.1× while reducing energy consumption by 33.2×, on average.

1 Introduction

Graph neural networks (GNNs) have recently emerged as a representative approach for learning graphs, point clouds, and manifolds. Compared to traditional graph analytic methods, GNNs exhibit much higher accuracy in a variety of prediction tasks [28, 49, 64, 90, 94, 101], and their generality across different types of graphs and algorithms allows GNNs to be applied by a broad range of applications such as social networks, knowledge graphs, molecular structure, and recommendation systems [8, 21, 29, 52]. The state-of-the-art GNN models such as GraphSAGE [27] further advance to infer unseen nodes or entire new graphs by generalizing geometric deep learning (DL). The modern GNN models in practice sample a set of subgraphs and DL feature vectors (called *embeddings*) from the target graph information, and aggregate the sampled embeddings for inductive node inferences [27, 95]. This *node sampling* can significantly reduce the amount of data to process, which can decrease the computation complexity to infer the results without an accuracy loss [9, 27, 96].

While these node sampling and prior model-level efforts for large graphs make the inference time reasonable, GNNs yet face system-level challenges to improve their performance. First, GNNs experience a completely different end-to-end inference scenario compared to conventional DL algorithms. In contrast to the traditional DLs, GNNs need to deal with real-world graphs consisting of billions of edges and node embeddings [19, 73]. The graph information (graph and node embeddings) initially reside in storage and are regularly updated as raw-format data owing to their large size and persistence requirements. As GNNs need to understand the structural geometry and feature information of given graph(s), the raw-format data should be loaded into working memory and reformatted in the form of an adjacency list before the actual inference services begin. These activities take a significant amount of time since the graph information often exceeds hundreds of GBs or even a TB of storage [18, 76]. We observe that the pure inference latency, that all the previous GNN studies try to optimize, accounts for only 2% of the end-to-end inference service time when we execute diverse GNN models in a parallel system employing high-performance GPUs [13, 14] and an SSD [12]. We will analyze this performance bottleneck issue with detailed information in Section 2.3.

Second, GNNs consist of various computing components, which are non-trivial to fully accelerate or parallelize over conventional computing hardware. As GNNs are inspired by conventional DL algorithms such as convolution neural networks and representative learning [27, 64, 79], several data processing parts of GNN computing are associated with dense matrix computing. While these matrix multiplications can be accelerated by existing data processing units (DPUs) such as systolic architectures, the graph-natured operations of GNNs can neither be optimized with DPU's multiplication hardware nor with GPUs' massive computing power [5].

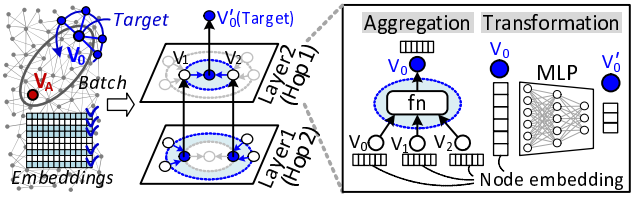
A promising alternative to address the aforementioned challenges is employing in-storage processing (ISP) to serve GNN inferences directly from the underlying storage. While ISP is very a well-known solution heavily studied in the literature for the past few decades [25, 34, 43, 46, 58, 65], it has

unfortunately not been widely adopted in real-world systems [6]. There are several reasons, but the most critical issue of ISP is ironically its concept itself, which co-locates flash and computing unit(s) into the same storage box. As flash is not a working memory but a block device, it is integrated into the storage box with complicated firmware and multiple controllers [11, 36, 99]. These built-in firmware and controllers are not easily usable for the computation that users want to offload as it raises many serious technical challenges such as programmability, data protection, and vendor dependency issues. In this work, we advocate a new concept of *computational SSD* (CSSD) architectures that locate reconfigurable hardware (FPGA) near storage in the same PCIe subsystem [20, 68, 80]. In contrast to ISP, CSSD can maximize peer-to-peer acceleration capability and make it independent from specific storage firmware and controller technologies. However, it is challenging to configure everything that users want to program and/or download in the form of completely full hardware logic into FPGA from scratch.

We propose *HolisticGNN*, a hardware and software co-programmable framework that leverages CSSD to accelerate GNN inference services near storage. HolisticGNN offers a set of software and hardware infrastructures that execute GNN tasks where data exist and infer the results from storage in a holistic manner. Generally speaking, the software part of HolisticGNN enables users to program various GNN algorithms and infer embedding(s) directly atop the graph data without the understanding complexities of the underlying hardware and device interfaces. On the other hand, our hardware framework provides fundamental hardware logic to make CSSD fully programmable. It also provides an architectural environment that can accelerate various types of GNN inferences with different hardware configurations.

For fast and energy-efficient GNN processing, our framework is specifically composed of three distinguishable components: i) graph-centric archiving system (*GraphStore*), ii) programmable inference client and server model (*GraphRunner*), and iii) accelerator building system (*XBuilder*). The main purpose of GraphStore is to bridge the semantic gap between the graph abstraction and its storage representation while minimizing the overhead of preprocessing. GraphStore manages the user data as a graph structure rather than exposing it directly as files without any intervention of host-side software. This allows diverse node sampling and GNN algorithms to process the input data near storage immediately. GraphStore also supports efficient mutable graph processing by reducing the SSD's write amplification.

To accommodate a wide spectrum of GNN models, it is necessary to have an easy-to-use, programmer-friendly interface. GraphRunner processes a series of GNN inference tasks from the beginning to the end by allowing users to program the tasks using a computational graph. The users can then simply transfer the computational graph into the CSSD and manage its execution through a remote procedure call (RPC). This does not require cross-compilation or storage



(a) Preprocessing. (b) GNN processing. (c) Layer's details.

Figure 1: Overview of basic GNN algorithm.

stack modification to program/run a user-defined GNN model. We enable RPC by leveraging the traditional PCIe interface rather than having an extra hardware module for the network service, which can cover a broad spectrum of emerging GNN model implementations and executions in CSSD.

On the other hand, XBuilder manages the FPGA hardware infrastructure and accelerates diverse GNN algorithm executions near storage. It first divides the FPGA logic die into two regions, *Shell* and *User*, using the dynamic function exchange (DFX) technique [82]. XBuilder then secures hardware logic necessary to run GraphStore and GraphRunner at Shell while placing DL accelerator(s) to User. The Shell and User hardware are programmed to CSSD as two separate bitstreams, such that we can reprogram the User with a different bitstream at any time. To this end, XBuilder implements a hardware engine in Shell by using an internal configuration access port, which downloads a bitstream and programs it to User.

We implement HolisticGNN on our CSSD prototype that places a 14*nm* FPGA chip [87] and 4TB NVMe device [12] under a same PCIe switch. We also prototype the software framework of HolisticGNN on the CSSD bare-metal, and we fabricate/test various GNN accelerator candidates within CSSD, such as a many-core processor, systolic arrays, and a heterogeneous (systolic+vector) processor. Our evaluations show that the inference time of HolisticGNN outperforms GNN inference services using high-performance GPUs by 7.1× while consuming 33.2× less energy, on average.

2 Background

2.1 Graph Neural Networks

Graph neural networks (GNNs) generalize conventional DL to understand structural information in the graph data by incorporating feature vectors (*embeddings*) in the learning algorithms [27, 28, 95]. GNNs can capture topological structures of the local neighborhood (per node) in parallel with a distribution of the neighborhood's node embeddings [7, 9, 27, 96].

General concept. As shown in Figure 1, GNNs in general take three inputs, a graph, the corresponding node embeddings (e.g., user profile features), a set of unseen/seen nodes to infer, called *batch*. Since the internal memory of GPUs is insufficient to accommodate all the inputs, it is essential to reduce the size of the graph and embeddings by preprocessing them appropriately (Figure 1a), which will be explained in Section 2.2. GNNs then analyze the preprocessed structural

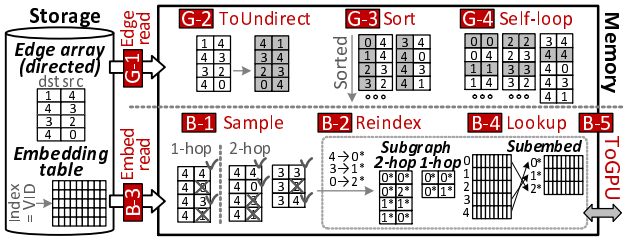


Figure 2: Holistic viewpoint of GNN computing (**G**: Graph preprocessing, **B**: Batch preprocessing).

information with node embeddings over multiple computational layers (Figure 1b). Each layer of GNNs is composed of two primary execution phases, called neighborhood *aggregation* and node *transformation* [79, 90], which are all performed for neighbors at different hop distances (connected to a target node in the batch). Specifically, as shown in Figure 1c, the aggregation is a simple function to accumulate node embeddings of the target node's neighbors, whereas the transformation converts the aggregated results to a new node embedding using one or more traditional *multi-layer perceptrons* (MLPs [31, 32]). Therefore, the aggregation processes data relying on graph structures and mainly exhibits irregular, graph-natured execution patterns. In contrast, the transformation computing procedure is very similar to that of conventional neural networks (e.g., CNNs and RNNs), but it does not require heavy computation. For example, GNNs mostly use only 2∼3 layers [15, 42, 72, 75, 90], whereas Google BERT employs more than 24 layers and needs to perform heavy matrix multiplications [17].

Note that, while the massive parallel computing of GPUs is very well-optimized for many DL algorithm executions, these characteristics of GNNs (e.g., irregular execution pattern and relatively lightweight computation) allow other processing architectures to be a better fit for GNN acceleration.

Model variations. Based on how to aggregate/transform embeddings, there is a set of variant GNNs, but *graph convolution network* (GCN [42]), *graph isomorphism network* (GIN [90]), and *neural graph collaborative filtering* (NGCF [75]) are the most popular GNN models used in node/graph classification and recommendation systems [21, 29, 49, 94, 96, 101].

Specifically, GCN uses an *"average-based aggregation"* that normalizes the embeddings by considering the degree of neighbor nodes. This prevents cases where a specific embedding has excessively large features, thereby losing other embeddings that have relatively small amounts of data in the aggregation phase. In contrast, GIN uses a *"summation-based aggregation"* that does not normalize the embeddings of both the target node (self-loop) and its neighbors. In addition, GIN gives a learnable self-weight to the target node embedding to avoid unexpectedly losing its feature information due to the heavy states and features of the target node's neighbors. To precisely capture the structural characteristics of the given graph, GIN uses a two-layer MLP structure, making the combination more expressively powerful. GCN and GIN suppose that all the feature vectors of a given graph have the same level of weight, which are widely used for node and graph classifications [54, 94, 101]. Instead of using a simple average or summation for the aggregation, NGCF takes the similarity among the given graph's embeddings into account by applying an element-wise product to neighbors' embeddings.

Even though there are several variants of GNNs, they all require the graph's geometric information to analyze embeddings during the aggregation and transformation phases. Thus, it is necessary for GNNs to have an easy-to-access, efficient graph and embedding data structures.

2.2 Graph Dataset Preprocessing

The graph data offered by a de-facto graph library such as SNAP [48] in practice deal with edge information in the form of a text file. The raw graph file includes an (unsorted) edge array, each being represented by a pair of destination and source *vertex identifiers* (VIDs). Most GNN frameworks such as deep graph library (DGL) [74] and pytorch geometric (PyG) [22] preprocess the graph dataset to secure such easy-to-access graph and embeddings as a VID-indexed table or tensor. In this work, we classify these graph dataset preprocessing tasks into two: i) *graph preprocessing* and ii) *batch preprocessing*. While graph preprocessing is required only for the geometrical data (including the initialization), batch preprocessing should be performed for each inference service.

Graph preprocessing. Since the majority of emerging GNN algorithms are developed based on spatial or spectral networks encoding *"undirected"* geometric characteristics [15, 42], the main goal of this graph preprocessing is to obtain a sorted, undirected graph dataset. As shown in the top of Figure 2, it first loads the edge array (raw graph) from the underlying storage to the working memory [G-1]. To convert the array to an undirected graph, the GNN frameworks (e.g., DGL) allocate a new array and copy the data from the edge array to the new array by swapping the destination and source VIDs for each entry ($\{dst, src\} \rightarrow \{src, dst\}$) [G-2]. The frameworks merge and sort the undirected graph, which turns it into a VID-indexed graph structure [G-3]. As the target node to infer is also included in the 1-hop neighborhood, the frameworks inject self-loop information (an edge connecting a vertex to itself) to the undirected graph as well ($\{0,0\}, \{1,1\}, \cdots \{4,4\}$) [G-4]. If there is no self-loop information, the aggregation of GNNs cannot reflect a visiting node's features, which in turn reduces the inference accuracy significantly.

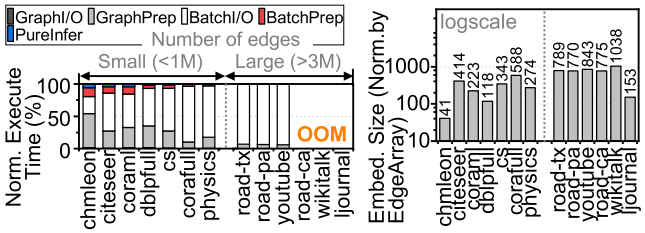
Batch preprocessing. Large-scale real-world graphs consist of a hundred billion edges, and each node of the edges is associated with its own embedding containing thousands of DL features. The number of nodes and the embedding size that the current GNN models process are typically an order of magnitude greater than heavy featured DL applications, such as natural language processing [18, 76]. Thus, for a given batch, the frameworks in practice perform *node sampling* such as random walk [92] and unique neighbor sampling [27]. The node sampling specifically extracts a set of subgraphs

and the corresponding embeddings from the original (undirected) graph datasets before aggregating and transforming the feature vectors, which can significantly reduce data processing pressures and decrease the computing complexity without an accuracy loss [27, 33]. Since the sampled graph should also be self-contained, the subgraphs and embeddings should be reindexed and restructured. We refer to this series of operations as *batch preprocessing*.

The bottom of Figure 2 shows an example of batch preprocessing. For the sake of brevity, this example assumes that the batch includes a just single target, V_4 (VID=4), the given sampling size is 2, and GNN is modeled with two layers (two hops). This example first reads all V_4's neighbors and extracts two nodes from the undirected graph (in a random manner) [B-1]. This generates a subgraph including the 1-hop neighbors, which is used for GNN's layer 2 computation (L2). For the sampled nodes (V_4 and V_3), it reads their neighbor nodes (2-hop) and samples the neighborhood again for GNN's layer 1 (L1). Since the number of nodes has been significantly reduced, the GNN frameworks allocate new VIDs in the order of sampled nodes (4→ 0*, 3→ 1*, and 0→ 2*) and create L1 and L2 subgraphs for 2-hop and 1-hop neighbors, respectively [B-2]. It then composes an embedding table whose index is the VID of each sampled node. To this end, the frameworks first need to load the embeddings from the underlying storage to working memory [B-3], called global embeddings, and lookup the embeddings of L1's subgraph (V_4, V_0, and V_3) [B-4]. Lastly, the subgraphs and sampled embedding table are required to transfer from the working memory to the target GPU's internal memory [B-5].

2.3 Challenge Analysis

While there is less system-level attention on the management of graph and batch preprocessing, their tasks introduce heavy storage accesses and frequent memory operations across the boundary of user space and storage stack. To be precise, we decompose the *"end-to-end GCN inference"* times across 14 real-world graph workloads (coming from [48, 61, 66, 93]) into the latency of graph preprocessing (`GraphPrep`), batch preprocessing (`BatchPrep`), GCN inference processing (`PureInfer`), and storage accesses for graph (`GraphI/O`) and embeddings (`BatchI/O`). Since the storage access latency being overlapped with the latency of preprocessing computation is invisible to users, this breakdown analysis excludes such latency, and the results are shown in Figure 3a. The detailed information of the evaluation environment is provided by Section 5. One can observe from this breakdown analysis that `PureInfer` only takes 2% of the end-to-end inference latency, on average. Specifically, `BatchI/O` accounts for 61% of the most end-to-end latency for the small graphs having less than 1 million edges. Before the sorted and undirected graph is ready for batch preprocessing, `BatchI/O` cannot be processed. Since `GraphPrep` includes a set of heavy (general) computing processes such as



(a) Latency breakdown. (b) Embed vs. Edge array.

Figure 3: End-to-End GNN execution.

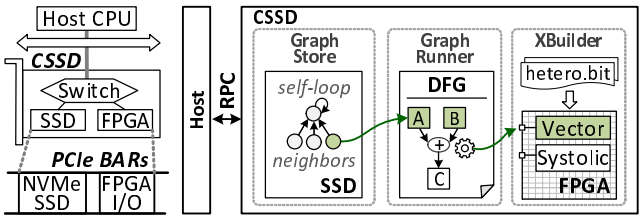
a radix sort, `GraphPrep` also consumes 28% of the end-to-end latency for these small graphs. As the graph size increases (> 3 million edges), `BatchI/O` becomes the dominant contributor of the end-to-end GNN inference time (94%, on average). Note that the inference system has unfortunately stopped the service during the preprocessing due to out-of-memory (OOM) when it handles large-scale graphs (>3 million edges) such as `road-ca`, `wikitalk`, and `ljournal`. This OOM issue can be simply addressed if one services the GNN inference from where the data exist. In addition, the heavy storage accesses and relatively lightweight computing associated with inference (`PureInfer`) make adoption of the in-storage processing concept [39] reasonable to shorten the end-to-end inference latency.

Figure 3b normalizes the size of the embedding table to that of the edge array (graph) across all the graphs that we tested. As shown in this comparison, embedding tables of the small and large graphs are greater than the edge arrays for those graphs by 285.7× and 728.1×, on average, respectively. This is because an embedding has thousands of DL features, each represented using floating-point values with high precision [61, 95]. In contrast, an entry of the edge arrays contains only a simple integer value (VID). This characteristic makes batching preprocessing I/O intensive while inducing graph preprocessing to be computation-intensive.

3 Storage as a GNN Accelerator

In-storage processing (ISP) is well-studied in the research literature [2, 25, 35, 41, 43, 46, 58, 62, 65], but it has been applied to accelerate limited applications such as compression and key-value management in real-world systems. There are several reasons, but the greatest weakness of ISP ironically is that it needs to process data where data is stored, i.e., at the flash device. Flash cannot be directly used as block storage because of its low-level characteristics, such as I/O operation asymmetry and low reliability [10, 37, 38, 55, 56, 78]. Thus, flash requires tight integration with multiple firmware and controller modules [98, 99], which renders ISP difficult to be implemented within an SSD.

In contrast to ISP, as shown in Figure 4a, the new concept of computational SSDs (CSSDs) decouples the computing unit from the storage resources by locating *reconfigurable hardware* (FPGA) near SSD in the same PCIe subsystem (card) [20]. CSSD allows the hardware logic fabricated in FPGA to access the internal SSD via the internal PCIe switch.



(a) CSSD. (b) Overview of HolisticGNN.

Figure 4: Enabling CSSD for near storage GNN processing.

To this end, the host is responsible for writing/reading data on the SSD using the I/O region of NVMe protocol while giving the data's block address to the FPGA through its own I/O region, whose address is designated by PCIe's base address register [84]. While CSSD is promising to realize near-data processing [44, 68], it is non-trivial to automate all end-to-end procedures of GNN inference over hardware-only logic because of the variety of GNN model executions. For example, the aggregation and/or combination of GNNs can be accelerated with parallel hardware architecture, but GNN's graph traversing, dataset preprocessing, and embedding handling are impractical to be programmed into hardware because of their graph-natured computing irregularities.

3.1 Overview of HolisticGNN

HolisticGNN is a hardware and software co-programmable framework that leverages CSSD to accelerate the end-to-end GNN inference services near storage efficiently. The software part of our framework offers easy-to-use programming/management interfaces and performs GNN preprocessing directly from where the data is stored, thereby minimizing the aforementioned storage access overhead. HolisticGNN can also eliminate the out-of-memory issue for deep learning on large-scale graphs. On the other hand, our framework's hardware logic and administration module provide a low-overhead bare-metal computing environment and reconfigurable hardware to accelerate GNN model executions.

Figure 4b illustrates a high-level view of HolisticGNN, which is composed of three major modules: i) graph-centric archiving system (*GraphStore*), ii) programmable inference model (*GraphRunner*), and iii) accelerator builder (*XBuilder*). Generally speaking, GraphStore prepares the target graph data and embeddings in a ready-to-access structure that the tasks of batch preprocessing can immediately use without preprocessing the datasets. On the other hand, GraphRunner executes a series of GNN inference tasks from the beginning to the end, and it processes the graph datasets by directly accessing SSD through GraphStore. GraphRunner also provides a *dataflow graph* (DFG) based program and execution model to support easy-to-use and flexible implementation of a wide spectrum of GNN algorithms. This enables the users to simply generate a DFG and deliver it to HolisticGNN, which can dynamically change the end-to-end GNN inference services without cross-compilation and/or understanding underlying hardware configurations. Lastly, XBuilder makes CSSD simply reconfigurable and has heterogeneous hardware components to satisfy the diverse needs of GNN inference acceleration services. XBuilder also provides several kernel building blocks, which abstract the heterogeneous hardware components. This can decouple a specific hardware acceleration from the GNN algorithm implementation.

Service type	RPC function	Service type	RPC function
GraphStore (Bulk)	`UpdateGraph (EdgeArray, Embeddings)`	GraphStore (Unit, Get)	`GetEmbed(VID)`
GraphStore (Unit, Update)	`AddVertex(VID,Embed)`		`GetNeighbors(VID)`
	`DeleteVertex(VID)`	Graph Runner	`Run(DFG, batch)`
	`AddEdge(dstVID,srcVID)`		`Plugin(shared_lib)`
	`DeleteEdge(dstVID,srcVID)`	XBuilder	`Program(bitfile)`
	`UpdateEmbed(VID,Embed)`		

Table 1: RPC services of HolisticGNN.

Each module of our framework exposes a set of APIs through remote procedure calls (RPCs) to users. These APIs are not related to GNN programming or inference services, but to framework management such as updating graphs, inferring features, and reprogramming hardware logic. Since CSSD has no network interface for the RPC-based communication, we also provide an *RPC-over-PCIe* (RoP) interface that overrides the conventional PCIe to enable RPC between a host and CSSD without an interface modification.

3.2 Module Decomposition

Graph-centric archiving system. The main goal of GraphStore is to bridge the semantic gap between graph and storage data without having a storage stack. As shown in Table 1, GraphStore offers two-way methods for the graph management, *bulk operations* and *unit operations*. The bulk operations allow users to update the graph and embeddings with a text form of edge array and embedding list. For the bulk operations, GraphStore converts the incoming edge array to an adjacency list in parallel with transferring the embedding table, and it stores them to the internal SSD. This makes the conversion and computing latency overlapped with the heavy embedding table updates, which can deliver the maximum bandwidth of the internal storage. In contrast, the unit operations of GraphStore deal with individual insertions (`AddVertex()`/`AddEdge()`), deletions (`DeleteVertex()`/`DeleteEdge()`), and queries (`GetEmbed()`/`GetNeighbors()`) for the management of graph datasets. When GraphStore converts the graph to storage semantic, it uses VID to *logical page number* (LPN) mapping information by being aware of a long-tailed distribution of graph degree as well as flash page access granularity. The LPNs are used for accessing CSSD's internal storage through NVMe, which can minimize the write amplification caused by I/O access granularity differences when CSSD processes GNN services directly on the SSD. The design and implementation details are explained in Section 4.1.

Programmable inference model. GraphRunner decouples CSSD task definitions from their actual implementations, which are called *C-operation* and *C-kernel*, respectively. To program a GNN model and its end-to-end service, the users can write a DFG and download/execute to CSSD by call-

API type	Function	API type	Operation format
DFG Creation	createIn(name)	XBuilder	GEMM(inputs, output)
	createOp(name)		ElementWise(inputs, output)
	createOut(name)		Reduce(inputs, output)
	save(graph)		SpMM(inputs, output)
Plugin	RegisterDevice(newDevice)		SDDMM(inputs, output)
	RegisterOpDefinition(newOp)		

Table 2: Programming interface of HolisticGNN.

ing GraphRunner's RPC interface (`Run()`) with a request batch containing one or more target nodes. Figure 10b shows a simple example of GCN implementation. The DFG has a set of input nodes for the target sampled subgraphs, embeddings, and weights, which are connected to a series of C-operations such as averaging features (Mean), matrix multiplication (Matmul), a non-linear function (ReLU), and output feature vector (Out_embedding). This DFG is converted to a computational structure by sorting the node (C-operation) and edge (input node information) in topological order. Once the DFG is downloaded through HolisticGNN's RoP serialization, GraphRunner's engine deserializes it and executes each node with appropriate inputs by checking the registered C-operations and C-kernels in CSSD. The users may want to register more C-operations/kernels because of adoption of a new GNN model or hardware logic. To meet the requirement, GraphRunner offers a Plugin mechanism registering a pair of C-operation/C-kernel and a new device configuration as a shared object. We will explain the details of GraphRunner in Section 4.2.

Accelerator builder. To make the FPGA of CSSD easy to use, we configure CSSD's hardware logic die into two groups, *Shell* and *User* logic, by leveraging a dynamic function exchange (DFX) mechanism [82]. DFX allows hardware to be modified blocks of logic with separate *bitfiles* that contain the programming information for an FPGA. XBuilder secures Shell logic associated with irregular tasks of GNN management, including GraphStore and GraphRunner executions, while managing User logic for users to reprogram the hardware in accelerating GNN algorithms via XBuilder's RPC interface (`Program()`). `Program()` moves a partial bitfile into the internal memory and asks an XBuilder engine to reconfigure User logic hardware using the bitfile via FPGA internal configuration access port [85, 86].

XBuilder also abstracts the registered device (at User logic) by providing a set of basic building blocks to the users as shown in Table 2. The building blocks basically implement what DL and GNN algorithms mostly use, such as general matrix multiplication (GEMM) and sparse matrix multiplication (SpMM), across different legacy acceleration hardware such as multi-core, vector processor, systolic architecture. XBuilder's building blocks operate specific hardware based on the device priority designated by C-kernel that user defines. We will discuss this in details in Section 4.3.

3.3 Enabling RPC over PCIe

While the key method to program GNN models (and request their inference services) is associated with DFG, the underpinning of such a device-to-device communication method

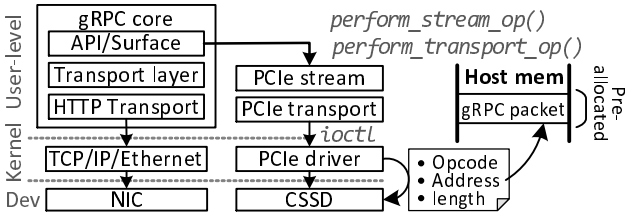


Figure 5: RPC over PCIe (RoP).

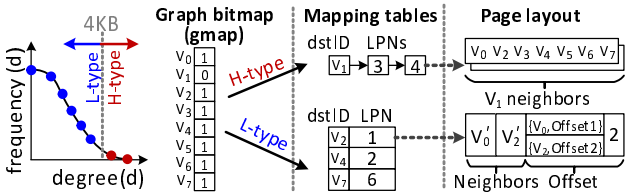
is RPC. As the investigation of efficient RPC is not the purpose of this work, we use Google's gRPC [24] and implement our RPC-based interfaces themselves (e.g., `UpdateGraph()`, `Run()`, etc.) using interface definition language (IDL) [16]. We also modify the gRPC stack to enable RPC services without changing hardware and storage interfaces.

Figure 5 explains the gRPC stack and how HolisticGNN enables gRPC over PCIe. The host-side gPRC interfaces are served by a user-level gRPC core, which manages transport and HTTP connection. We place two gRPC plugin interfaces (`perform_stream_op()` and `perform_transport_op()`), each forwarding the requests of gRPC core's transport layer and HTTP transport to our PCIe stream and PCIe transport modules. Specifically, the PCIe stream is responsible for managing `stream` data structures, which are used for gPRC packet handling. Similarly, the PCIe transport deals with the host and CSSD connection by allocating/releasing `transport` structures. While the original gPRC core is built upon a kernel-level network stack including TCP/IP and Ethernet drivers, we place a PCIe kernel driver connected to the PCIe transport. It supports gRPC's send/receive packet services and other channel establishment operations to the PCIe transport module via `ioctl`. The PCIe kernel driver also provides preallocated buffer memory to the PCIe stream through a memory-mapped I/O (`mmap`). This buffer memory contains gPRC packet's metadata and message such that the PCIe driver lets the underlying CSSD know the buffer's location and offset. Specifically, the PCIe drive prepares a PCIe command that includes an opcode (send/receive), address (of memory-mapped buffer), and length (of the buffer). When the driver writes the command to FPGA's designated PCIe memory address, CSSD parses the command and copies the data from the memory-mapped buffer into FPGA-side internal memory for gRPC services.

4 Design Details and Implementation

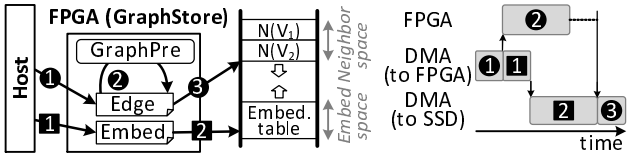
4.1 Efficient Storage Accesses for Graphs

GraphStore maintains the graph datasets as an adjacency list and an embedding table to handle the geometric information and feature vectors. While the embedding table is stored in sequential order (and thus it does not require page-level mapping), the adjacency list is maintained in two different ways by considering the efficiency of graph searches/updates: i) high-degree graph mapping (*H-type*) and ii) low-degree graph mapping (*L-type*). As shown in Figure 6a, the power-law graph's natures make a few nodes have severely heavy



(a) Distribution. (b) Two types of mapping.

Figure 6: GraphStore's mapping structure.



(a) Procedure. (b) Overlapped execution.

Figure 7: Bulk operations.

neighbor nodes (high-degree) [59]. These high-degree nodes account for a small fraction of the entire graph, but they have a high potential to be frequently accessed and updated (because of their many neighbors). H-type mapping is therefore designed towards handling the graph's long-tailed distribution well, while L-type mapping is structured to achieve high efficiency of flash page management.

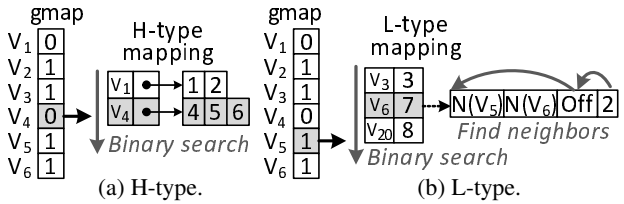
Mapping structure. As shown in Figure 6b, GraphStore has a graph bitmap (*gmap*), which explains what kind of tables are used for mapping (per VID). Basically, the mapping entry for both types of mapping tables pairs a VID and an LPN (VID-to-LPN), but the corresponding page stores different data with its own page layout. The H-type page maintains many neighbors' VID in a page, and its mapping table entry indicates a linked list in cases where the neighbors of the target (source) VID cannot be stored in a flash page (4KB). The L-type page also contains many VIDs, but their source VIDs vary. To this end, the end of page has meta-information that indicates how many nodes are stored and where each node exists on the target page (offset). Thus, L-type mapping table's VID is the biggest VID among VIDs stored in the corresponding page.

Bulk operation. As shown in Figure 7, while the embedding table is stored from the end of LPN (embedding space), the graph pages are recorded from the beginning of storage (neighbor space), similar to what the conventional memory system stack does. Note that, the actual size of graph(s) is small enough, but it is involved in heavy graph preprocessing, and the majority of graph datasets are related to their node embeddings (cf. Section 2.3). Thus, when an edge array (graph) arrives, GraphStore performs graph preprocessing and flushes pages for the graph, but it does not immediately update them to the target storage. Instead, GraphStore begins to write the embedding table into the embedding space in a sequential manner while preprocessing the graph. This can make heavy storage accesses (associated with the embeddings) entirely overlap with the computation burst of graph preprocessing (associated with adjacency list conversions). From the user's viewpoint, the latency of bulk operation is the same as that of data transfers and embedding table writes.

Unit operations. GraphStore's unit operations support mutable graph management corresponding to individual vertex/edge updates or queries. Figure 8 shows how to find out neighbors of V_4 and V_5, each being classified as high-degree and low-degree nodes. In this example, as the gmap indicates that V_4 is managed by the H-type mapping, the neighbors can be simply retrieved by searching where the target VID is. In contrast, the page managed by L-type contains many neighborhoods each being associated with different VID. Therefore, when GraphStore searches the mapping table, it considers the range of VIDs, stored in each entry. For example, V_5 is within the range of V_4 and V_6, GraphStore first needs to retrieve the page corresponding V_6. It finds out V_5's offset and the next VID's offset (V_6) by considering the number of node counts in the page's meta-information, which indicates the data chunk containing V_5's neighbors.

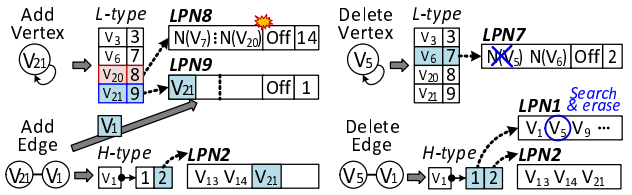
Figures 9a and 9b show add operations (`AddEdge()` / `AddVertex()`) and delete operations (`DeleteEdge()` / `DeleteVertex()`). Let us suppose that V_{21} is given by `AddVertex()` (Figure 9a). GraphStore checks the last entry's page (LPN8) of L-type and tries to insert V_{21} into the page. However, as there is no space in LPN8, GraphStore assigns a new entry ([V_{21},9]) to the L-type mapping table by allocating another page, LPN9, and simply appends the vertex information (V_{21}) to the page. Note that, when adding a vertex, it only has the self-loop edge, and thus, it starts from L-type. When $V_{21}\rightarrow V_1$ is given by `AddEdge()`, GraphStore makes it an undirected edge ($V_{21}\rightarrow V_1$ & $V_{21}\leftarrow V_1$). As V_1 is H-type, GraphStore checks V_1's linked list and places V_{21} to the last page (LPN2). If there is no space in LPN2, it allocates a new page and updates the linked list with the newly allocated page. In contrast, since V_{21} is L-type, GraphStore scans the meta-information of LPN9 and appends V_1 to the page. Note that, in cases where there is no space in an L-type page, GraphStore evicts a neighbor set (represented in the page) whose offset of the meta-information is the most significant value. This eviction allocates a new flash page, copies the neighbor set, and updates L-type mapping table. Since each L-type's destination node has a few source nodes, this eviction case is very rare in practice (lower than 3% of the total update requests for all graph workloads we tested).

On the other hand, delete operations (Figure 9b) consist of search and erase tasks. If `DeleteVertex()` is called with V_5, GraphStore finds out LPN7 and deletes all the neighbors of V_5, N(V_5). During this time, other neighbors having V_5 should also be updated together. For `DeleteEdge()` with the given $V_5\rightarrow V_1$, GraphStore checks all the LPNs indicated by the



(a) H-type. (b) L-type.

Figure 8: Unit operations (Get).



(a) AddEdge/AddVertex. (b) DeleteEdge/DeleteVertex.

Figure 9: Unit operations (Update).

linked list of V_1 and updates the corresponding page (LPN1 in this example) by removing V_5. Note that, GraphStore does not have explicit page compaction for the node/edge deletions in an L-type page. This is because, when there is a deletion, GraphStore keeps the deleted VID and reuses it (and the corresponding neighbor set space) for a new node allocation.

4.2 Reconfiguring Software Framework

HolisticGNN provides a CSSD library package, which includes the interfaces for C-kernel registration and DFG management as explained previously (cf. Table 2).

C-kernel registration and management. GraphRunner manages C-kernels by employing a registration mechanism and an execution engine. GraphRunner has two metadata structures, *Device table* and *Operation table*. As shown in Table 3, the device table includes currently registered device names and the corresponding priority. On the other hand, the operation table maintains C-operation names and the address pointers of their C-kernel implementation. When users implement a C-kernel, it should invoke two registration interface methods of the Plugin library, `RegisterDevice()` and `RegisterOpDefinition()`, at its initial time. `RegisterDevice()` configures the priority value of the device that users want to execute for any of C-kernels (e.g., "Vector processor", 150). On the other hand, `RegisterOpDefinition()` registers the device that this C-kernel. When GraphRunner registers the C-kernel, it places the registration information as a pair of the device name and such C-kernel's pointer. If there are multiple calls of `RegisterOpDefinition()` with the same name of a C-operation (but a different name of device), GraphRunner places it in addition to the previously registered C-kernels as a list. In this example, GraphRunner can recognize that GEMM C-operation defines three C-kernels each using "CPU", "Vector processor", and "Systolic array" by referring to the operation table. Since the device table indicates "Systolic array" has the highest priority, GraphRunner takes the C-kernel associated with "Systolic array" for the execution of GEMM C-operation.

Device table

Name	Priority
"CPU"	50
"Vector processor"	150
"Systolic array"	300
...	...

Operation table

Name	C-kernel
"GEMM"	<"CPU", ptr>
	<"Vector processor", ptr>
	<"Systolic array", ptr>
...	...

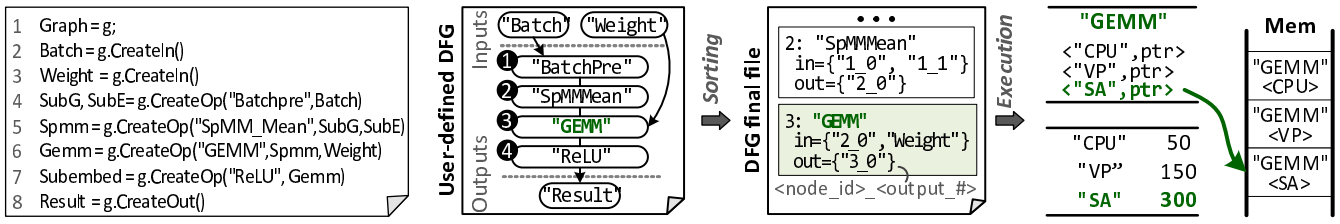
Table 3: GraphRunner's metadata structure.

Handling computational graphs. DFG management interfaces of the CSSD library (`CreateIn()`, `CreateOut()` and `CreateOp()`) are used for explaining how C-operations are mapped to DFG's nodes and how their input and output parameters are connected together (Table 2).

Figure 10a shows how the users can create a DFG to implement a GCN inference service as an example. The input and output of this DFG is `Batch`, `Weight`, and `Result`. `BatchPre` is the first C-operation that takes Batch as its input (❶), and the result is forwarded to `SpMM_Mean` C-operation (❷), which performs GCN's average-base aggregation. Then, the result of `SpMM_Mean` is fed to GCN's transformation consisting of GEMM (having Weight) and ReLU C-operations (❸/❹). The final output should be `Result` in this DFG. Note that, ReLU is a function of MLPs, which prevents the exponential growth in the computation and vanishing gradient issue [40]. The user can write this DFG using our computation graph library as shown in Figure 10b. It declares `Batch` and `Weight` by calling `CreateIn()` (lines 2∼3). `BatchPre` (❶), `SpMM_Mean` (❷), `GEMM` (❸), and `ReLU` (❹) are defined through `CreateOp()`, which are listed in lines 4∼7.

GraphRunner then sorts the calling sequence of CSSD library interfaces and generates a markup file as shown in Figure 10c. This DFG final file includes a list of nodes, each defining its node sequence number, C-operation name, where the input(s) come from, and what the output(s) are. For example, the third node is `GEMM(3: "GEMM")`, and its inputs come from the second node's first output (`2_0`) as well as input node, `Weight` (`in={"2_0", "Weight"}`). This node generates one output only (`out={"3_0"}`).

Execution of DFG. The host can run CSSD with the programmed GNN by downloading the corresponding DFG and a given batch through `Run()` RPC. As shown in Figure 10d, GraphRunner's engine visits each node of the DFG and checks the node's C-operation name. For each node, the engine finds the set of C-kernels (matched with the C-operation name) by checking the operation table. It then refers to the device



(a) Example of DFG programming. (b) Example of DFG. (c) DFG file generation. (d) Execution.

Figure 10: Overview of reconfigurable software framework (GraphRunner).

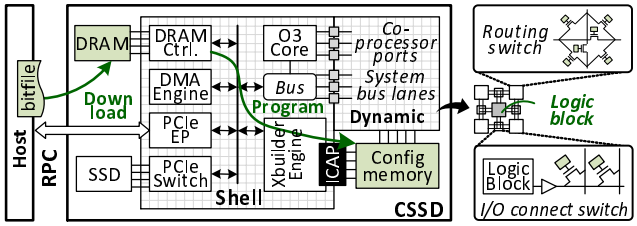


Figure 11: Reconfigurable hardware.

table and selects the appropriate implementation among the retrieved C-kernels based on the device priority, assigned by `RegisterDevice()`. The engine de-refers the C-kernel's address pointer and calls it by passing the C-kernel's parameters, which can also be checked up with offloaded DFG's node information (e.g., `in={...}`). Note that, GraphRunner's engine performs these dynamic binding and kernel execution for all the nodes of DFG per GNN inference.

4.3 Managing Reconfigurable Hardware

As shown in Figure 11, XBuilder provides static logic at Shell logic that includes an out-of-order core, a DRAM controller, DMA engines, and a PCIe switch. This static logic is connected to User (dynamic) logic through a co-processor port such as RoCC [4] and system bus (e.g., TileLink [67]).

XBuilder exposes the boundary position of the FPGA logic die in the form of a design checkpoint file [81, 89]. The boundary is wire circuits that separate Shell and User logic, called *partition pin*. Since the static logic is fixed at the design time, we place the maximum number of co-processor ports and system bus lanes to the partition pin, which can be united with the hardware components fabricated in Shell logic. In addition, we locate an XBuilder hardware engine in Shell, which includes the internal configuration access port (ICAP [85, 86]). Note that, FPGA logic blocks are connected by many wires of routing switches and input/output connection switches. Since the switches maintain the status of connection in built-in FPGA memory, called *configuration memory*, we can reconfigure the FPGA hardware by reprogramming the connection states on the configuration memory. As the configuration memory should be protected from anonymous accesses, FPGA only allows the users to reprogram the configuration memory only through the primitive hardware port, ICAP. The users can simply call HolisticGNN's RPC interface, `Program` with their own hardware (partial) bitfile to reconfigure User logic. XBuilder copies the bitfile into CSSD's FPGA internal DRAM first, and then, it reconfigures User logic by programming the logic using the bitfile via ICAP. While User logic is being reconfigured, it would unfortunately be possible to make the static logic of Shell unable to operate appropriately. Thus, XBuilder ties the partition pin's wires (including a system bus) by using DFX decoupler IP [83, 89] and makes User logic programming separate from the working logic of Shell. In default, XBuilder implements Shell by locating an out-of-core processor and PCIe/memory subsystems that run GraphRunner and GraphStore. Figure 12 shows three example implementation views of our Shell and User logic. Shell logic locates an out-of-core processor and PCIe/memory subsystems that run GraphRunner and GraphStore. In this example, we program an open-source RISC-V CPU, vector processor, and systolic array. We will explain details of example implementations in Section 5.

5 Evaluation

Prototypes. While CSSD is officially released in storage communities [20, 63, 70], there is no commercially available device yet. We thus prototype a customized CSSD that employs a 14*nm* 730MHz FPGA chip [87, 88], 16GB DDR4-2400 DRAM [71], and a 4TB high-performance SSD [12] together within the same PCIe 3.0×4 subsystem [57] as shown in Figure 13. We prepare three sets of hardware accelerators for XBuilder's User logic; a multi-core processor (*Octa-HGNN*), large systolic array processors (*Lsap-HGNN*), and a heterogeneous accelerator having a vector processor and a systolic array (*Hetero-HGNN*), as shown in Figure 12. Octa-HGNN employs eight out-of-order (O3) cores and performs all GNN processing using multi-threaded software. Each O3 core is implemented based on open-source RISC-V [3, 100] having 160KB L1 and 1MB L2 caches. For Lsap-HGNN and Hetero-HGNN, we modify an open-source SIMD (Hwacha [47]) and systolic architecture (Gemmini [23]). In our evaluation, SIMD employs four vector units, and the systolic architecture employs 64 floating-point PEs with 128KB scratchpad memory. Note that, all these prototypes use the same software part of HolisticGNN (GraphStore, GraphRunner, and XBuilder) as it can handle the end-to-end GNN services over DFG.

GPU-acceleration and testbed. For a fair performance com-

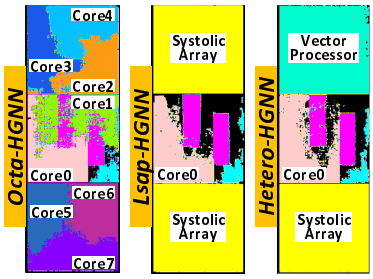


Figure 12: Shell/User prototypes.

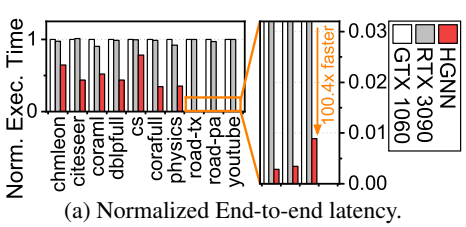
Figure 13: HolisticGNN prototype.

Host Setup
AMD Ryzen 3900X 2.2GHz, 12 cores DDR4-2666 16GB x4
GTX 1060 6GB [13] 1.8GHz, 1024 cores (10 SMs)
RTX 3090 24GB [14] 1.74GHz, 10496 cores (82 SMs)
FPGA Setup
Xilinx Virtex UltraScale+ [87] DDR4-2400 16GB x2
Storage
Intel SSD DC P4600 [12] 3D TLC NAND, 4TB

Table 4: Host and FPGA setup.

	Legend	Original Graph			Sampled Graph		
		Vertices	Edges	Feature Size	Vertices	Edges	Feature Length
Small (< 1M Edges)	chmleon [61]	2.3K	65K	20 MB	1,537	7,100	2326
	citeseer [93]	2.1K	9K	29 MB	667	1,590	3704
	coraml [93]	3.0K	19K	32 MB	1,133	2,722	2880
	dblpfull [93]	17.7K	123K	110 MB	2,208	3,784	1639
	cs [66]	18.3K	182K	475 MB	3,388	6,236	6805
	corafull [93]	19.8K	147K	657 MB	2,357	4,149	8710
	physics [66]	34.5K	530K	1,107 MB	4,926	8,662	8415
Large (>3M Edges)	road-tx [48]	1.39M	3.84M	23.1 GB	517	904	4353
	road-pa [48]	1.09M	3.08M	18.1 GB	580	1,010	4353
	youtube [48]	1.16M	2.99M	19.2 GB	1,936	2,193	4353
	road-ca [48]	1.97M	5.53M	32.7 GB	575	999	4353
	wikitalk [48]	2.39M	5.02M	39.8 GB	1,768	1,826	4353
	ljournal [48]	4.85M	68.99M	80.5 GB	5,756	7,423	4353

Table 5: Graph dataset characteristics.



(a) Normalized End-to-end latency.

Legend	GTX 1060
chmleon	140 ms
citeseer	162 ms
coraml	166 ms
dblpfull	323 ms
cs	618 ms
corafull	1233 ms
physics	2335 ms
road-tx	426732 ms
road-pa	332391 ms
youtube	341035 ms

(b) Latency.

Figure 14: End-to-end latency comparison.

parison, we also prepare two high-performance GPUs, *GTX 1060* and *RTX 3090*. While GTX 1060's 10 streaming multiprocessors (SMs) operate at 1.8GHz with 6GB DRAM, RTX 3090 employs 82 SMs working at 1.7 GHz with 24GB DRAM. To enable GNN services, we use deep graph library (DGL) 0.6.1 [74] and TensorFlow 2.4.0 [1], which use CUDA 11.2 and cuDNN 8.2 for GPU acceleration. DGL accesses the underlying SSD via the XFS file system to pre-/processing graphs. The testbed uses a 2.2GHz 12-core processor with DDR4-2666 64GB DRAM and a 4TB SSD (same with the device that we used for CSSD prototype), and connect all GPUs and our CSSD prototype. The detailed information of our real evaluation system is shown in Table 4.

GNN models and graph datasets. We implement three popular GNN models, GCN [42], GIN [90], and NGCF [75], for both GPUs and CSSD. We also select 14 real-graph datasets (workloads) from LBC [45], MUSAE [61], and SNAP [48]. Since the workloads coming from SNAP [48] do not provide the features, we generate the features based on the feature length that the prior work (pinSAGE [95]) uses (4K). The important characteristics of our graph datasets and workloads are described in Table 5. Note that, the workloads that we listed in Table 4 is sorted in ascending order of their graph size. For better understanding, we summarize the characteristics for graph before batch preprocessing (*Original Graph*) and after batch preprocessing (*Sampled Graph*).

5.1 End-to-end Performance Comparisons

Overall latency. Figure 14a compares the end-to-end inference latency of GTX 1060, RTX 3090, and our HolisticGNN (*HGNN*) using the heterogeneous hardware acceleration. For better understanding, the end-to-end latency of RTX 3090 and HGNN is normalized to that of GTX 1060. The actual latency value of GTX 1060 is also listed in Table 14b. We use GCN as representative of GNN models for the end-to-end performance analysis; since we observed that the pure inference computing latency only accounts for 1.8% of total latency, the performance difference among the GNN models that we tested are negligible in this analysis (<1.1%). We will show the detailed inference latency analysis on the different GNN models in Section 5.2.

One can observe from the figure that HGNN shows 7.1× and 7.0× shorter end-to-end latency compared to GTX 1060 and RTX 3090 across all the graph datasets except for `road-ca`, `wikitalk`, and `ljournal`. Note that both GTX 1060 and RTX 3090 cannot execute such large-scale graphs

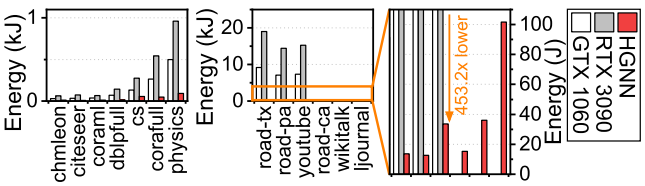


Figure 15: Estimated energy consumption comparison.

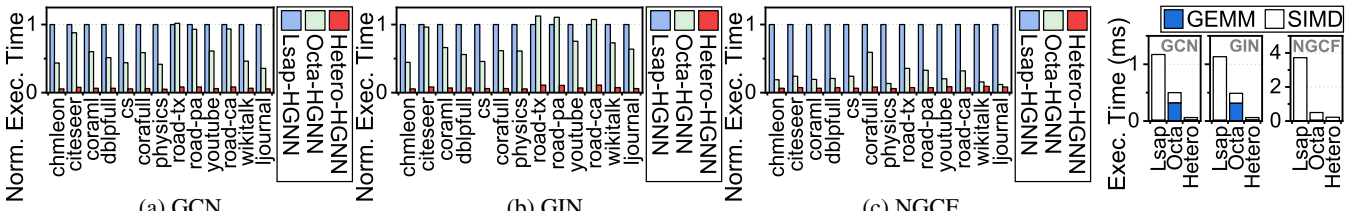
due to the out-of-memory issue, and thus, we exclude them in this comparison. Specifically, for the small graphs (<1M edges), HGNN outperforms GPUs by 1.69×, on average. This performance superiority of HGNN becomes higher when we infer features on large-scale graphs (>3M edges), which makes HGNN 201.4× faster than GTX 1060 and RTX 3090, on average. Even though the operating frequency and computing power of GTX 1060 and RTX 3090 are much better than HGNN, most of data preprocessing for both graphs and batches are performed by the host, and its computation is involved in storage accesses. This in turn makes the end-to-end inference latency longer. In contrast, HGNN can preprocess graphs in parallel with the graph updates and prepare sampled graphs/embeddings directly from the internal SSD, which can successfully reduce the overhead imposed by preprocessing and storage accesses. We will dig deeper into the performance impact of preprocessing/storage (GraphStore) and hardware accelerations (XBuilder) shortly.

Energy consumption. Figure 15 analyzes the energy consumption behaviors of all three devices we tested. Even though GTX 1060 and RTX 3090 show similar end-to-end latency behaviors in the previous analysis, RTX 3090 consumes energy 2.04× more than what GTX 1060 needs because it has 8.2× and 4× more SMs and DRAM, respectively. In contrast, HGNN exhibits 33.2× and 16.3× better energy consumption behaviors compared to RTX 3090 and GTX 1060, on average, respectively. Note that, HGNN processes large-scale graphs by consuming as high as 453.2× less energy than the GPUs we tested. This is because, in addition to the latency reduction of HGNN, our CSSD consumes only 111 Watts at the system-level thanks to the low-power computing of FPGA (16.3 Watts). This makes HGNN much more promising on GNN computing compared to GPU-based acceleration approaches. Note that, RTX 3090 and GTX 1060 consume 214 and 447 Watts at the system-level, respectively.

5.2 Pure Inference Acceleration Comparison

Figure 16 shows the pure inference performance of Hetero-HGNN and Octa-HGNN, normalized to Lsap-HGNN; before analyzing the end-to-end service performance, we first compare HolisticGNN itself different User logic here.

One can observe from this figure that, even though systolic arrays are well optimized for conventional DL such as CNN and RNN, Lsap-HGNN exhibits much worse performance than software-only approach. For all the graph datasets that we tested, Octa-HGNN exhibits shorter GNN inference latency compared to Lsap-HGNN by 2.17×, on average. This is crystal clear evidence that the conventional DL hardware



(a) GCN. (b) GIN. (c) NGCF.

Figure 16: Inference latency comparison.

Figure 17: Breakdown.

acceleration is not well harmonized with GNN inference services. Since the computation of aggregation is involved in traversing the graph data, the systolic arrays (bigger than any hardware logic that we tested) cannot unfortunately accelerate the inference latency. In contrast, Octa-HGNN processes the aggregation (including transformation) over multi-processing with many cores in User logic. As shown in Figure 16c, this phenomenon is more notable on the inference services with NGCF (4.35× faster than Lsap-HGNN) because NGCF has more heavier aggregation (similarity-aware and element-wise product explained in Section 2.1).

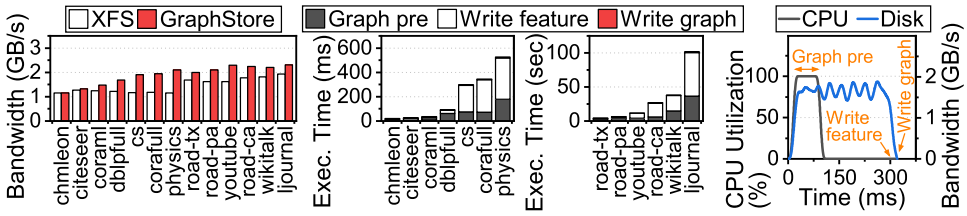
However, the performance of Octa-HGNN is also limited because matrix computation on dense embeddings (GEMM) is not well accelerated by its general cores. In contrast, Hetero-HGNN has both SIMD and systolic array units, which are selectively executed considering the input C-kernel, such that Hetero-HGNN shortens the inference latency of Octa-HGNN and Lsap-HGNN by 6.52× and 14.2×, on average, respectively. Figure 17 decomposes the inference latency of three HGNN that we tested into SIMD and GEMM for a representative workload, physics. As shown in figure, Lsap-HGNN mostly exhibits GEMM as its systolic arrays accelerate the transformation well, but its performance slows down due to a large portion of SIMD. The latency of Octa-HGNN suffers from GEMM computation, which accounts for 34.8% of its inference latency, on average. As Hetero-HGNN can accelerate both SIMD and GEMM, it successfully shortens the aggregation and transformation for all GNN models that we tested. This is the reason why we evaluate the end-to-end GNN latency using Hetero-HGNN as a default hardware acceleration engine in the previous section.

5.3 Performance Analysis on GraphStore

Bulk operations. Figures 18a and 18b show the bandwidth and latency of GraphStore's bulk operations. While the GPU-enabled host system writes the edge array and corresponding embeddings to the underlying SSD through its storage stack, GraphStore directly writes the data to internal storage without any storage stack involvement. This does not even exhibit data copies between page caches and user-level buffers, which in turn makes GraphStore exposes performance closer to what the target SSD actually provides. As a result, GraphStore shows 1.3× better bandwidth on graph updates compared to conventional storage stack (Figure 18a). More importantly, GraphStore hides the graph preprocessing overhead imposed by converting the input dataset to the corresponding adjacency list with the update times of heavy embeddings. We also show how much the embedding update (`Write feature`) can hide the latency of graph preprocessing (`Graph pre`) in Figure 18b. Since `Write feature` in the figure only shows the times longer than `Graph pre`, we can observe that GraphStore can make `Graph pre` completely invisible to users. For better understanding, we also perform a time series analysis of `cs` as an example of other workloads, and the results are shown Figure 18c. The figure shows the dynamic bandwidth in addition to the per-task utilization of Shell's simple core. As shown in the figure, GraphStore starts the preprocessing as soon as it begins to write the embeddings to the internal SSD. `Graph pre` finishes at 100*ms* while `Write feature` ends at 300*ms*. Thus, `Write feature` is performed with the best performance of the internal SSD (around 2GB/s). Note that, even though writing the adjacency list `Write graph` is performed right after `Write feature` (Figure 18b), it is almost invisible to users (Figure 18c) as the size of graph is much smaller than the corresponding embeddings (357.1×, on average).

Batch preprocessing (Get). Figure 19 shows batch preprocessing, which is the only task to read (sub)graphs from the storage in the end-to-end viewpoint; node sampling and embedding lookup use `GetNeighbor()` and `GetEmbed()`, respectively. In this evaluation, we compare batch preprocessing performance of GPU-enabled host and CSSD using `chmleon` and `youtube` each being representative of small and large graphs. For the earliest batch preprocessing, GraphStore performs batch preprocessing 1.7× (`chmleon`) and 114.5× (`youtube`) faster than that of the GPU-enabled host, respectively. Even though GraphStore is working at a lower frequency (3× than the host CPU), `GetNeighbor()` and `GetEmbed()` are much faster because the graph data has been already converted into an adjacency list at the graph update phase. In contrast, the host needs to process the graph data at the first batch, such that node sampling and embedding lookup can find out the appropriate targets. After the first batch, both cases, mostly accessing the neighbors and the corresponding embeddings are processed in memory thereby showing sustainable performance. Note that, even though we showed the batch preprocessing performance for only `chmleon` and `youtube` (due to the page limit), this performance trend is observed across all the workloads that we tested.

Mutable graph support (Unit operations). Since there is no publicly available dataset for mutable graph support, we evaluate the unit operations (requested by the host to CSSD) by processing historical DBLP datasets [30]. The top of Figure 20 shows the number of per-day add and delete operations



(a) Peak write bandwidth. (b) Latency breakdown. (c) Timeline of cs.

Figure 18: Performance analysis of GraphStore bulk operations.

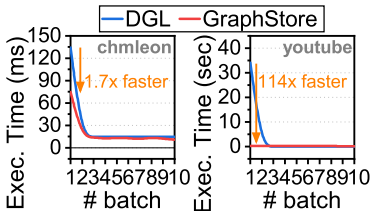


Figure 19: Multiple batches.

for the past 23 years (1995∼2018), and its bottom shows the corresponding per-day (accumulated) latency of GraphStore. The workload adds 365 new nodes and 8.8K new edges into GraphStore, and deletes 16 nodes and 713 edges per day, on average. As shown in Figure, GraphStore exhibits 970*ms* for per-day updates, on average, and the accumulated latency in the worst case of GraphStore is just 8.4 sec, which takes reasonably short in the workload execution time (0.01%).

6 Related Work and Discussion

There are many studies for in-storage processing (ISP) [25, 34, 43, 46, 58, 65], including DL accelerating approaches such as [51, 53, 76]. All these studies successfully brought significant performance benefits by removing data transferring overhead. However, these in-storage, smart storage approaches require fully integrating their computations into an SSD, which unfortunately makes the data processing deeply coupled with flash firmware and limited to a specific computing environment that the storage vendor/device provides. These approaches also use a thin storage interface to communicate with the host and the underlying SSD, which require a significant modification of application interface management. More importantly, all they are infeasible to accelerate GNN computing, which contains both graph-natured processing and DL-like dense computing operations.

On the other hand, architectural research [5, 50, 91] focuses on accelerating GNN core over a fixed hardware design such as vector units and systolic processors. While this simulation-based achieves the great performance benefit on GNN inference, they are ignorant of performance-critical components such as graph preprocessing and node sampling. These simulation-based studies also assume that their accelerator can have tens of hundreds of preprocessing elements (PEs), which may not be feasible to integrate into CSSD because of the hardware area limit. In contrast, HolisticGNN accelerates GNN-related tasks from the beginning to the end near storage, and its real system implementation only contains 64 PEs for the GNN inference acceleration.

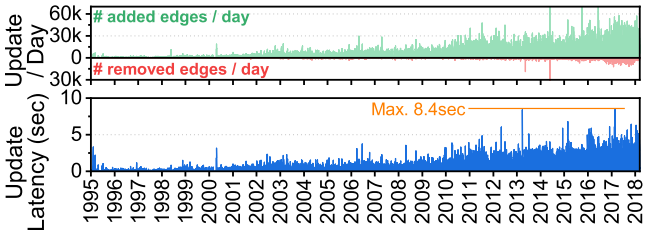


Figure 20: GraphStore update performance.

Lastly, there are FPGA approaches to deep learning accelerations [26, 77, 97]. Angel-Eye [26] quantizes data to compress the original network to a fixed-point form and decrease the bit width of computational parts. A frequency-domain hybrid accelerator [97] applies discrete Fourier transformation methods to reduce the number of multiplications of convolutions. On the other hand, a reconfigurable processing array design [77] tries to increase the operating frequency of any target FPGA in order to build a high throughput reconfigurable processing array. These studies are unfortunately not feasible to capture the GNN acceleration, and cannot eliminate the preprocessing overhead imposed by graph-natured complex computing near storage. Note that, it would be possible to use cross-platform abstraction platforms, such as OpenCL [69] or SYCL [60], rather than using RPC. OpenCL/SYCL is excellent for managing all hardware details at a very low-level, but they can bump up the complexity of what users need to control. For example, users should know all heterogeneities of reconfigurable hardware for the end-to-end GNN acceleration and handle CSSD's memory space over OpenCL/SYCL.

7 Conclusion

We propose HolisticGNN that provides an easy-to-use, near-storage inference infrastructure for fast, energy-efficient GNN processing. To achieve the best end-to-end latency and high energy efficiency, HolisticGNN allows users to implement various GNN algorithms close to the data source and execute them directly near storage in a holistic manner. Our empirical evaluations show that the inference time of HolisticGNN outperforms GNN inference services using high-performance modern GPUs by 7.1× while reducing energy consumption by 33.2×, on average.

Acknowledgements

This research is supported by Samsung Research Funding & Incubation Center of Samsung Electronics (SRFC-IT2101-04). This work is protected by one or more patents, and Myoungsoo Jung is the corresponding author. The authors would like to thank the anonymous reviewers for their comments and suggestions. The authors also thank Raju Rangaswami for shepherding this paper.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[3] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 40(4):10–21, 2020.

[4] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.

[5] Adam Auten, Matthew Tomei, and Rakesh Kumar. Hardware Acceleration of Graph Neural Networks. In *57th Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[6] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro*, 34(4), 2014.

[7] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational Inductive Biases, Deep Learning, and Graph Networks. *arXiv preprint arXiv:1806.01261*, 2018.

[8] Chi Chen, Weike Ye, Yunxing Zuo, Chen Zheng, and Shyue Ping Ong. Graph Networks as a Universal Machine Learning Framework for Molecules and Crystals. *Chemistry of Materials*, 31(9):3564–3572, 2019.

[9] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[10] Wonil Choi, Mohammad Arjomand, Myoungsoo Jung, and Mahmut Kandemir. Exploiting Data Longevity for Enhancing the Lifetime of Flash-based Storage Class Memory. In *2017 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2017.

[11] John Colgrove, John D Davis, John Hayes, Ethan L Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[12] Intel Corporation. Intel SSD DC P4600 Series. https://ark.intel.com/content/www/us/en/ark/products/96998/intel-ssd-dc-p4600-series-4-0tb-12-height-pcie-3-1-x4-3d1-tlc.html.

[13] NVIDIA Corporation. GeForce GTX 1060. https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1060/.

[14] NVIDIA Corporation. GeForce RTX 3090. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090/.

[15] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems 29 (NIPS)*, 2016.

[16] Google Developers. Protocol Buffers. https://developers.google.com/protocol-buffers.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using Non-volatile Memory for Storing Deep Learning Models. *arXiv preprint arXiv:1811.05922*, 2018.

[19] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. Pixie: A system for recommending 3+ billion items to 200+ million users in

real-time. In *Proceedings of the 2018 world wide web conference*, 2018.

[20] Samsung Electronics. Samsung SmartSSD. https://samsungsemiconductor-us.com/smartssd-archive/pdf/SmartSSD_ProductBrief_13.pdf.

[21] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference*, pages 417–426, 2019.

[22] Matthias Fey and Jan E. Lenssen. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[23] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *58th Design Automation Conference (DAC)*, 2021.

[24] gRPC Authors. gRPC. https://grpc.io/.

[25] 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 *43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[26] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-Eye: A Complete Design Flow for Mapping CNN onto Embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2017.

[27] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30 (NIPS)*, pages 1025–1035, 2017.

[28] William L Hamilton, Rex Ying, and Jure Leskovec. Representation Learning on Graphs: Methods and Applications. *arXiv preprint arXiv:1709.05584*, 2017.

[29] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation. In *43rd International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 639–648, 2020.

[30] Oliver Hoffmann and Florian Reitz. hdblp: Historical Data of the DBLP Collection. https://doi.org/10.5281/zenodo.3051910, May 2019.

[31] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 1991.

[32] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 1989.

[33] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating Graph Sampling for Graph Machine Learning using GPUs. In *16th European Conference on Computer Systems (EuroSys)*, pages 311–326, 2021.

[34] 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. *2016 Proceedings of the VLDB Endowment (PVLDB)*, 9(12):924–935, 2016.

[35] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. Grafboost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[36] Hoeseung Jung, Sanghyuk Jung, and Yong Ho Song. Architecture Exploration of Flash Memory Storage Controller through a Cycle Accurate Profiling. *IEEE Transactions on Consumer Electronics*, 57(4):1756–1764, 2011.

[37] Myoungsoo Jung, Wonil Choi, Miryeong Kwon, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut Taylan Kandemir. Design of a host interface logic for gc-free ssds. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[38] Myoungsoo Jung, Ramya Prabhakar, and Mahmut Taylan Kandemir. Taking garbage collection overheads off the critical path in SSDs. In *International Conference on Distributed Systems Platforms and Open Distributed Processing (USENIX Middleware)*, 2012.

[39] Kang, Yangwook and Kee, Yang-suk and Miller, Ethan L. and Park, Chanik. Enabling cost-effective data processing with smart ssd. In *29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2013.

[40] Bekir Karlik and A Vehbi Olgac. Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 2011.

[41] Kimberly Keeton, David A Patterson, and Joseph M Hellerstein. A case for intelligent disks (IDISKs). *ACM SIGMOD Record*, 27(3):42–52, 1998.

[42] Thomas N Kipf and Max Welling. Semi-supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907*, 2016.

[43] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading Communication with Computing Near Storage. In *50th International Symposium on Microarchitecture (MICRO)*, pages 219–231. IEEE, 2017.

[44] Dongup Kwon, Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim. A Fast and Flexible Hardware-based Virtualization Mechanism for Computational Storage Devices. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 729–743, 2021.

[45] Machine learning research group from the University of Maryland. Link-based classification project.

[46] Young-Sik Lee, Luis Cavazos Quero, Youngjae Lee, Jin-Soo Kim, and Seungryoul Maeng. Accelerating External Sorting via On-the-fly Data Merge in Active SSDs. In *6th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[47] Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and K Asanovic. The Hwacha Microarchitecture Manual, Version 3.8. Technical Report UCB/EECS-2015-263, EECS Department, University of California, Berkeley, 2015.

[48] Jure Leskovec and Rok Sosič. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[49] Jia Li, Yu Rong, Hong Cheng, Helen Meng, Wenbing Huang, and Junzhou Huang. Semi-supervised Graph Classification: A Hierarchical Graph Perspective. In *The World Wide Web Conference*, pages 972–982, 2019.

[50] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks. *IEEE Transactions on Computers (TC)*, 2020.

[51] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A Deep Learning Engine for In-Storage Data Retrieval. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 395–410, 2019.

[52] Xuan Lin, Zhe Quan, Zhi-Jie Wang, Tengfei Ma, and Xiangxiang Zeng. KGNN: Knowledge Graph Neural Network for Drug-Drug Interaction Prediction. In *29th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 380, pages 2739–2745, 2020.

[53] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia De Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. DeepStore: In-storage Acceleration for Intelligent Queries. In *52nd International Symposium on Microarchitecture (MICRO)*, pages 224–238, 2019.

[54] Diego Marcheggiani and Ivan Titov. Encoding Sentences with Graph Convolutional Networks for Semantic Role Labeling. *arXiv preprint arXiv:1703.04826*, 2017.

[55] Muthukumar Murugan and David HC Du. Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead. In *27th International Conference on Mass Storage Systems and Technologies (MSST)*, 2011.

[56] Yangyang Pan, Guiqiang Dong, and Tong Zhang. Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance. In *9th Conference on File and Storage Technologies (FAST)*, volume 11, pages 18–18, 2011.

[57] PCI-SIG. PCI Express Base Specification Revision 3.1a. https://members.pcisig.com/wg/PCI-SIG/document/download/8257.

[58] Luis Cavazos Quero, Young-Sik Lee, and Jin-Soo Kim. Self-sorting SSD: Producing sorted data inside active SSDs. In *31st International Conference on Mass Storage Systems and Technologies (MSST)*, 2015.

[59] Hannu Reittu and Ilkka Norros. On the power-law random graph model of massive data networks. *Performance Evaluation*, 2004.

[60] Ruyman Reyes and Victor Lomüller. Sycl: Single-source c++ accelerator programming. In *Parallel Computing: On the Road to Exascale*. IOS Press, 2016.

[61] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. Multi-scale Attributed Node Embedding. *Journal of Complex Networks*, 9(2):cnab014, 2021.

[62] 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)*, 2019.

[63] ScaleFlux. ScaleFlux Computational Storage. https://www.scaleflux.com/.

[64] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[65] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *11st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[66] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *arXiv preprint arXiv:1811.05868*, 2018.

[67] SiFive. TileLink Specification. https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13_tilelink_spec_1.8.1.pdf.

[68] SNIA. Computational Storage. https://www.snia.org/computational.

[69] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 2010.

[70] NGD Systems. NGD Systems Newport Platform. https://www.ngdsystems.com/technology/computational-storage.

[71] Micron Technology. MTA18ASF2G72PZ. https://www.micron.com/products/dram-modules/rdimm/part-catalog/mta18asf2g72pz-2g3.

[72] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph Attention Networks. *arXiv preprint arXiv:1710.10903*, 2017.

[73] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[74] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315*, 2019.

[75] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. Neural Graph Collaborative Filtering. In *42nd International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 165–174, 2019.

[76] 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 *26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 717–729, 2021.

[77] Ephrem Wu, Xiaoqian Zhang, David Berman, and Inkeun Cho. A High-Throughput Reconfigurable Processing Array for Neural Networks. In *27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[78] Guanying Wu and Xubin He. Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality. In *7th Eeuropean Conference on Computer Systems (EuroSys)*, pages 253–266, 2012.

[79] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2020.

[80] Xilinx. Computational Storage. https://www.xilinx.com/applications/data-center/computational-storage.html.

[81] Xilinx. Design Checkpoints. https://www.rapidwright.io/docs/Design_Checkpoints.html.

[82] Xilinx. Dynamic Function eXchange. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_1/ug909-vivado-partial-reconfiguration.pdf.

[83] Xilinx. Dynamic Function eXchange Decoupler. https://www.xilinx.com/content/dam/xilinx/support/documentation/ip_documentation/dfx_decoupler/v1_0/pg375-dfx-decoupler.pdf.

[84] Xilinx. SmartSSD Computational Storage Drive. https://www.xilinx.com/content/dam/xilinx/support/documentation/boards_and_kits/accelerator-cards/1_2/ug1382-smartssd-csd.pdf.

[85] Xilinx. UltraScale Architecture Configuration. https://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf.

[86] Xilinx. UltraScale Architecture Libraries Guide. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_1/ug974-vivado-ultrascale-libraries.pdf.

[87] Xilinx. Virtex UltraScale+ FPGA. https://www.xilinx.com/content/dam/xilinx/support/documentation/product-briefs/virtex-ultrascale-product-brief.pdf.

[88] Xilinx. Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics. https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf.

[89] Xilinx. Vivado Design Suite Tutorial: Dynamic Function eXchange. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2020_2/ug947-vivado-partial-reconfiguration-tutorial.pdf.

[90] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks? *arXiv preprint arXiv:1810.00826*, 2018.

[91] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. HyGCN: A GCN Accelerator with Hybrid Architecture. In *26th International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[92] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. Knightking: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[93] Zhilin Yang, William Cohen, and Ruslan Salakhudinov. Revisiting Semi-supervised Learning with Graph Embeddings. In *33rd International Conference on Machine Learning (ICML)*, pages 40–48. PMLR, 2016.

[94] Liang Yao, Chengsheng Mao, and Yuan Luo. Graph Convolutional Networks for Text Classification. In *33rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 7370–7377, 2019.

[95] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph Convolutional Neural Networks for Web-scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983, 2018.

[96] Jiaxuan You, Bowen Liu, Rex Ying, Vijay Pande, and Jure Leskovec. Graph Convolutional Policy Network for Goal-directed Molecular Graph Generation. *arXiv preprint arXiv:1806.02473*, 2018.

[97] Chi Zhang and Viktor Prasanna. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. In *2017 International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 35–44, 2017.

[98] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[99] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable Parallel Flash Firmware for Many-core Architectures. In *18th Conference on File and Storage Technologies (FAST)*, 2020.

[100] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV)*, May 2020.

[101] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction. *IEEE Transactions on Intelligent Transportation Systems*, 21(9):3848–3858, 2019.

Operational Characteristics of SSDs in Enterprise Storage Systems: A Large-Scale Field Study

Stathis Maneas
University of Toronto

Kaveh Mahdaviani
University of Toronto

Tim Emami
NetApp

Bianca Schroeder
University of Toronto

Abstract

As we increasingly rely on SSDs for our storage needs, it is important to understand their operational characteristics in the field, in particular since they vary from HDDs. This includes operational aspects, such as the level of write amplification experienced by SSDs in production systems and how it is affected by various factors; the effectiveness of wear leveling; or the rate at which drives in the field use up their program-erase (PE) cycle limit and what that means for the transition to future generations of flash with lower endurance. This paper presents the first large-scale field study of key operational characteristics of SSDs in production use based on a large population of enterprise storage systems covering almost 2 million SSDs of a major storage vendor (NetApp).

1 Introduction

Solid state drives (SSDs) have become a popular choice for storage systems over the past decade, increasingly replacing hard disk drives (HDDs). The performance and expected lifespan of an SSD are affected by operational characteristics in ways that are fundamentally different than for HDDs. For example, the lifespan is affected by write rates, as flash wears out, while performance is affected by the workload's read/write ratio due to the big differences between read and write latencies. Moreover, SSDs require background work, such as garbage collection and wear leveling, which generates write amplification and affects a drive's performance and lifespan. Usage characteristics, such as workload intensity (in particular write rates), the read/write ratio, and how full a drive is, affect how effectively a drive can manage these housekeeping tasks. Finally, drive specific details, such as whether a drive supports multi-stream writes or the amount of over-provisioned space, are expected to impact lifetime and performance as well.

As we increasingly rely on SSDs for our storage needs, it is important to understand what these operational characteristics look like for drives in the field and how they impact drives. Unfortunately, there are no large-scale field studies providing a comprehensive view of these characteristics for SSDs in the field. While there are a few recent field studies involving large-scale deployments, these have a different focus studying failure characteristics [30, 33, 36, 41, 46], fail-slow faults [13, 38], and performance instabilities [16] associated with SSDs in production.

In this paper, we present the first large-scale field study of several key operational characteristics of NAND-based SSDs in the field, based on NetApp's enterprise storage systems. Our study is based on telemetry data collected over a period of 4+ years for a sample of NetApp's total SSD population, which covers more than one billion drive days in total. Specifically, our study's SSD population comprises almost 2 million drives, which span 3 manufacturers, 20 different families (product batches, see detailed definition in §2), 2 interfaces (i.e., SAS and NVMe), and 4 major flash technologies, i.e., cMLC (*consumer-class*), eMLC (*enterprise-class*), 3D-TLC, and 3D-eTLC. Our data set is very rich, and includes information on usage, such as host reads and writes, total physical device writes, along with information on each drive's wear leveling and write amplification. Furthermore, our data contains each system's configuration, including all its RAID groups and the role of every drive within a RAID group (i.e., data or parity), among a number of other things.

We use this rich data set to answer questions, such as:

- What are the write rates that drives experience in production systems, and how close do drives get to reaching wear-out? What does this mean for future generations of flash with lower endurance limits?
- What are the write amplification factors that drives experience in production systems? How do those numbers compare to those reported in academic work?
- How effective are SSDs in production environments at wear leveling?
- How is write amplification affected by various factors, including FTL-related factors (e.g., drive model, firmware versions, over-provisioned space, support of multi-stream writes) and workload factors (e.g., write rates and read/write ratios, whether the drive is used as a cache or for persistent storage, whether the drive's role is data, parity or partitioned)?

2 Methodology

2.1 System Description

Our study is based on a rich collection of telemetry data from a large population of enterprise storage systems in production, comprising almost 2 million SSDs. The systems employ the WAFL file system [17] and NetApp's ONTAP operating system [37], while they run on custom Fabric-Attached Storage (FAS) hardware and use drives from multiple manufacturers. The systems are general-purpose, multi-tenant, and multi-protocol (NFS, FCP, iSCSI, NVMe_oF, S3), used by thousands of customers for very different applications (sometimes on the same node. In contrast to cloud data centers, which use mostly block- and object-based protocols, the majority of the systems in our data set use NFS (i.e., a file-based protocol). Applications running on our systems include file services, (enterprise) databases, financial technologies (fintechs), retail, electronic design automation (EDA) workloads, media entertainment, data analytics, artificial intelligence, and machine learning. Note though that storage vendors (e.g., NetApp) have no direct insight into what applications a customer is running on their systems, or who are the individual users of each system. Therefore, it is not trivial to break down our analysis by the type of application a system is running.

The operating system uses software RAID to protect against drive failures. Table 2 shows the breakdown of RAID group sizes in our systems, along with the breakdown of RAID schemes per range of RAID group sizes. As we observe, SSDs widely adopt RAID schemes protecting beyond single-device failures, especially with AFFs and larger arrays.

Our data set comprises systems with a wide range of hardware configurations, concerning CPU, memory, and total SSDs. Each system contains a large dynamic random access memory (DRAM) cache. Incoming write data is first buffered into the system's DRAM and then logged to non-volatile memory (NVRAM). Once the buffered (dirty) data is stored into persistent storage, during a consistency point (CP), it is then cleared from NVRAM and is (safely) retained in DRAM until it is overwritten by new data. We refer the reader to prior work for more information on the design and implementation details of NetApp systems [22, 23, 30, 31].

According to their usage, systems are divided into two different *types*: one that uses SSDs as a write-back cache layer on top of HDDs (referred to as **WBC**), and another consisting of flash-only systems, called **AFF** (All Flash Fabric-Attached-Storage (FAS)). An AFF system uses either SAS or NVMe SSDs, and is an enterprise end-to-end all-flash storage array. In WBC systems, SSDs are used as an additional caching layer that aims to provide low read latency and increased system throughput. Still, not all reads and writes are served from the SSD cache. Depending on the cache replacement policy, reads and writes can bypass the SSD cache and get served directly from the underlying HDD layer. For example, sequential user writes will typically get stored from DRAM

	Drive characteristics					Usage Characts.	
Drive Family	**Cap. (GB)**	**Flash Tech.**	**DWPD**	**PE Cycles Limit**	**OP**	**First Deploy-ment**	**Drive Power Years**
I - A	200	eMLC	10	10K	44%	Apr '14	5.69
	400					Apr '14	5.66
	800					Mar '14	5.01
	1600					Mar '14	5.49
I - B	400	eMLC	10	10K	44%	Dec '15	4.44
	800					Jan '16	4.15
	1600					Jan '16	4.25
I - C	400	eMLC	3	10K	28%	Jan '17	3.37
	800					Jan '17	3.06
	1600					Mar '17	3.32
	3800					Dec '16	2.87
I - D	3800	eMLC	1	10K	7%	Jul '17	2.82
I - E	800	3D-eTLC	1	7K	20%	Dec '18	1.67
	960					Dec '18	1.45
	3800					Dec '18	1.12
	7600					Jan '19	1.32
	15000					Jan '19	1.26
II - A	3840	3D-TLC	1	10K	7%	Jan '16	4.39
II - B	3800	3D-TLC	1	10K	7%	Oct '16	3.58
II - C	8000	3D-TLC	1	10K	7%	Sep '17	2.89
	15300					Sep '16	2.99
II - D	960	3D-TLC	1	10K	7%	Oct '16	3.37
	3800					Oct '16	3.57
II - E	400	3D-TLC	3	10K	28%	Dec '16	3.81
	800				28%	Jan '17	3.45
	3800				7%	Dec '16	3.75
II - F	960	3D-TLC	1	10K	7%	Dec '19	0.40
	3800					Mar '20	0.46
II - G	400	3D-TLC	3	10K	28%	Jan '16	4.17
	800					Feb '16	4.32
	1600					Jan '16	4.58
II - H	800	3D-TLC	3	10K	28%	Apr '18	1.91
	960		1		7%	Jan '18	1.77
	3800		1		7%	Jan '18	1.69
	8000		1		7%	May '18	1.63
	15000		1		7%	May '18	1.44
	30000		1		7%	Jul '18	1.43
II - I	800	cMLC	3	10K	28%	Sep '13	6.48
II - J	200	eMLC	10	30K	28%	Sep '13	6.92
	400					Sep '13	6.47
	800					Oct '13	6.74
II - K	400	eMLC	3	30K	28%	May '15	5.07
	800					Jul '15	4.98
	1600					Jun '15	5.11
III-A	960	3D-eTLC	1	7K	20%	Oct '19	0.69
	3800					Oct '19	0.54
	7600					Oct '19	0.57
II - X	3800	TLC	1	10K	7%	Aug '18	1.83
	7600					Jul '18	2.07
II - Y	3800	TLC	1	10K	7%	Jan '19	0.78
	7600					May '19	1.05
	15000					Dec '18	1.08
II - Z	3800	TLC	1	10K	7%	Jul '20	0.25
	7600					Jun '20	0.40
	15000					Jun '20	0.35

Table 1: *Summary statistics describing the key characteristics of the different drive families in our data set. The last three rows involve SSDs with an NVMe storage interface, whereas all the other rows involve SAS drives. The standard deviation in the drives' power-on years ranges from 0.05 to 0.98 for most drive families.*

Distribution of RAID Group Sizes		
Range	**AFF**	**WBC**
[3, 9]	16.21%	56.49%
[10, 19]	36.21%	28.98%
[20, 29]	47.58%	14.53%

Distribution of RAID schemes for AFF systems			
Scheme \ **Range**	**[3, 9]**	**[10, 19]**	**[20, 29]**
RAID-4 [39]	11.55%	0.99%	0.95%
RAID-DP [8]	88.43%	98.62 %	98.21%
RAID-TEC [11]	0.02%	0.39%	0.84%
Distribution of RAID schemes for WBC systems			
Scheme \ **Range**	**[3, 9]**	**[10, 19]**	**[20, 29]**
RAID-4 [39]	61.65%	21.50%	0.62%
RAID-DP [8]	38.18%	77.45%	97.55%
RAID-TEC [11]	0.17%	1.05%	1.83%

Table 2: *The top table shows the breakdown of RAID group sizes per system type, while the bottom table shows the breakdown of RAID schemes per range of RAID group sizes.*

directly to HDDs (as these can be executed efficiently on the HDDs and are also likely to pollute the SSD cache). Similarly, reads that result in an SSD cache miss are brought into DRAM and will be written to the SSD cache as well only if the cache replacement policy determines that the chance of reuse is high and it is worth evicting another block from the SSD cache.

In the remainder of the paper, we make use of the following terms (adapted from [3]):

• **Drive family**: A particular drive product, which may be shipped in various capacities, from one manufacturer, using a specific generation of SSD controller and NAND. Our data set contains 20 different families (denoted by a capital letter A–Z) from three different manufacturers (denoted as I, II, and III). We prepend the manufacturer's symbol to each drive family in order to explicitly associate each family with its manufacturer (e.g., I-A, II-C).

• **Drive model**: The combination of a drive family and a particular capacity. For instance, the I-B drive family comes in three models whose capacity is equal to 400, 800 or 1600GB.

• **Drive age**: The amount of time a drive has been in production since its ship date, rather than its manufacturing date.

The first six columns in Table 1 describe the key characteristics associated with the different drive families in our data set. Specifically, for each drive family, Table 1 includes all the corresponding drive models (in an anonymized form), along with the capacity, flash technology, endurance (specified in Drive Writes Per Day (DWPD) and the program-erase (PE) cycles limit), and over-provisioning (OP) associated with each model. As shown in Table 1, the SSD population in our study spans a large number of configurations that have been common in production settings over the last several years.

2.2 Data Collection and Description

Most systems in the field send telemetry data in the form of NetApp Active IQ® (previously called AutoSupport) bundles, which track a large set of system and device parameters (without containing copies of the customers' actual data). These bundles are collected and used for detecting potential issues.

Device and System Metrics	Sections
Host Write Rates/Read Rates	§3.1.1, §5
Annualized NAND Usage Rate	§3.1.2
Write Amplification Factor (WAF)	§3.2, §4
Avg/Max Erase Operations	§3.3
System Fullness	§3.4

Table 3: *The list of metrics analyzed in this study.*

Our study is based on mining and analyzing this rich collection of messages. Specifically, our data set is organized into 21 snapshots, each of which is generated after parsing the corresponding support messages collected at the following 21 points in time: Jan/Jun '17, Jan/May/Aug/Dec '18, Feb–Dec '19, Jan/Jul/Nov '20, and Mar '21. Each snapshot contains monitoring data for every system (and its drives). Table 3 shows all the metrics that are analyzed in this study.

3 What does SSD overall usage look like?

The performance and endurance of an SSD depend significantly on a number of operational characteristics. In this section, we use our field data to study four of the most important characteristics (described below). To the best of our knowledge, our work is the first to present details on these characteristics for a large-scale population of flash-based production systems.

• We begin by studying *write rates* (§3.1), as experienced by enterprise drives in the field (including both host and physical writes), as they significantly impact the lifetime of an SSD.

• We then examine the *write amplification factors (WAF)* observed by the SSDs in our study (§3.2), as write amplification is another major factor that can reduce a drive's endurance.

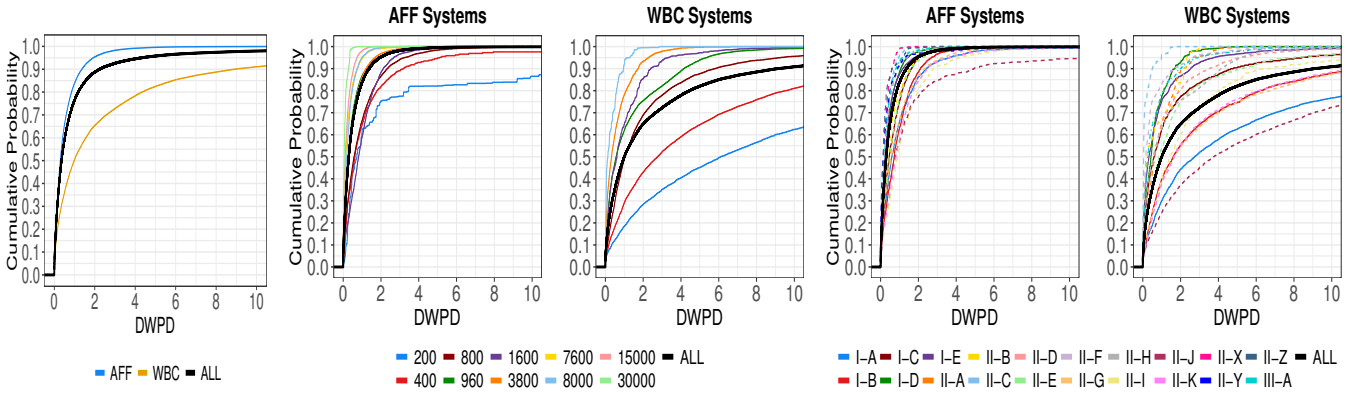
• Next, we look at how efficient drives are at *wear leveling* (§3.3), as it is a key mechanism that prolongs a drive's lifetime by preventing heavily used blocks from premature wear-out.

• Finally, we look at the *fullness* of systems (§3.4), i.e., what fraction of a system's total storage capacity is actually used. Fullness can significantly affect SSD operations, as a full system will trigger garbage collection more frequently and also has less free space to facilitate wear leveling and other housekeeping tasks.

3.1 Host Write Rates and NAND Usage Rates

A major concern when deploying SSDs are the write rates these devices will experience in the field, as erase operations wear out flash cells; therefore, the write intensity of a workload significantly affects the lifetime of flash-based SSDs. This is a particular concern looking forward since future generations of flash are expected to have an order of magnitude lower endurance than today's drives.

The goal of this section is to study a number of important aspects associated with write rates, as experienced by drives in enterprise systems, including how close drives get to their point of wear-out, how write rates vary across systems, differences between host and physical writes seen by a drive,



(a) *Per System Type.* (b) *Per Drive Capacity.* (c) *Per Drive Family.*

Figure 1: *Distribution of the drives' (normalized) host writes broken down by system type (1a), drive capacity (1b), and drive family (1c). In Figure 1c only, each line type corresponds to a different manufacturer.*

and an analysis of how feasible it would be to migrate workloads observed on today's systems to future generations of flash with lower program-erase (PE) cycle limits (i.e., the maximum number of PE cycles each SSD is rated for by its manufacturer).

3.1.1 Host Write Rates

We first look at write rates from the angle of *host writes*, i.e., the writes as they are generated by the applications and the storage stack running on top of the SSDs and measured and collected by the operating system (in contrast to physical NAND writes, which we examine in §3.1.2).

Because of the significance of write rates in the context of SSD endurance, drive manufacturers specify for each model its *Drive Writes Per Day* (DWPD), which is defined as the average number of times a drive's entire capacity can be written per day over its lifetime without wearing out prematurely. Typical DWPD numbers in drive datasheets are 1 and 3, and our population also includes some models with a DWPD of 10 (see Table 1 for all drive models in our study). However, trends associated with recent technologies suggest DWPD will drop below 1 in the future [34].

Understanding host writes is also important in other contexts. For example, when setting up workload generators or benchmarks for experimental system research, it is important to understand what realistic workloads one wants to emulate look like, and write rates are an important aspect of that.

Despite the significance of host writes and the fact that flash-drives have been routinely deployed at large scale for the past decade, we are not aware of any study reporting on host write rates in such systems. The goal of our measurements is to close this gap.

We present our results in Figure 1a. The black solid line in Figure 1a (left) shows the Cumulative Distribution Function (CDF) of the DWPD experienced by the drives across our entire population. In addition, the graph also breaks the results down into AFF (all flash) systems and WBC systems (where the flash is used as a write-back cache).

We make a number of high-order observations:

• The DWPD varies widely across drives: the median DWPD of the population is only 0.36, well below the limit that today's drives can sustain. However, there is a significant fraction of drives that experiences much higher DWPD. More than 7% of drives see DWPD above 3, higher than what many of today's drive models guarantee to support. Finally, 2% of drives see DWPD above 10, pushing the limits even of today's drive models with the highest endurance.

• When separating the data into AFF and WBC systems, we observe (probably not surprisingly) that WBC systems experience significantly higher DWPD. Only 1.8% of AFF drives see DWPD above 3 compared to a quarter of all WBC drives. The median DWPD is 3.4× higher for WBC than AFF, while the the 99th percentile is 10.6× higher.

• We note vast differences in DWPD across the WBC systems, including a long tail in the distribution. While the median is equal to 1, the drives in the 99th and the 99.9th %-ile experience DWPD of 40 and 79, respectively. What that means is that designers and operators of WBC systems need to be prepared for their systems to handle a vast range of DWPD values, including very high ones. It also means that optimally provisioning the drive endurance for a WBC system is much harder due to the huge range of DWPD in such systems.

Next, we perform a more fine-grained analysis of the DWPD experienced by different SSDs, by grouping drives based on their capacity (Figure 1b) and by drive family (Figure 1c). The reasoning is that different customers will purchase drives of different capacities depending on their applications' needs, so drives of different capacities likely see different types of workloads. Similarly, different drive families might be deployed in different types of systems that differ in the workloads that run on them.

• Turning to Figure 1b, we were surprised to see how significantly DWPDs vary depending on *drive capacity*. In particular, there is a very clear trend that smaller capacity drives see larger DWPD. While this trend is consistent for both

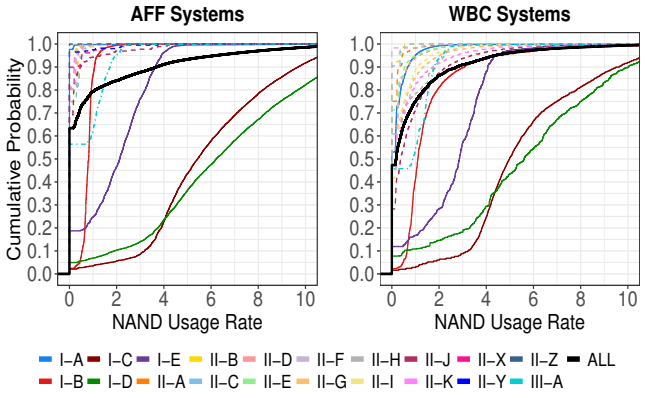


Figure 2: *Distribution of the drives' Annualized NAND Usage Rates broken down by drive family and system type. Each line type corresponds to a different manufacturer.*

AFF and WBC systems, the effect is particularly pronounced for WBC systems: here, the median DWPD for the smallest capacity drives is more than 100× higher than for the largest capacity drives (DWPD of 0.05 compared to 6). We note that this trend also holds when comparing the DWPD of different capacity drives within the same drive family, so the effect can be clearly attributed to drive capacity rather than family.

• Interestingly, we also observe significant differences in DWPD across drive families (Figure 1c). For example, for WBC systems, the median DWPD ranges from 0.04 to 3.75 across drive families. We also observe that for both AFF and WBC systems, it is the same drive families that experience higher DWPD than the average population.

3.1.2 NAND Usage Rates

The second metric associated with write operations focuses on *physical NAND device writes*. Physical device writes are typically higher than the raw host writes due to the device's background operations (e.g., garbage collection, wear leveling). For each drive model, manufacturers specify a limit on the number of physical writes it can tolerate before wearing out, in terms of the program-erase (PE) cycle limit (see Table 1 for the PE cycle limit of the drives in our population).

We are interested in studying the rate at which drives in the field approach their PE cycle limit, a question that is of particular concern as future generations of flash are expected to have significantly lower PE cycle limits [32]. Towards this end, for each drive, we determine the percentage of its PE cycle limit that it uses up per year, on average, a metric that we refer to as *Annualized NAND Usage Rate*:

$$Ann.\ NAND\ Usage\ Rate = \frac{\%\ of\ PE\ Cycle\ Limit\ Used\ So\ Far}{Power\text{-}On\ Years} \quad (1)$$

Figure 2 shows the NAND usage rates for AFF and WBC systems. The black solid line in each graph shows the CDF of the NAND usage rates across all the drives, irrespective of drive family. Since physical writes depend heavily on a drive's FTL (unlike host writes which are mostly driven by the applications), the figure also shows the CDF of NAND usage rates separately for each drive family.

We make the following key observations:

• Annualized NAND Usage Rates are generally low. The majority of drives (60% across the entire population) report a NAND usage rate of zero[1], indicating that they use less than 1% of their PE cycle limit per year. At this rate, these SSDs will last for more than 100 years in production without wearing out.

• There is a huge difference in NAND Usage Rates across drive families. In particular, drive families I-C, I-D, and I-E experience much higher NAND usage rates compared to the remaining population. These drive families do not report higher numbers of host writes (recall Figure 1c), so the difference in NAND usage rates cannot be explained by higher application write rates for those models.

We therefore attribute the extremely high NAND usage rates reported by I-C/I-D drives to other housekeeping operations which take place within the device (e.g., garbage collection, wear leveling, and data rewrite mechanisms to prevent retention errors [6]). We study this aspect in more detail in Section 3.2 and in Section 4, where we consider Write Amplification Factors (WAF).

• There is little difference in NAND usage rates of AFF systems and WBC systems. This is surprising given that we have seen significantly higher host write rates for WBC systems than for AFF systems. At first, we hypothesized that WBC systems more commonly use drives with higher PE cycle limits, so higher DWPD could still correspond to a smaller fraction of the PE cycle limit. However, we observe similar NAND usage rates for WBC systems and AFF systems, even when comparing specific drive families and models with the same PE cycle limit. Interestingly, as we will see in Section 3.2, the reason is that WBC systems experience lower WAF, which compensates for the higher host write rates.

Projections for next generation drives: We can use NAND usage rates to make projections for next generation QLC drives. Considering that endurance is estimated to be reduced for QLC drives [32], we are interested in determining how many SSDs in our data set could be replaced by a QLC SSD without wearing out within a typical drive lifetime of 5 years.

• If we assume that the PE cycle limit of QLC drives drops to 1K, then we find that the vast majority of our population (~95% of drives when excluding the two outlier models I-C and I-D) could have used QLC drives without wearing them out prematurely.

3.2 Write Amplification Factor (WAF)

The write amplification factor (WAF) plays a critical role, as the added writes due to garbage collection, wear leveling, and other SSD-internal housekeeping tasks, can negatively impact

[1] Unfortunately, the % of PE cycle limit used per SSD is reported as a truncated integer.

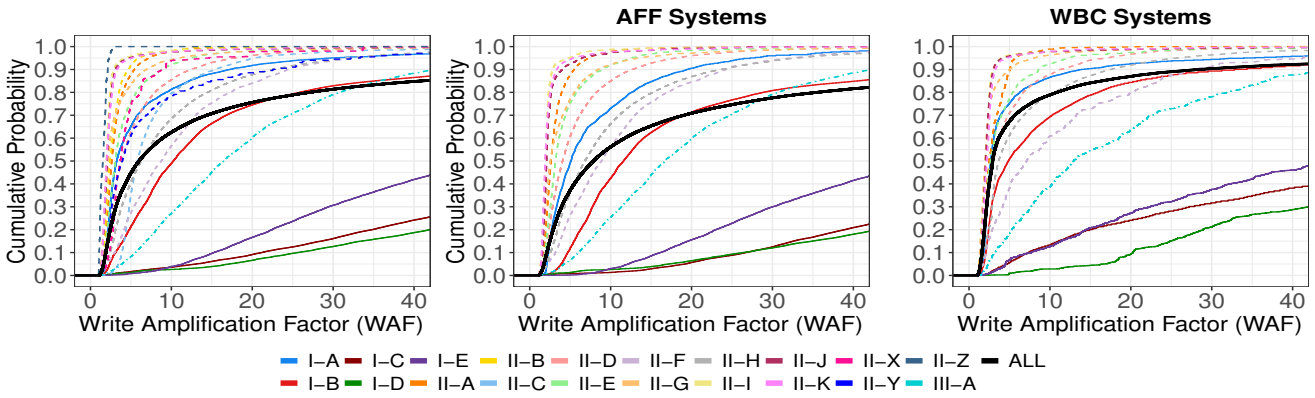


Figure 3: *Distribution of the drives' WAF broken down by drive family (left), along with both drive family and system type (middle and right). Each line type corresponds to a different manufacturer.*

both the endurance and performance of an SSD. It is therefore not surprising that a large body of work has been dedicated to reducing WAF and its impact, for example by optimizing FTLs in different ways [7, 14, 15, 18, 20, 25, 44, 47–49] or by making use of Multi-stream SSDs [4, 21, 40].

Unfortunately, despite the large body of work in industry and academia on WAF, we do not have a good understanding of how effective real drives in production systems are in controlling WAF. To the best of our knowledge, there is no large-scale field study reporting and analyzing WAF in production systems. The existing field studies that mention WAF for production systems are either limited to one particular type of application (financial services) [29] or are based on a small population of one flash technology (3D-TLC) [28]; both studies simply report an average WAF across their systems of 1.3 and 1.5, respectively (without any further analysis).

One goal of this paper is to improve our understanding of WAF in production systems. We begin in this section with some high-level statistics on WAF and then later in Section 4 study in more detail the impact of various factors on WAF.

The black solid line in Figure 3 (left) shows the distribution of WAF across all the drives in our population. In the same graph, we also show WAF broken down by drive family, as a drive's FTL affects WAF.

We make a number of high-level observations:

• For the vast majority of our SSDs, the WAF they experience is higher than the WAF of 1.3 observed in [29] (a field study reporting WAF numbers, but only in the context of financial services applications) and the WAF of 1.5 observed in [28] (based on a sample of 3D-TLC SSDs from Huawei's storage systems). Specifically, 98.8% and 96% of our SSDs observe a WAF larger than 1.3 and 1.5, respectively. This observation underlines the importance of field studies spanning a large range of systems with different applications and devices.

• The drives in our population span a huge range of WAF values. While the 10th percentile is only 2, the 99th percentile is 480. This motivates us to study the effect of several different factors on WAF. We start below with a high-level study of the role of the FTL and workloads, and continue with a more detailed study of factors impacting WAF in Section 4.

WAF and the FTL: As different drive families vary in their firmware, comparing the WAF across drive families provides insights into the relationship between the FTL and WAF.

• Figure 3 (left) shows that some drive families have *drastically* higher WAF than others. In particular, the I-C, I-D, and I-E families experience WAF that is an order of magnitude higher than that for most of the other drive families, with median WAF values of around 100 (!) in the case of I-C and I-D. Note that these drive families do not experience a different host write rate and we have no indication that they are being deployed in systems that tend to run different types of applications, so there is no obvious explanation due to workload characteristics. Also, differences in WAF persist even when we compare with drive families of the same age and capacity.

Upon closer inspection, we found that these particular models perform background work every time the SSD has idle cycles to spare, thereby consuming their PE cycles as a side effect. Interestingly, it seems that this background work is not due to garbage collection or wear leveling (the best studied contributors to WAF), but due to aggressive rewriting of blocks to avoid retention problems, where stored data is (periodically) remapped before the corresponding flash cells accumulate more retention errors than what can be corrected by error correction codes (ECC) [6].

We note that this unexpected effect drives up WAF not only for drives with extremely low utilization, but also for the busiest drives (e.g., top 5%) of the two outlier families.

In summary, the FTL has a huge impact on WAF.

WAF and workload: Our data also provides evidence of the impact of workload characteristics on WAF:

• First, we observe that there is significant variation in WAF even when comparing only drives within the same drive family (rather than across families). The 95th percentile of a drive family's WAF is often 9× larger than the corresponding median. These differences are likely due to different drives within a family being exposed to different workloads.

• Second, when we compare the WAF for AFF and WBC

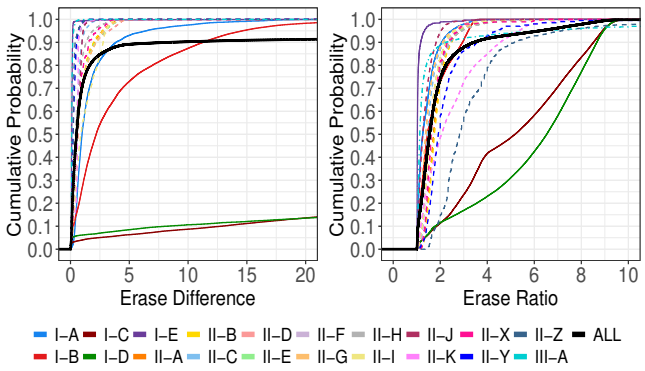


Figure 4: *Distribution of metrics associated with wear leveling, calculated based on the number of erase operations per block. Each line type corresponds to a different manufacturer.*

systems in Figure 3 (two right-most figures), we observe that for the same drive families, WBC systems experience significantly lower WAF than AFF systems, indicating that WBC workloads are more flash friendly. This results in an another interesting observation:

• Thanks to their lower WAF, WBC drives in our systems do not see a higher NAND usage rate than AFF systems, despite their higher DWPD (recall Figure 1c). This observation is significant, because the application of SSDs in caches is considered the most demanding, in terms of endurance requirements, and widely accepted best practices recommend to use only drives with the highest endurance for these applications. Our observations indicate that this might not always be necessary.

Comparison with simulation studies: Due to the dearth of field data on WAF, a number of authors have resorted to trace-driven simulation studies to explore WAF and how it is affected by various factors; therefore, it is interesting to compare the numbers we observe against those studies.

• The values reported for WAF in trace-driven simulation studies [5, 9, 10, 19, 42, 45] are at the low end of the WAF range we observe for AFF production systems, and even the *maximum* values reported in these studies fall only into the mid range (often below median) of the WAF values we observe. For example, the *highest WAF* in [45] is 2, in [42] it is 7, and in [5, 9, 10, 19] it is 12, which correspond to the 9th, 49th, and 62th percentile respectively, of the WAFs in our AFF population.

We draw two possible conclusions from this differences:

• A significant challenge that researchers in our community face is the lack of publicly available I/O traces from SSD-based storage systems. As a result, existing experimental work, including the simulation studies cited above, is based on traces that are i) based on HDD systems and ii) mostly relatively old (more than a decade for some popular traces). These traces might not be representative of today's workloads running on SSD-based systems and also do not cover aspects relevant to SSD-based systems (e.g., the TRIM command). As a community, it is important that we find more relevant traces to use as base to our work.

• The differences in WAF between our production systems and existing simulation studies also indicate that it is very difficult to reproduce all complexities and subtleties of modern FTLs in simulation.

3.3 Wear Leveling

A critical job of an SSD controllers is wear leveling, which aims to spread the erase operations evenly over all blocks on the device. This is important for multiple reasons. First of all, it serves to increase the device's lifetime by preventing frequently used blocks from wearing out prematurely. Second, it can help avoid performance problems, since blocks with higher erasure cycles are associated with higher error rates [12], and retries and other error-correction efforts can significantly add to latency.

Wear leveling is a difficult problem because real-world workloads are rarely uniform and commonly exhibit strong skew in per-block update frequencies. An ideal wear leveling mechanism distributes write operations in such a way so that all blocks within an SSD wear out at the same rate. At the same time, there is a delicate trade-off, as aggressive wear leveling will increase the number of write operations, thereby increasing WAF and overall drive wear-out.

In this section, we explore how effective modern FTLs are at wear leveling. Specifically, our data set contains the average number of times the blocks in a drive have been erased, along with the corresponding maximum value. Based on these values, we calculate two different metrics that characterize how evenly erase operations are distributed across all blocks:

The *Erase Ratio* is the ratio between the maximum and average number of erase operations per SSD:

$$Erase\ Ratio = \frac{Max.\ Erase\ Ops}{Avg.\ Erase\ Ops} \quad (2)$$

The *Erase Difference* is the absolute difference between maximum and the average number of erase operations normalized by the PE cycle limit:

$$Erase\ Difference = \frac{Max.\ Erase\ Ops - Avg.\ Erase\ Ops}{PE\ Cycle\ Limit}(\%) \quad (3)$$

Figure 4 shows the Erase Difference and Erase Ratio across our entire population (black solid line) and broken down by drive family. The ideal value of the Erase Ratio is 1, whereas the ideal value of the Erase Difference is 0. We make the following observations:

• Not surprisingly, wear leveling is not perfect. The median Erase Ratio is 1.55, indicating that the maximum block undergoes 55% more erase operations than the average block. 5% of the drives have an erase ratio larger than 6 meaning their maximum block wears out 6× faster than the average - that means when the maximum block has reached end of life the

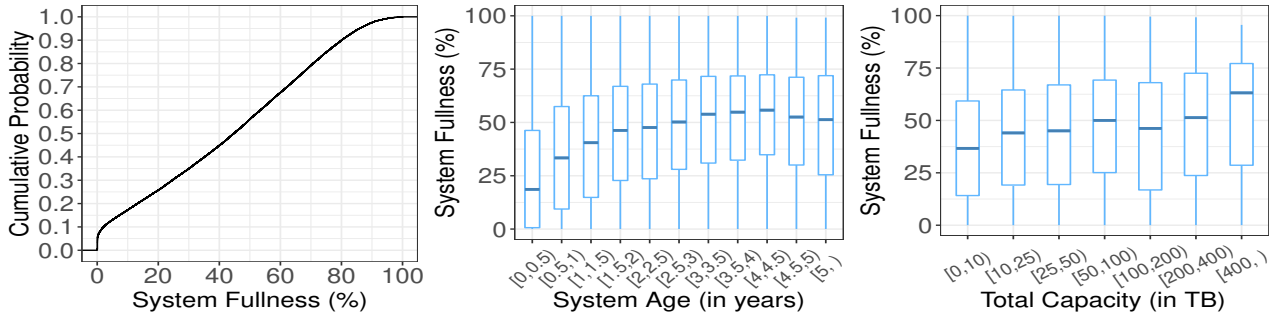


Figure 5: *Distribution of AFF systems' fullness.*

average block has only used 16% of its PE cycle limit.[2].

• There is a striking difference in the wear leveling metrics across drive families. The fact that I-C and I-D drives, for example, report significantly higher wear leveling metrics (despite having similar age, capacity, and DWPD to some other families) indicates that different firmware implementations take vastly different approaches to wear leveling. More generally, it seems that different *manufacturers* follow very different philosophies with respect to wear leveling: when looking at the Erase Difference metric, we see that the four families with the largest Erase Difference all belong to the same manufacturer (i.e., I).

• It is surprising that drive models I-C and I-D do a significantly worse job at wear leveling than other drive models, despite the fact that these two models experience higher WAF (recall §3.2). This means that the additional background work that these drives are performing is not contributing towards better wear leveling. Instead, we believe that the additional background work of those two drive families is because they *rewrite data more aggressively than others in order to avoid retention errors*. This is a very interesting observation, since data rewrite for retention errors has received much less attention than other sources of WAF (e.g., garbage collection, wear leveling). In fact, current SSD simulators and emulators (e.g., FEMU [26]) do not implement data rewrite for retention errors, and therefore do not capture this source of WAF.

3.4 Fullness

Another critical factor in the operation of an SSD-based storage is the system's *fullness*. We define *fullness* as the fraction of the drives' nominal capacity that is filled with valid data, i.e., the fraction of the Logical Block Address (LBA) space currently allocated. Fullness can affect the overall performance of a system, as it can impact the frequency of garbage collections, and also determines how much free room there is for operations like wear leveling. Also, fullness is of practical importance for capacity planning, as systems that run out of available space before the end of their lifetime need to be expanded with additional storage.

On the other hand, from the garbage collection's point of view, fullness denotes what fraction of blocks inside the drive are currently not programmable. This includes blocks containing valid data, but also blocks containing invalidated data which have not been erased yet. In our analysis, we focus only on the (allocated) LBA space.

In this section, we are interested in exploring what fullness looks like for enterprise storage systems, how it changes over a drive's lifetime, and how it varies as a function of factors such as drive capacity. Our study is the first to characterize this important system aspect for flash-based storage systems.

We begin with a high-level view of fullness by considering the CDF of fullness across the entire population of AFF systems, as shown in Figure 5 (left); we consider only AFF systems in our study of fullness, as the concept of fullness does not apply in the same way to WBC systems, which use SSDs only as a cache on top of HDDs.

• We observe that the average system is around 45% full, and the median is also around 45%, i.e., more than half of the storage capacity is free.

• The distribution of fullness across systems is roughly uniform. The CDF flattens only above 80%, i.e., values below 80% are all roughly equally likely, while values above 80% are relatively less common.

Next, in Figure 5 (middle), we look at how fullness changes over a system's lifetime. Understanding this aspect of fullness is relevant, for example, in the context of capacity planning.

• Maybe not surprisingly, system fullness increases with age. (Consider for example the median over time, as indicated by the dark link in the center of each box plot). However, the rate of increase is not uniform: fullness grows relatively fast over the first two years and stabilizes after that.

• Interestingly, despite the fact that generally fullness increases over time, there are some very young systems that are quite full and some old systems that are quite empty: slightly more than 5% of young systems (less than 1 year old) are more than 80% full, whereas 19% of old systems (more than 4 years old) are less than 25% full.

• An interesting observation from a capacity planning point of view is that systems who end up being full at the end of their life are also among the fullest systems early in their life. In other words, if a system has not used a significant amount of its physical space after its first couple of years in

[2]We do not have data on the minimum number of erase operations of a drive's blocks – naturally the difference between the minimum and maximum block would be even more pronounced.

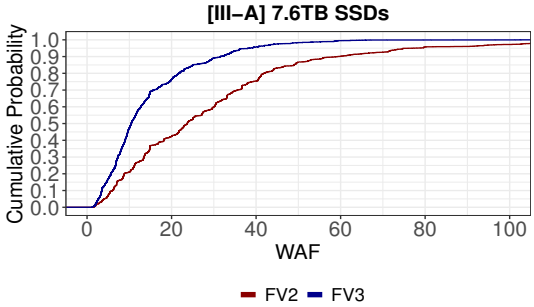


Figure 6: *WAF comparison between two firmware versions within the same drive family.*

production, its fullness will most probably remain (relatively) low in the future.

Given that systems vary hugely in their total capacity, ranging from tens of TBs to a couple of PBs, another interesting question is whether users of larger capacity systems actually make use of the additional capacity of their systems. Towards this end, Figure 5 (right) presents boxplots of system fullness, broken down by system capacity.

• Interestingly, we observe that system with *larger* total capacity tend to be *more* full: the largest systems are 1.7× fuller (in terms of median) than the other systems. This seems to indicate that customers who purchase larger capacity systems do indeed have larger capacity needs and are also better at predicting how much storage capacity they need.

Comparison with fullness reported for other types of systems: A seminal work by Agrawal et al. [1], published more than a decade ago, studied file system characteristics, including fullness, for personal desktop computers at Microsoft. They observed average fullness values ranging from 45–49% and a uniform distribution. This is quite similar to our observations – which is surprising given that their study looks at completely different types of systems (personal desktop computers using HDDs).

The only other work we found that reports on fullness in production systems is by Stokely et al. [43], which studies the usage characteristics of an HDD-based distributed file systems within a private cloud environment. Their results indicate that the fraction of the quota used by an average user of those systems is significantly larger than the levels of fullness we observe: on average users use 55% of their purchased quota; for the largest quota requests this number increases to 69%. The reason might be that it is easier for a user to increase their individual quota in a distributed storage system when running out of space, compared to increasing the physical capacity of an enterprise storage system. Therefore capacity planning for an enterprise storage system has to be more conservative.

4 Which factors impact WAF?

There are a number of factors that are commonly assumed to impact a drive's WAF, including the design of a drive's FTL, usage characteristics, how full the system is, along with the size of the drive's over-provisioned space. In this section, we try to shed more light on the impact of each of these factors on WAF, as experienced in production systems.

4.1 Flash Translation Layer (FTL)

This points to the importance of different design choices made by different FTL implementations. In Section 3.2, we have observed huge differences in WAFs across different drive families, even when controlling for other factors, such as drive capacity and DWPD.

In this section, we attempt a more fine-grained look at the impact of FTLs on WAF. Instead of comparing WAF across different drive families, we now look at different firmware versions within a given drive family.

Firmware version and WAF: We performed this study for several drive families, and discuss the results for drive family III-A, as a representative sample. The most common firmware versions for this drive family are versions FV2 and FV3. We see consistently across all capacities of this drive model that the more recent firmware version FV3 is associated with lower WAF than the earlier FV2 version.

For illustration, we present the CDF of WAFs for firmware versions FV2 and FV3 for the 7.6TB capacity model of drive family III-A in Figure 6. We chose this particular capacity because it offers the cleanest comparison, as the population of FV2 drives and the population of FV3 drives are quite similar with respect to other factors, such as DWPD, deployment time, and system type.

We observe a clear difference between the WAF of drives on firmware version FV2 versus version FV3. For example, both the median and the 90th percentile of WAF are around 2× larger for FV2 than the more recent FV3 version.

4.2 Workload Characteristics

Many aspects of workload characteristics can impact a drive's WAF. Unfortunately, the analysis of many of these aspects (e.g., sequentiality of writes and deletes, skewness of updates across blocks) would require block-level IO-traces, whose collection for production systems at large scale is infeasible.

Instead, we focus on the following five aspects of workload characteristics for which we were able to collect data: the first is write intensity as measured by drive writes per day (DWPD) seen by a drive. The second is the *role* of a drive within a RAID group[3]; we distinguish among *data* and *partitioned* drives, where each partition of the drive is part of a different RAID group and different partitions can play different roles in their RAID groups. We exclude *parity* drives due to insufficient data. The third and fourth factors are the drive capacity and drive interface (SAS vs. NVMe), respectively, as drives of different capacities and different interfaces will be used in different types of systems, which might be used for different types of workloads. The fifth factor is the read/write ratio of

[3] In our data set's RAID systems, parity blocks are not rotated.

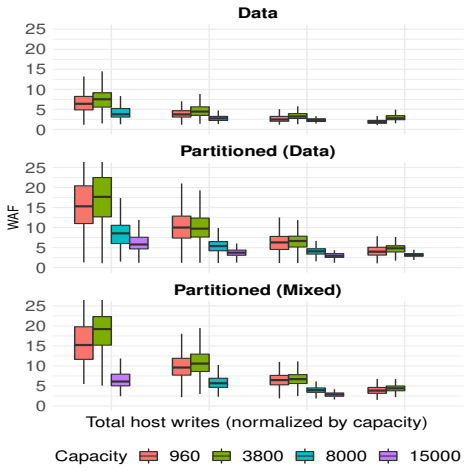


Figure 7: *Impact of different factors on WAF, while focusing on different roles within a single drive family.*

the workload.

Figure 7 shows WAF broken down by the first three aspects described above. We describe our observations below.

Drive Writes Per Day (DWPD) and WAF: We observe that consistently across different capacities and drive roles, *WAF decreases* as the number of *DWPD increases*. Median WAF is up to 4.4× higher for the drive populations with the lowest DWPD (left-most group of bars in Figure 7) than the group with the highest DWPD (right-most group of bars). This could suggest that SSDs operate more efficiently (in terms of background tasks and WAF) under higher write rates. It could also mean that some FTL background work is constant, i.e., not strongly dependent on DWPD; therefore, higher DWPD will reduce the effect of this constant work on the WAF ratio.

Drive role and WAF: We observe a significant difference in WAF depending on the drive *role*. In particular, the median WAF associated with *partitioned* SSDs is *significantly higher* (by up to 3X) than that for *data* SSDs. One possible explanation for the higher WAF of partitioned SSDs might be that they are forced to handle requests coming from different workloads with potentially different characteristics, thus experiencing a mixture of write patterns.

We do note that the difference across roles decreases as the number of (normalized) total host writes increases, suggesting that write rates have a stronger impact on WAF than its role.

Drive capacity and WAF: When we explore the impact of capacity, we observe that *higher-capacity* SSDs (i.e., 8TB and 15TB) experience *lower WAF* compared to the two smaller-capacities, for the same range of total host writes and the same drive role. In particular, their median WAF can be up to 2–3× smaller, with the difference being more pronounced when the amount of total host writes is low. Still, 3.8TB SSDs experience slightly higher WAF compared to 960GB SSDs, suggesting that smaller-capacity SSDs do not necessarily experience higher WAF (i.e., other factors have a stronger impact on WAF).

Drive interface and WAF: The workloads that customers

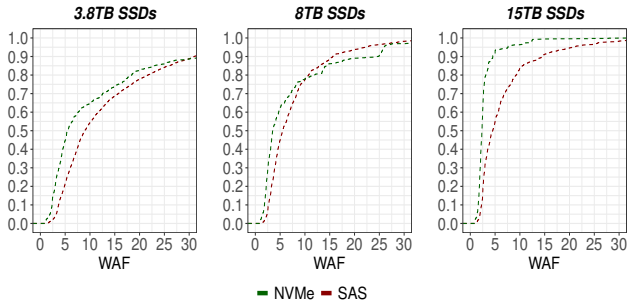


Figure 8: *WAF comparison between SAS and NVMe SSDs for different drive capacities.*

choose to run on NVMe drives tend to experience slightly smaller WAF than those on SAS drives. Results are shown in Figure 8, which compares the distribution of WAF as experienced by SAS and NVMe drives respectively, broken down by three different drive capacities. The populations of SAS and NVMe drives were chosen in such a way as to control for other factors, such as DWPD, total time in production, drive role, and system fullness. We removed the outlier drive families, for which we observed earlier an extremely high WAF, from the SAS population, so they do not bias our results.

Considering that NVMe SSDs make use of a similar FTL compared to SAS drives, we expect differences in WAF to come mostly from them being used differently. For instance, the NVMe technology is still relatively new and as a result, in our data set, the population of NVMe-based systems is smaller than the SAS-based population. The NVMe systems are mostly used by (a small set of) customers who are early adopters of this new technology. These customers, and their workloads, might be different from the average customers across the whole population. Therefore, the workloads experienced by NVMe and SAS drives can be quite different (for now); the difference will likely become less pronounced over time, as more customers move to NVMe-based systems.

Read/write ratios and WAF: We observe a positive correlation between a workload's R/W ratio and WAF. More precisely, we used the buckets of drives we created for Figure 7 (so that we control for capacity, write rates and drive role), and computed for each bucket the Spearman correlation coefficient between the R/W ratio and WAF for the drives in the bucket. The correlation coefficient is between 0.2 and 0.4 for most buckets, indicating a positive correlation.

4.3 Fullness

The fullness of a drive can affect how effectively it can manage its internal housekeeping tasks, such as garbage collection and wear leveling, especially when its total free space becomes (too) low.

We study the effect of fullness on WAF by dividing our population into drives that are more than 80% full and those that are less than 80% full, and comparing their WAF.

Interestingly, we observe *no significant differences* in the WAF experienced by the two sub-populations; in fact, SSDs which are more full experience (slightly) smaller WAF overall,

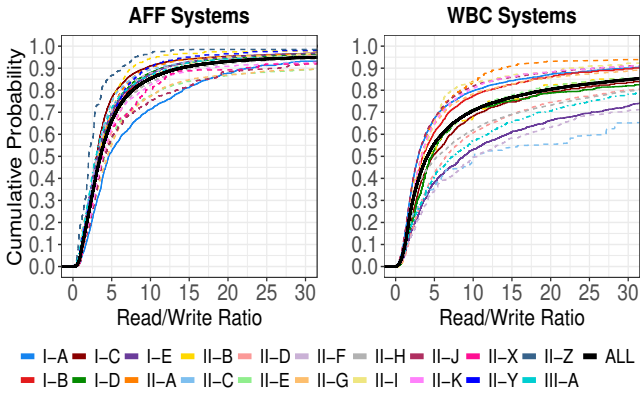


Figure 9: *Distribution of the drives' R/W ratios per system type. Each line type corresponds to a different manufacturer.*

suggesting that the drives' WAF is dominated by other factors than fullness, such as their firmware.

4.4 Over-provisioning (OP)

Another interesting question is whether WAF varies depending on the drives' amount of over-provisioning (OP). OP refers to the fraction of a drive's capacity that is reserved as spare capacity to improve a drive's wear leveling, garbage collection, and random write performance; thus, it is expected to help reduce WAF. In fact, prior work uses simulations to demonstrate the positive effect of higher OP on WAF [10, 19].

Common OP percentages for real drives are 7% and 28% and our SSD population includes drives with both OP percentages, allowing us to compare their effect on WAF.

Surprisingly, we observe that the drives with the *higher OP* (i.e., 28%) actually experience *higher*, rather than lower WAF. One possible explanation could be that many of the drives in our population are not very full (§3.4), and therefore the extra capacity in the OP space does not make much of a difference. We therefore look at the effect of OP for only those drives that are full (more than 80% of capacity in use) and still observe slightly higher WAF for SSDs with higher OP. This suggests that there are likely other factors (e.g., workload characteristics and firmware) that are more dominant than OP. For example, the drives with 7% OP have support for multi-stream writes, which might help lower their WAF. Finally, 7% OP is a younger technology and thus, the corresponding systems can be (potentially) adopted by a different set of customers, whose workload characteristics might be different from the average customers across the 28% OP population.

4.5 Multi-stream Writes

Several drive models in our data set support multi-stream writes (MSW) [21], which can help reduce WAF by allowing the host to dictate the data placement on the SSD's physical blocks. In fact, our analysis of OP showed that drives with 7% OP, all of which have MSW support, report lower WAF. Therefore, we perform a detailed analysis on the impact of MSW, while controlling for other factors (e.g., DWPD, *role*).

We observe relatively clear trends for *data* drives, where populations with MSW have 30-40% lower WAF than comparable populations without MSW.

However, the trend is not clear, and in fact sometimes reversed for *partitioned* drives. It's possible that workload factors (which we previously saw are strong for partitioned drives) dominate those populations. It's also possible that for partitioned drives the streams are mostly used for performance isolation of different partitions, rather than for reducing WAF.

5 Read/Write (R/W) Ratios

In this section, we characterize the read/write (R/W) ratios exhibited by the workloads in our systems. R/W ratios are an interesting aspect of SSD-based systems for multiple reasons:

First, the combination of reads and writes can significantly impact the observed performance of reads in SSDs, as read operations compete internally with (slower) write operations. Second, newer generation of SSDs, such as QLC SSDs, are targeted for read-intensive workloads, as their PE cycle limits are much smaller than previous generations (up to 10× compared to TLC [32]). Therefore, exploring the trends in existing workloads is interesting. Third, providing data on R/W ratios in production systems helps researchers and practitioners to set up more realistic testbeds, as a workload's R/W ratio is a key configuration parameter of existing benchmark tools, such as FIO [2]. The results of our study can be used to parameterize simulation and experimental testbeds with R/W ratios that are representative of production workloads. Finally, in WBC systems, where the SSDs are used as a caching layer on top of HDDs, the read/write ratio can be viewed as a measure of the cache's effectiveness in caching reads.

In this section, we perform the analysis of read/write ratios separately for WBC systems and AFF systems, as read/write ratios have a different interpretation for these two systems. We distinguish between the two system *types*, as customers who buy HDD-based systems tend to use them differently from those who buy SSD-based systems; in our analysis, we characterize the differences.

5.1 R/W ratios and AFF systems

Figure 9 (left) shows the distribution of R/W ratios associated with the SSDs in AFF systems, computed based on host reads and host writes reported by our systems. We begin with a few high-level observations based on this figure:

- We observe that the vast majority of drives, around 94%, experience more reads than writes. The median R/W ratio is 3.6:1 and the 95th percentile is 61:1.
- These R/W ratios are in stark contrast to trace analysis results from HDD-based storage systems, which generally have more writes than reads. For example, the FIU traces [24], the majority of volumes in the Microsoft traces [35], and the recent block I/O traces from Alibaba data centers [27], all experience more writes than reads.
- The significant difference between the R/W rates of SSD-

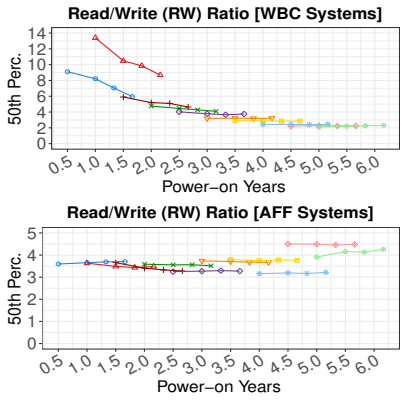


Figure 10: *Evolution of the drives' R/W ratios over time, for WBC (top) and AFF (bottom) systems.*

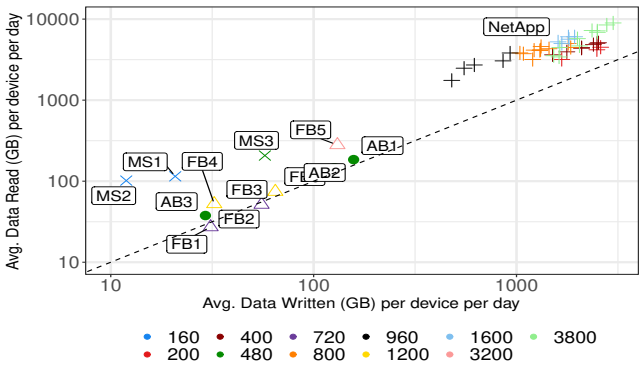


Figure 11: *Comparison of the daily workload experienced by SSDs in data centers versus enterprise storage systems. The dotted line represents equal amount of daily data reads and writes; read-dominant workloads are above the line.*

and HDD-based systems underlines the importance of our earlier observation that replaying traces from HDD systems for experiments or simulations of SSD systems is problematic. Our community needs to find a way to collect and make publicly available block traces from SSD-based systems.

R/W ratios over time: The next question we look at is whether R/W ratios remain stable over time. Towards this end, we group SSDs into cohorts based on their age and monitor each cohort of drives over time. Each cohort spans a 6-month time frame (e.g., months 12–18, representing the first six months of the 2nd year in production). Note that there is no overlap between cohorts; for instance, if an SSD is placed into the cohort corresponding to total deployment time up to 18 months, it is not placed into any other cohort before that, even though at some point in its lifetime it had been deployed for that amount of time. For each cohort, we report its (median) R/W ratio at different points in time.

The results for R/W ratios over time are shown in Figure 10 (bottom). Each line segment in the graph corresponds to one of the cohorts described above. We make two observations:

• R/W ratios in AFF systems remain rather stable over time, suggesting that the characteristics of the corresponding workloads do not drastically change over time.

• The only time in a drive's lifetime when R/W ratios tend to change is towards their end of life. In particular, we see ratios increasing after around 4.5 years in production. This might likely be due to systems being drained before being retired.

R/W ratios and system capacity and fullness: Next, we explore whether R/W ratios look different based on system capacity and system fullness.

We find that systems with smaller capacities are associated with higher R/W ratios; the 50th and 90th percentiles of the R/W rates associated with smaller systems are up to 2× higher than those for larger systems.

When we examine how R/W ratios look like for different levels of fullness, we interestingly observe no significant differences in the R/W ratios among systems which use more than 25% of their total space, suggesting that systems which are more full do not necessarily experience read-dominant workloads only.

Comparison with data center drives: Three recent field studies on data center drives at Facebook, Microsoft, and Alibaba, which mainly focus on failure characteristics, also report some aggregate statistics on the read and write rates associated with these drives [33, 36, 46].

Figure 11 plots the physical NAND read and write rates for those data center drives (except for Alibaba drives which involve host reads/writes), as well as the (host) read and write rates of the SSDs in our enterprise storage systems (which do not involve any requests served from the DRAM cache); we have selected only those SSDs from our data set with a capacity comparable to the data center drives.

We make two observations:

• First, the workloads associated with the SSDs in our data set are significantly more intensive: the corresponding read and write rates are at least one order of magnitude higher than the ones in the other two studies (note the log scale on both axes). Keeping in mind that our rates involve host reads and writes, while those of the two data center studies report physical reads and writes, the actual differences are even larger.

• Second, in contrast to the drives at Facebook [33] and Alibaba [46], which report a comparable number of reads and writes, our systems see a larger number of reads than writes. Still, concerning drives at Facebook, the difference might be due to the fact the we report host writes while that study reports physical writes (which include WAF writes).

The R/W rates of Microsoft drives [36] look comparable to ours in Figure 11, however given that their write rates are physical NAND writes, while our write rates are host writes (not accounting for WAF), the R/W ratios of their applications are likely much higher than the average across our systems.

In summary, the read and write rates and the read/write ratios experienced by SSDs in enterprise storage systems vary significantly from those reported for data center drives, highlighting the differences (in terms of workload characteristics) between enterprise storage systems and data centers.

5.2 R/W ratios and Write-back cache systems

R/W ratios in WBC systems have a different significance than for AFF systems (where they mostly characterize the difference in reads and writes generated by applications running on the systems). SSDs in WBC systems are used as a cache layer that aims to increase performance, while persistent storage is provided by another layer consisting of HDDs. The R/W ratio of accesses to the SSDs can therefore be viewed as one measure of the effectiveness of the cache: the R/W ratio provides some indication of how many cache reads (hits) we get for one write to the cache[4].

As we observe in Figure 9, WBC systems experience higher R/W ratios than AFF systems. Specifically, the median R/W ratio across the entire population is 4.1:1 and the 95th percentile is 150:1 (4.4× higher than for AFF). The high R/W ratios associated with WBC systems suggest that the cache layer is used effectively.

We make some interesting observations regarding R/W ratios of WBC systems over time. Figure 10 (top) again fixes cohorts of drives of similar age and monitors them over time.

• When following an individual cohort of drives over time (i.e., one specific line segment in the graph), we observe a clear drop in R/W ratio over time, particularly in the first half of a drive's life. This indicates the cache is becoming less effective as a read cache over time, likely because the total amount of data stored on the system increases over time and as a result, the (fixed-size) cache can cache increasingly smaller fractions of the total data. In particular, towards the end of a drive's life R/W ratios are quite low, with only two reads for every write.

• We make another interesting observation when comparing the R/W ratios for different line segments against each other, in particular in areas of the x-axis where they overlap. More recently deployed systems tend to have higher R/W ratios, even when comparing them with older systems at the same age. This might either indicate a trend of workloads changing over time towards higher R/W ratios, or customers configuring their storage systems differently (with a larger cache size relative to the amount of data stored for more recent systems).

6 Conclusions

We briefly summarize the key findings of our study in Table 4.

7 Acknowledgements

We would like to acknowledge all the internal reviewers at NetApp whose feedback improved this paper a lot. We owe a debt of gratitude to NetApp's ActiveIQ team; Asha Gangolli, Kavitha Degavinti, and Vinaya Nagaraj, who generated the data sets we used for this paper. We also thank our FAST reviewers and our shepherd, Xiaosong Ma, for their detailed and valuable feedback. This work was supported by an NSERC Discovery Grant and an NSERC Canada Research Chair.

[4]Note that the R/W ratio is not exactly a cache hit rate, as we do not know how many reads bypass the cache and read straight from the HDD layer.

Most Important Findings
§3.1.2: The majority of SSDs in our data set consume PE cycles at a very slow rate. Our projections indicate that the vast majority of the population (~95%) could move toward QLC without wearing out prematurely.
§3.1, 3.2: The host write rates for SSDs used as caches are significantly higher than for SSDs used as persistent storage. Yet, they do not see higher NAND write rates as they also experience lower WAF. It is thus not necessarily required to use higher endurance drives for cache workloads (which is a common practice).
§3.2: WAF varies significantly (orders of magnitude) across drive families and manufacturers. We conclude that the degree to which a drive's firmware affects its WAF can be surprisingly high, compared to other factors also known to affect WAF.
§3.2: We identify as the main contributor to WAF, for those drive families with the highest WAF, the aggressive rewriting of blocks to avoid retention issues. This is surprising, as other maintenance tasks (e.g., garbage collection, wear-leveling) generally receive more attention; common flash simulators and emulators (e.g., FEMU) do not even model rewriting to avoid retention issues.
§3.2: The WAF of our drives is higher than values reported in various academic studies based on trace-driven simulation. This demonstrates that it is challenging to recreate the real-world complexities of SSD internals and workloads in simulation.
§3.3: Wear leveling is not perfect. For instance, 5% of all SSDs report an erase ratio above 6, i.e., there are blocks in the drive which will wear out six times as fast as the average block. This is a concern not only because of early wear-out, but also because those blocks are more likely to experience errors and error correction contributes to tail latencies.
§3.4: AFF systems are on average 43% full. System fullness increases faster during the first couple of years in production, and after that increases only slowly. Systems with the largest capacity are fuller than smaller systems.
§4.3, §4.4: We find that over-provisioning and fullness have little impact on WAF in practice, unlike commonly assumed.
§5: The vast majority of workloads (94%) associated with SSDs in our systems are read-dominant, with a median R/W ratio of 3.62:1, highlighting the differences in usage between SSD-based and HDD-based systems. Many widely-used traces from HDD-based systems see more writes than reads, raising concerns when using these traces for SSD research, as is common in practice.
§5: The read and write rates for the drives in our enterprise storage systems are an order of magnitude higher than those reported for data center drives (comparing same-capacity drives).
§5: The read/write ratio reported by SSDs that act as caches decreases significantly over their lifetime. This might indicate a decreasing effectiveness of the SSD cache over time.
§3.2, §5: The differences between some of our results and those reported based on the analysis of widely used HDD-based storage traces emphasize the importance for us as a community to bring some representative SSD-based traces into the public domain.

Table 4: *The most important findings per section.*

References

[1] Nitin Agrawal, William J Bolosky, John R Douceur, and Jacob R Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)*, 3(3):9–40, 2007.

[2] Jens Axboe. Flexible I/O tester. https://github.com/axboe/fio. Accessed: 2020-01-05.

[3] 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 SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, pages 289–300, 2007.

[4] Janki Bhimani, Jingpei Yang, Zhengyu Yang, Ningfang Mi, NHV Krishna Giri, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. Enhancing ssds with multi-stream: What? Why? How? In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–2. IEEE, 2017.

[5] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[6] Yu Cai, Yixin Luo, Erich F Haratsch, Ken Mai, and Onur Mutlu. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 551–563. IEEE, 2015.

[7] Chandranil Chakraborttii and Heiner Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–12, 2021.

[8] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '05)*, pages 1–14. USENIX Association, 2004.

[9] Peter Desnoyers. Analytic modeling of SSD write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–10, 2012.

[10] Peter Desnoyers. Analytic models of SSD write performance. *ACM Transactions on Storage (TOS)*, 10(2):1–25, 2014.

[11] Atul Goel and Peter Corbett. RAID triple parity. *ACM SIGOPS Operating Systems Review*, 46(3):41–49, 2012.

[12] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 24–33, 2009.

[13] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M Fields, Kevin Harms, Robert B Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, Birali H Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):1–26, 2018.

[14] Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. To collect or not to collect: Just-in-time garbage collection for high-performance SSDs with long lifetimes. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[15] Longzhe Han, Yeonseung Ryu, and Keunsoo Yim. CATA: A garbage collection scheme for flash memory file systems. In *International Conference on Ubiquitous Intelligence and Computing*, pages 103–112. Springer, 2006.

[16] 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 '16)*, pages 263–276, 2016.

[17] Dave Hitz, James Lau, and Michael A Malcolm. File System Design for an NFS File Server Appliance. In *USENIX Winter*, volume 94, 1994.

[18] Jen-Wei Hsieh, Tei-Wei Kuo, and Li-Pin Chang. Efficient identification of hot data for flash memory storage systems. *ACM Transactions on Storage (TOS)*, 2(1):22–40, 2006.

[19] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–9, 2009.

[20] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *12th USENIX Conference on File and Storage Technologies (FAST '14)*, pages 61–74, 2014.

[21] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '14)*, 2014.

[22] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. Countering Fragmentation in an Enterprise Storage System. *ACM Trans. Storage (TOS)*, 15(4), jan 2020.

[23] Ram Kesavan, Jason Hennessey, Richard Jernigan, Peter Macko, Keith A Smith, Daniel Tennant, and VR Bharadwaj. FlexGroup Volumes: A Distributed WAFL File System. In *2019 USENIX Annual Technical Conference (ATC '19)*, pages 135–148. USENIX Association, 2019.

[24] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

[25] Eunji Lee, Julie Kim, Hyokyung Bahn, Sunjin Lee, and Sam H Noh. Reducing write amplification of flash storage through cooperative data management with NVM. *ACM Transactions on Storage (TOS)*, 13(2):1–13, 2017.

[26] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies (FAST '18)*, pages 83–90, 2018.

[27] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. An In-Depth Analysis of Cloud Block Storage Workloads in Large-Scale Production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 37–47. IEEE, 2020.

[28] 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, 2021.

[29] Shuwen Liang, Zhi Qiao, Jacob Hochstetler, Song Huang, Song Fu, Weisong Shi, Devesh Tiwari, Hsing-Bung Chen, Bradley Settlemyer, and David Montoya. Reliability characterization of solid state drives in a scalable production datacenter. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3341–3349. IEEE, 2018.

[30] 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 '20)*, Santa Clara, CA, 2020. USENIX Association.

[31] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. Reliability of SSDs in Enterprise Storage Systems: A Large-Scale Field Study. *ACM Trans. Storage (TOS)*, 17(1), jan 2021.

[32] Chris Mellor. WD and Tosh talk up penta-level cell flash. https://blocksandfiles.com/2019/08/07/penta-level-cell-flash/. Accessed: 2021-01-04.

[33] 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 SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '15)*, pages 177–190, 2015.

[34] Micron. Comparing SSD and HDD Endurance in the Age of QLC SSDs. https://www.micron.com/-/media/client/global/documents/products/white-paper/5210_ssd_vs_hdd_endurance_white_paper.pdf. Accessed: 2021-01-03.

[35] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):1–23, 2008.

[36] 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)*, pages 7:1–7:11, 2016.

[37] NetApp Inc. Data ONTAP 9. https://docs.netapp.com/ontap-9/index.jsp. Accessed: 2020-10-12.

[38] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S Gunawi. IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services. In *2019 USENIX Annual Technical Conference (ATC '19)*, pages 47–62. USENIX Association, 2019.

[39] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[40] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. FStream: Managing flash streams in the file system. In *16th USENIX Conference on File and Storage Technologies (FAST '18)*, pages 257–264, 2018.

[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 '16)*, pages 67–80, Santa Clara, CA, 2016. USENIX Association.

[42] Mansour Shafaei, Peter Desnoyers, and Jim Fitzpatrick. Write amplification reduction in flash-based SSDs through extent-based temperature identification. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '16)*, 2016.

[43] Murray Stokely, Amaan Mehrabian, Christoph Albrecht, Francois Labelle, and Arif Merchant. Projecting disk usage based on historical trends in a cloud environment. In *Proceedings of the 3rd workshop on Scientific Cloud Computing*, pages 63–70, 2012.

[44] Shunzhuo Wang, You Zhou, Jiaona Zhou, Fei Wu, and Changsheng Xie. An Efficient Data Migration Scheme to Optimize Garbage Collection in SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(3):430–443, 2020.

[45] Guanying Wu and Xubin He. Delta-FTL: improving SSD lifetime via exploiting content locality. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*, pages 253–266, 2012.

[46] 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 (ATC '19)*, pages 961–976, Renton, WA, 2019. USENIX Association.

[47] Jing Yang, Shuyi Pei, and Qing Yang. WARCIP: Write amplification reduction by clustering I/O pages. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 155–166, 2019.

[48] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. Reducing garbage collection overhead in SSD based on workload prediction. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[49] Qi Zhang, Xuandong Li, Linzhang Wang, Tian Zhang, Yi Wang, and Zili Shao. Lazy-RTGC: A real-time lazy garbage collection mechanism with jointly optimizing average and worst performance for NAND flash memory storage systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(3):1–32, 2015.

Hydra : Resilient and Highly Available Remote Memory

Youngmoon Lee*[1], Hasan Al Maruf*[2], Mosharaf Chowdhury[2], Asaf Cidon[3], and Kang G. Shin[2]

[1]Hanyang University, [2]University of Michigan, [3]Columbia University

Abstract

We present Hydra, a low-latency, low-overhead, and highly available resilience mechanism for remote memory. Hydra can access erasure-coded remote memory within a single-digit μs read/write latency, significantly improving the performance-efficiency tradeoff over the state-of-the-art – it performs similar to in-memory replication with 1.6× lower memory overhead. We also propose CodingSets, a novel coding group placement algorithm for erasure-coded data, that provides load balancing while reducing the probability of data loss under correlated failures by an order of magnitude. With Hydra, even when only 50% memory is local, unmodified memory-intensive applications achieve performance close to that of the fully in-memory case in the presence of remote failures and outperforms the state-of-the-art remote-memory solutions by up to 4.35×.

1 Introduction

Modern datacenters are embracing a paradigm shift toward disaggregation, where each resource is decoupled and connected through a high-speed network fabric [4, 9, 13, 35–37, 58, 61, 62, 81]. In such disaggregated datacenters, each server node is specialized for specific purposes – some are specialized for computing, while others for memory, storage, and so on. Memory, being the prime resource for high-performance services, is becoming an attractive target for disaggregation [18, 19, 22, 32, 39, 47, 50, 58, 61].

Recent remote-memory frameworks allow an unmodified application to access remote memory in an implicit manner via well-known abstractions such as distributed virtual file system (VFS) and distributed virtual memory manager (VMM) [18, 47, 50, 58, 65, 81, 87]. With the advent of RDMA, remote-memory solutions are now close to meeting the single-digit μs latency required to support acceptable application-level performance [47, 58]. However, realizing remote memory for heterogeneous workloads running in a large-scale cluster faces considerable challenges [19, 24] stemming from two root causes:

1. *Expanded failure domains:* As applications rely on memory across multiple machines in a remote-memory cluster, they become susceptible to a wide variety of failure scenarios. Potential failures include independent and correlated failures of remote machines, evictions from and corruptions of remote memory, and network partitions.
2. *Tail at scale:* Applications also suffer from stragglers or late-arriving remote responses. Stragglers can arise from many sources including latency variabilities in a large network due to congestion and background traffic [41].

While one leads to catastrophic failures and the other manifests as service-level objective (SLO) violations, both are unacceptable in production [58, 68]. Existing solutions take three primary approaches to address them: (i) local disk backup [50, 81], (ii) remote in-memory replication [30, 42, 46, 64], and (iii) remote in-memory erasure coding [76, 80, 84, 86] and compression [58]. Unfortunately, they suffer from some combinations of the following problems.

High latency: Disk backup has no additional memory overhead, but the access latency is intolerably high under any correlated failures. Systems that take the third approach do not meet the single-digit μs latency requirement of remote memory even when paired with RDMA (Figure 1).

High cost: Replication has low latency, but it doubles memory consumption and network bandwidth requirements. Disk backup and replication represent the two extreme points in the performance-vs-efficiency tradeoff space (Figure 1).

Low availability: All three approaches lose availability to low latency memory when even a very small number of servers become unavailable. With the first approach, if a single server fails its data needs to be reconstituted from disk, which is a slow process. In the second and third approach, when even a small number of servers (e.g., three) fail simultaneously, some users will lose access to data. This is due to the fact that replication and erasure coding assign replicas and coding groups to *random* servers. Random data placement is susceptible to data loss when a small number of servers fail at the same time [27, 28] (Figure 2).

In this paper, we consider how to mitigate these problems and present Hydra, a low-latency, low-overhead, and highly available resilience mechanism for remote memory. While erasure codes are known for reducing storage overhead and for better load balancing, it is challenging for remote memory with μs-scale access requirements (preferably, 3-5μs) [47]. We demonstrate how to achieve resilient erasure-coded cluster

*These authors contributed equally to this work

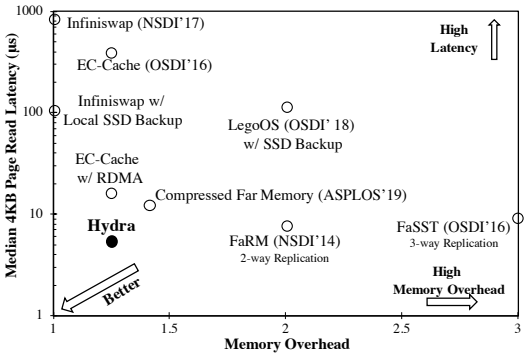


Figure 1: ***Performance-vs-efficiency tradeoff in the resilient cluster memory design space. Here, the Y-axis is in log scale.***

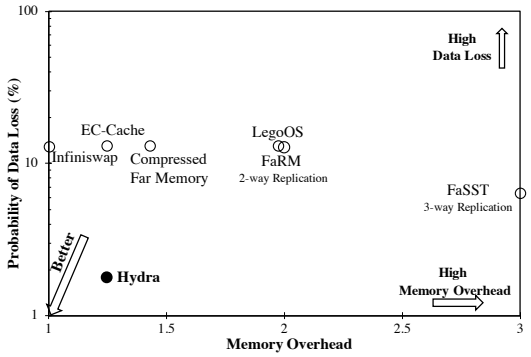


Figure 2: ***Availability-vs-efficiency tradeoff considering 1% simultaneous server failures in a 1000-machine cluster.***

memory with single-digit μs latency even under simultaneous failures at a reduced data amplification overhead.

We explore the challenges and tradeoffs for resilient remote memory without sacrificing application-level performance or incurring high overhead in the presence of correlated failures (§2). We also explore the trade-off between load balancing and high availability in the presence of simultaneous server failures. Our solution, Hydra, is a configurable resilience mechanism that applies online erasure coding to individual remote memory pages while maintaining high availability (§3). Hydra's carefully designed data path enables it to access remote memory pages within a single-digit μs median and tail latency (§4). Furthermore, we develop CodingSets, a novel coding group placement algorithm for erasure codes that provides load balancing while reducing the probability of data loss under correlated failures (§5).

We develop Hydra as a drop-in resilience mechanism that can be applied to existing remote memory frameworks [18, 22, 50, 65, 81]. We integrate Hydra with the two major remote memory approaches widely embraced today: disaggregated VMM (used by Infiniswap [50], and Leap [65]) and disaggregated VFS (used by Remote Regions [18]) (§6). Our evaluation using production workloads shows that Hydra achieves the best of both worlds (§7). Hydra closely matches the performance of replication-based resilience with 1.6× lower memory overhead with or without the presence of failures. At the same time, it improves latency and throughput of the benchmark applications by up to 64.78× and 20.61×, respectively, over SSD backup-based resilience with only 1.25× higher memory overhead. While providing resiliency, Hydra also improves the application-level performance by up to 4.35× over its counterparts. CodingSets reduces the probability of data loss under simultaneous server failures by about 10×. Hydra is available at https://github.com/SymbioticLab/hydra.

In this paper, we make the following contributions:

- Hydra is the first in-memory erasure coding scheme that achieves single-digit μs tail memory access latency.
- Novel analysis of load balancing and availability trade-off for distributed erasure codes.
- CodingSets is a new data placement scheme that balances availability and load balancing, while reducing probability of data loss by an order of magnitude during failures.

2 Background and Motivation

2.1 Remote Memory

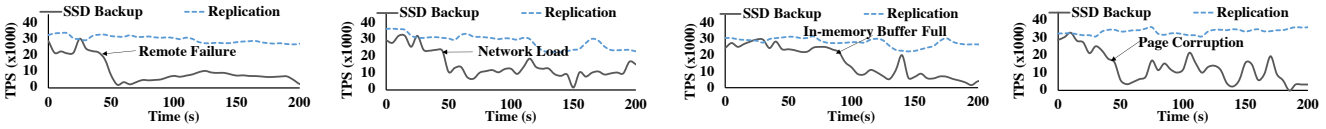
Remote memory exposes memory available in remote machines as a pool of memory shared by many machines. It is often implemented logically by leveraging stranded memory in remote machines via well-known abstractions, such as the file abstraction [18], remote memory paging [22,47,50,59,65], and virtual memory management for distributed OS [81]. In the past, specialized memory appliances for physical memory disaggregation were proposed as well [61, 63].

All existing remote-memory solutions use the 4KB page granularity. While some applications use huge pages for performance enhancement [57], the Linux kernel still performs paging at the basic 4KB level by splitting individual huge pages because huge pages can result in high amplification for dirty data tracking [23]. Existing remote-memory systems use disk backup [50, 81] and in-memory replication [46, 64] to provide availability during failures.

2.2 Failures in Remote Memory

The probability of failure or temporary unavailability is higher in a large remote-memory cluster, since memory is being accessed remotely. To illustrate possible performance penalties in the presence of such unpredictable events, we consider a resilience solution from the existing literature [50], where each page is asynchronously backed up to a local SSD. We run transaction processing benchmark TPC-C [16] on an in-memory database system, VoltDB [17]. We set VoltDB's available memory to 50% of its peak memory to force remote paging for up to 50% of its working set.

1. Remote Failures and Evictions Machine failures are the norm in large-scale clusters where thousands of machines crash over a year due to a variety of reasons, including software and hardware failures [31,33,38,88]. Concurrent failures within a rack or network segments are quite common and typ-



(a) *Remote failure* (b) *Background Network Load* (c) *Request burst* (d) *Page Corruption*

Figure 3: ***TPC-C throughput over time on VoltDB when 50% of the working set fits in memory. Arrows point to uncertainty injection time.***

ically occur dozens of times a year. Even cluster-wide power outage is not uncommon – occurs once or twice per year in a given data center. For example, during a recent cluster-wide power outage in Google Cloud, around 23% of the machines were unavailable for hours [6].

Without redundancy, applications relying on remote memory may fail when a remote machine fails or remote memory pages are evicted. As disk operations are significantly slower than the latency requirement of remote memory, disk-based fault-tolerance is far from being practical. In the presence of a remote failure, VoltDB experiences almost 90% throughput loss (Figure 3a); throughput recovery takes a long time after the failure happens.

2. Background Network Load Network load throughout a large cluster can experience significant fluctuations [41, 53], which can inflate RDMA latency and application-level stragglers, causing unpredictable performance issues [40, 89]. In the presence of an induced bandwidth-intensive background load, VoltDB throughput drops by about 50% (Figure 3b).

3. Request Bursts Applications can have bursty memory access patterns. Existing solutions maintain an in-memory buffer to absorb temporary bursts [18, 50, 74]. However, as the buffer ties remote access latency to disk latency when it is full, the buffer can become the bottleneck when a workload experiences a prolonged burst. While a page read from remote memory is still fast, backup page writes to the local disk become the bottleneck after the 100^{th} second in Figure 3c. As a result, throughput drops by about 60%.

4. Memory Corruption During remote memory access, if any one of the remote servers experiences a corruption, or if the memory gets corrupted over the network a memory corruption event will occur. In such case, disk access causes failure-like performance loss (Figure 3d).

Performance vs. Efficiency Tradeoff for Resilience In all of these scenarios, the obvious alternative – in-memory $2\times$ or $3\times$ replication [46, 64] – is effective in mitigating a small-scale failure, such as the loss of a single server (Figure 3a). When an in-memory copy becomes unavailable, we can switch to an alternative. Unfortunately, replication incurs high memory overhead in proportion to the number of replicas. This defeats the purpose of remote memory. Hedging requests to avoid stragglers [41] in a replicated system doubles its bandwidth requirement as well.

This leads to an impasse: one has to either settle for high latency in the presence of a failure or incur high memory overhead. Figure 1 depicts this performance-vs-efficiency tradeoff under failures and memory usage overhead to provide resilience. Beyond the two extremes in the tradeoff space, there are two primary alternatives to achieve high resilience with low overhead. The first is replicating pages to remote memory after compressing them (e.g., using zswap) [58], which improves the tradeoff in both dimensions. However, its latency can be more than 10μs when data is in remote memory. Especially, during resource scarcity, the presence of a prolonged burst in accessing remote compressed pages can even lead to orders of magnitude higher latency due to the demand spike in both CPU and local DRAM consumption for decompression. Besides, this approach faces similar issues as replication such as latency inflation due to stragglers.

The alternative is erasure coding, which has recently made its way from disk-based storage to in-memory cluster caching to reduce storage overhead and improve load balancing [20,25, 76, 83, 84, 86]. Typically, an object is divided into k *data splits* and encoded to create r equal-sized *parity splits* ($k > r$), which are then distributed across $(k+r)$ failure domains. Existing erasure-coded memory solutions deal with large objects (e.g., larger than 1 MB [76]), where hundreds-of-μs latency of the TCP/IP stack can be ignored. Simply replacing TCP with RDMA is not enough either. For example, the EC-Cache with RDMA (Figure 1) provides a lower storage overhead than compression but with a latency around 20μs.

Last but not least, all of these approaches experience high unavailability in the presence of correlated failures [28].

2.3 Challenges in Erasure-Coded Memory

High Latency Individually erasure coding 4 KB pages that are already small lead to even smaller data chunks ($\frac{4}{k}$ KB), which contributes to the higher latency of erasure-coded remote memory over RDMA due to following primary reasons:

1. ***Non-negligible coding overhead:*** When using erasure codes with on-disk data or over slower networks that have hundreds-of-μs latency, its 0.7μs encoding and 1.5μs decoding overheads can be ignored. However, they become non-negligible when dealing with DRAM and RDMA.
2. ***Stragglers and errors:*** As erasure codes require k splits before the original data can be constructed, any straggler can slow down a remote read. To detect and correct an error, erasure codes require additional splits; an extra read adds another round-trip to double the overall read latency.
3. ***Interruption overhead:*** Splitting data also increases the total number of RDMA operations for each request. Any

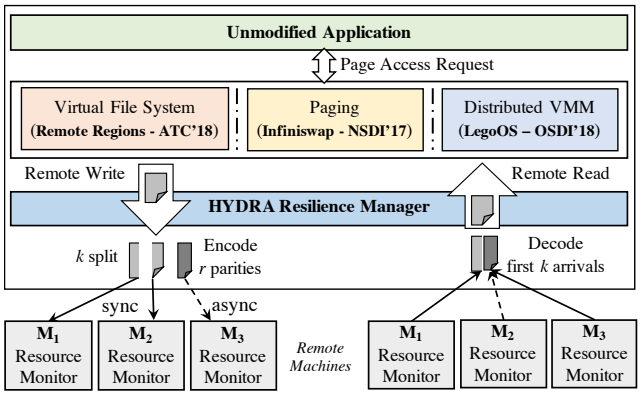


Figure 4: ***Resilience Manager provides with resilient, erasure-coded remote memory abstraction. Resource Monitor manages the remote memory pool. Both can be present in a machine.***

context switch in between can further add to the latency.

4. ***Data copy overhead:*** In a latency-sensitive system, additional data movement can limit the lowest possible latency. During erasure coding, additional data copy into different buffers for data and parity splits can quickly add up.

Availability Under Simultaneous Failures Existing erasure coding schemes can handle a small-scale failure without interruptions. However, when a relatively modest number of servers fail or become unavailable at the same time (e.g., due to a network partition or a correlated failure event), they are highly susceptible to losing availability to some of the data.

This is due to the fact that existing erasure coding schemes generate coding groups on random sets of servers [76]. In a coding scheme with k data and r parity splits, an individual coding group, will fail to decode the data if $r+1$ servers fail simultaneously. Now in a large cluster with $r+1$ failures, the probability that those $r+1$ servers will fail for a *specific* coding group is low. However, when coding groups are generated randomly (i.e., each one of them compromises a random set of $k+r$ servers), and there are a large number of coding groups per server, then the probability that those $r+1$ servers will affect *any* coding group in the cluster is much higher. Therefore, state-of-the-art erasure coding schemes, such as EC-Cache, will experience a very high probability of unavailability even when a very small number of servers fail simultaneously.

3 Hydra Architecture

Hydra is an erasure-coded resilience mechanism for existing remote-memory techniques to provide better performance-efficiency tradeoff under remote failures while ensuring high availability under simultaneous failures. It has two main components (Figure 4): (i) **Resilience Manager** coordinates erasure-coded resilience operations during remote read/write; (ii) **Resource Monitor** handles the memory management in a remote machine. Both can be present in every machine and work together without central coordination.

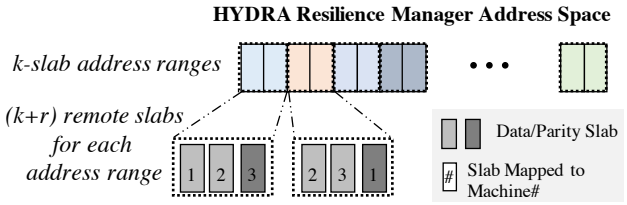


Figure 5: ***Hydra's address space is divided into fixed-size address ranges, each of which spans*** $(k+r)$ ***memory slabs in remote machines; i.e.,*** k ***for data and*** r ***for parity (*** k***=2 and*** r***=1 in this figure).***

3.1 Resilience Manager

Hydra Resilience Manager provides remote memory abstraction to a client machine. When an unmodified application accesses remote memory through different state-of-the-art remote-memory solutions (e.g., via VFS or VMM), the Resilience Manager transparently handles all aspects of RDMA communication and erasure coding. Each client has its own Resilience Manager that handles slab placement through CodingSets, maintains remote slab-address mapping, performs erasure-coded RDMA read/write. Resilience Manager communicates to Resource Monitor(s) running on remote memory host machines, performs remote data placement, and ensures resilience. As a client's Resilience Manager is responsible for the resiliency of its remote data, the Resilience Managers do not need to coordinate with each other.

Following the typical (k,r) erasure coding construction, the Resilience Manager divides its remote address space into fixed-size *address ranges*. Each address range resides in $(k+r)$ remote *slabs*: k slabs for page data and r slabs for parity (Figure 5). Each of the $(k+r)$ slabs of an address range are distributed across $(k+r)$ independent failure domains using CodingSets (§5). Page accesses are directed to the designated $(k+r)$ machines according to the address–slab mapping. Although remote I/O happens at the page level, the Resilience Manager coordinates with remote Resource Monitors to manage coarse-grained memory slabs to reduce metadata overhead and connection management complexity.

3.2 Resource Monitor

Resource Monitor manages a machine's local memory and exposes them to the remote Resilience Manager in terms of fixed-size (*SlabSize*) memory slabs. Different slabs can belong to different machines' Resilience Manager. During each control period (*ControlPeriod*), the Resource Monitor tracks the available memory in its local machine and proactively allocates (reclaims) slabs to (from) remote mapping when memory usage is low (high). It also performs slab regeneration during remote failures or corruptions.

Fragmentation in Remote Memory During the registration of Resource Monitor(s), Resilience Manager registers the RDMA memory regions and allocates slabs on the remote machines based on its memory demand. Memory regions are usually large (by default, 1GB) and the whole address space is

homogeneously splitted. Moreover, RDMA drivers guarantee the memory regions are generated in a contiguous physical address space to ensure faster remote-memory access. Hydra introduces no additional fragmentation in remote machines.

3.3 Failure Model

Assumptions In a large remote-memory cluster, (a) remote servers may crash or networks may become partitioned; (b) remote servers may experience memory corruption; (c) the network may become congested due to background traffic; and (d) workloads may have bursty access patterns. These events can lead to catastrophic application-failures, high tail latencies, or unpredictable performance. Hydra addresses all of these uncertainties in its failure domain. Although Hydra withstands a remote-network partition, as there is no local-disk backup, it cannot handle local-network failure. In such cases, the application is anyways inaccessible.

Single vs. Simultaneous Failure A single node failure means the unavailability of slabs in a remote machine. In such an event, all the data or parity allocated on the slab(s) become unavailable. As we spread the data and parity splits for a page across multiple remote machines (§5), during a single node failure, we assume that only a single data or parity split for that page is being affected.

Simultaneous host failures typically occur due to large-scale failures, such as power or network outage that cause multiple machines to become unreachable. In such a case, we assume multiple data and/or parity splits for a page become unavailable. Note that in both cases, the data is unavailable, but not compromised. Resilience Manager can detect the unreachability and communicate to other available Resource Monitor(s) on to regenerate specific slab(s).

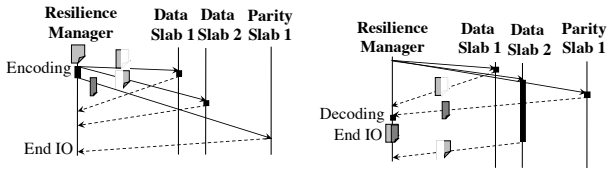
4 Resilient Data Path

Hydra can operate on different resilient modes based on a client's need – **(a)** *Failure Recovery*: provides resiliency in the presence of any remote failure or eviction; **(b)** *Corruption Detection*: only detects the presence of corruption in remote memory; **(c)** *Corruption Correction*: detects and corrects remote memory corruption; and **(d)** *EC-only mode*: provides erasure-coded faster remote-memory data path without any resiliency guarantee. Note that both of the corruption modes by default inherit the *Failure Recovery* mode.

Before initiating the Resilience Manager, one needs to configure Hydra to a specific mode according to the resilience requirements and memory overhead concerns (Table 1). Multiple resilience modes cannot act simultaneously, and the modes do not switch dynamically during runtime. In this section, we present Hydra's data path design to address the resilience challenges mentioned in §2.3.

4.1 Hydra Remote Memory Data Path

To minimize erasure coding's latency overheads, Resilience Manager's data path incorporate following design principles.



(a) *Remote Write* (b) *Remote Read*

Figure 6: ***To handle failures, Hydra (a) first writes data splits, then encodes/writes parities to hide encoding latency; (b) reads from $k+\Delta$ slabs to avoid stragglers, finishes with first k arrivals.***

4.1.1 Asynchronously Encoded Write

To hide the erasure coding latency, existing systems usually perform batch coding where multiple pages are encoded together. The encoder waits until a certain number of pages are available. This idle waiting time can be insignificant compared to disk or slow network (e.g., TCP) access. However, to maintain the tail latency of a remote I/O within the single-digit μs range, this "batch waiting" time needs to be avoided.

During a remote write, Resilience Manager applies erasure coding within each individual page by dividing it into k splits (for a 4 KB page, each split size is $\frac{4}{k}$KB), encodes these splits using Reed-Solomon (RS) codes [77] to generate r parity splits. Then, it writes these $(k+r)$ splits to different $(k+r)$ slabs that have already been mapped to unique remote machines. Each Resilience Manager can have their own choice of k and r. This individual page-based coding decreases latency by avoiding the "batch waiting" time. Moreover, the Resilience Manager does not have to read unnecessary pages within the same batch during remote reads, which reduces bandwidth overhead. Distributing remote I/O across many remote machines increases I/O parallelism too.

Resilience Manager sends the data splits first, then it encodes and sends the parity splits asynchronously. Decoupling the two hides encoding latency and subsequent write latency for the parities without affecting the resilience guarantee. In the absence of failure, any k successful writes of the $(k+r)$ allow the page to be recovered. However, to ensure resilience guarantee for r failures, all $(k+r)$ must be written. In the *failure recovery* mode, a write is considered complete after all $(k+r)$ have been written. In the *corruption correction (detection)* mode, to correct (detect) Δ corruptions, a write waits for $k+2\Delta+1$ $(k+\Delta)$ to be written. If the acknowledgement fails to reach the Resilience Manager due to a failure in the remote machine, the write for that split is considered failed. Resilience Manager tries to write that specific split(s) after a timeout period to another remote machine. Figure 6a depicts the timeline of a page write in the *failure recovery mode*.

4.1.2 Late-Binding Resilient Read

During read, any k out of the $k+r$ splits suffice to reconstruct a page. However, in *failure recovery* mode, to be resilient in the presence of Δ failures, during a remote read, Hydra Resilience Manager reads from $k+\Delta$ randomly chosen splits

Resilience Mode	# of Errors	Minimum # of Splits	Memory Overhead
Failure Recovery	r	k	$1+\frac{r}{k}$
Corruption Detection	Δ	$k+\Delta$	$1+\frac{\Delta}{k}$
Corruption Correction	Δ	$k+2\Delta+1$	$1+\frac{2\Delta+1}{k}$
EC-only	–	k	$1+\frac{r}{k}$

Table 1: ***Minimum number of splits needs to be written to/read from remote machines for resilience during a remote I/O.***

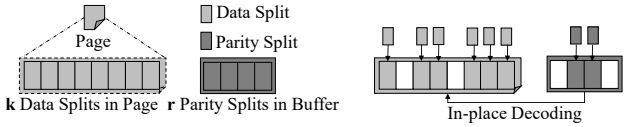
in parallel. A page can be decoded as soon as any k splits arrive out of $k+\Delta$. The additional Δ reads mitigate the impact of stragglers on tail latency as well. Figure 6b provides an example of a read operation in the *failure recovery* mode with $k=2$ and $\Delta=1$, where one of the data slabs (Data Slab 2) is a straggler. $\Delta=1$ is often enough in practice.

If simply "detect and discard corrupted memory" is enough for any application, one can configure Hydra with *corruption detection* mode and avoid the extra memory overhead of *corruption correction* mode. In *corruption detection* mode, before decoding a page, the Resilience Manager waits for $k+\Delta$ splits to arrive to *detect* Δ corruptions. After the detection of a certain amount of corruptions, Resilience Manager marks the machine(s) with corrupted splits as probable erroneous machines, initiates a background slab recovery operation, and avoids them during future remote I/O.

To correct the error, in *corruption correction* mode, when an error is detected, it requests additional $\Delta+1$ reads from the rest of the $k+r$ machines. Otherwise, the read completes just after the arrival of the $k+\Delta$ splits. If the error rate for a remote machine exceeds a user-defined threshold (*ErrorCorrectionLimit*), subsequent read requests involved with that machine initiates with $k+2\Delta+1$ split requests as there is a high probability to reach an erroneous machine. This will reduce the wait time for additional $\Delta+1$ reads. This continues until the error rate for the involved machine gets lower than the *ErrorCorrectionLimit*. If this continues for long and/or the error rate goes beyond another threshold (*SlabRegenerationLimit*), Resilience Manager initiates a slab regeneration request for that machine.

One can configure Hydra with *EC-only* mode to access erasure-coded remote memory and benefit from the fast data path without any resiliency guarantee. In this mode, a remote I/O completes just after writing/reading any k splits. Table 1 summarizes the minimum number of splits the Resilience Manager requires to write/read during a remote I/O operation to provide resiliency in different modes.

Overhead of Replication To remain operational after r failures, in-memory replication requires at least $r+1$ copies of an entire 4 KB page, and hence the memory overhead is $(r+1)\times$. However, a remote I/O operation can complete just after the confirmation from one of the $r+1$ machines. To detect and fix Δ corruptions, replication needs $\Delta+1$ and $2\Delta+1$ copies of the *entire* page, respectively. Thus, to provide the correctness guarantee over Δ corruptions, replication needs to wait until



(a) *Remote Write* (b) *Remote Read*

Figure 7: ***Hydra performs in-place coding with an extra buffer of r splits to reduce the data-copy latency.***

it writes to or reads from at least $2\Delta+1$ of the replicas along with a memory overhead of $(2\Delta+1)\times$.

4.1.3 Run-to-Completion

As Resilience Manager divides a 4 KB page into k smaller pieces, RDMA messages become smaller. In fact, their network latency decrease to the point that run-to-completion becomes more beneficial than a context switch. Hence, to avoid interruption-related overheads, the remote I/O request thread waits until the RDMA operations are done.

4.1.4 In-Place Coding

To reduce the number of data copies, Hydra Resilience Manager uses in-place coding with an extra buffer of r splits. During a write, the data splits are always kept in-page while the encoded r parities are put into the buffer (Figure 7a). Likewise, during a read, the data splits arrive at the page address, and the parity splits find their way into the buffer (Figure 7b).

In the failure recovery mode, a read can complete as soon as any k valid splits arrive. Corrupted/straggler data split(s) can arrive late and overwrite valid page data. To address this, as soon as Hydra detects the arrival of k valid splits, it deregisters relevant RDMA memory regions. It then performs decoding and directly places the decoded data in the page destination. Because the memory region has already been deregistered, any late data split cannot access the page. During all remote I/O, requests are forwarded directly to RDMA dispatch queues without additional copying.

4.2 Handling Uncertainties

Remote Failure Hydra uses reliable connections (RC) for all RDMA communication. Hence, we consider unreachability due to machine failures/reboots or network partition as the primary cause of failure. When a remote machine becomes unreachable, the Resilience Manager is notified by the RDMA connection manager. Upon disconnection, it processes all the in-flight requests in order first. For ongoing I/O operations, it resends the I/O request to other available machines. Since RDMA guarantees strict ordering, in the read-after-write case, read requests will arrive at the same RDMA dispatch queue after write requests; hence, read requests will not be served with stale data. Finally, Hydra marks the failed slabs and future requests are directed to the available ones. If the Resource Monitor in the failed machine revives and communicates later, Hydra reconsiders the machine for further remote I/O.

Adaptive Slab Allocation/Eviction Resource Monitor al-

locates memory slabs for Resilience Managers as well as proactively frees/evicts them to avoid local performance impacts (Figure 8). It periodically monitors local memory usage and maintains a headroom to provide enough memory for local applications. When the amount of free memory shrinks below the headroom (Figure 8a), the Resource Monitor first proactively frees/evicts slabs to ensure local applications are unaffected. To find the eviction candidates, we avoid random selection as it has a higher likelihood of evicting a busy slab. Rather, we uses the decentralized batch eviction algorithm [50] to select the least active slabs. To evict E slabs, we contact $(E+E')$ slabs (where $E' \leq E$) and find the least-frequently-accessed slabs among them. This doesn't require to maintain a global knowledge or search across all the slabs.

When the amount of free memory grows above the headroom (Figure 8b), the Resource Monitor first attempts to make the local Resilience Manager to reclaim its pages from remote memory and unmap corresponding remote slabs. Furthermore, it proactively allocates new, unmapped slabs that can be readily mapped and used by remote Resilience Managers.

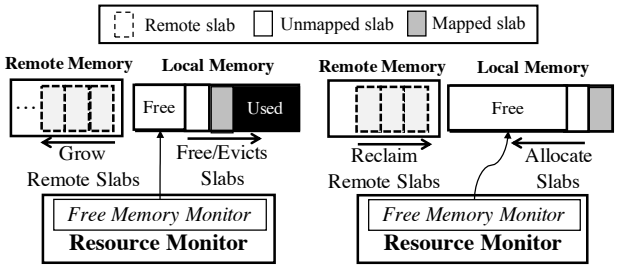
Background Slab Regeneration The Resource Monitor also regenerates unavailable slabs – marked by the Resilience Manager – in the background. During regeneration, writes to the slab are disabled to prevent overwriting new pages with stale ones; reads can still be served without interruption.

Hydra Resilience Manager uses the placement algorithm to find a new regeneration slab in a remote Resource Monitor with a lower memory usage. It then hands over the task of slab regeneration to that Resource Monitor. The selected Resource Monitor decodes the unavailable slab by directly reading the k randomly-selected remaining valid slab for that address region. Once regeneration completes, it contacts the Resilience Manager to mark the slab as available. Requests thereafter go to the regenerated slab.

5 CodingSets for High Availability

Hydra uses CodingSets, a novel coding group placement scheme to perform load-balancing while reducing the probability of data loss. Prior works show orders-of-magnitude more frequent data loss due to events causing multiple nodes to fail simultaneously than data loss due to independent node failures [27, 31]. Several scenarios can cause multiple servers to fail or become unavailable simultaneously, such as network partitions, partial power outages, and software bugs. For example, a power outage can cause 0.5%-1% machines to fail or go offline concurrently [28]. In case of Hydra, data loss will happen if a concurrent failure kills more than $r+1$ of $(k+r)$ machines for a particular *coding group*.

We are inspired by copysets, a scheme for preventing data loss under correlated failures in replication [27, 28], which constraints the number of replication groups, in order to *reduce the frequency of data loss events*. Using the same terminology as prior work, we define each unique set of $(k+r)$ servers within a coding group as a *copyset*. The number of



(a) *High memory pressure* (b) *Low memory pressure*

Figure 8: ***Resource Monitor proactively allocates memory for remote machines and frees local memory pressure.***

copysets in a single coding group will be: $\binom{k+r}{r+1}$. For example, in an (8+2) configuration, where nodes are numbered $1, 2, \ldots, 10$, the 3 nodes that will cause failure if they fail at the same time (i.e., copysets) will be every 3 combinations of 10 nodes: $(1,2,3), (1,2,4), \ldots, (8,9,10)$, and the total number of copysets will be $\binom{10}{3} = 120$.

For a data loss event impacting exactly $r+1$ random nodes simultaneously, the probability of losing data of a single specific coding group: $\mathbb{P}[Group] = \frac{\text{Num. of Copysets in Coding Group}}{\text{Total Copysets}} = \frac{\binom{k+r}{r+1}}{\binom{N}{r+1}}$, where N is the total number of servers.

In a cluster with more than $(k+r)$ servers, we need to use more than one coding group. However, if each server is a member of a single coding group, hot spots can occur if one or more members of that group are overloaded. Therefore, for load-balancing purposes, a simple solution is to allow each server to be a member of multiple coding groups, in case some members of a particular coding group are over-loaded at the time of online coding.

Assuming we have G *disjoint* coding groups, and the correlated failure rate is $f\%$, the total probability of data loss is: $1-(1-\mathbb{P}[Group]\cdot G)^{\binom{N\cdot f}{r+1}}$. We define disjoint coding groups where the groups do not share any copysets; or in other words, they do not overlap by more than r nodes.

Strawman: Multiple Coding Groups per Server In order to equalize load, we consider a scheme where each slab forms a coding group with the least-loaded nodes in the cluster at coding time. We assume the nodes that are least loaded at a given time are distributed randomly, and the number of slabs per server is S. When $S\cdot(r+k) \ll N$, the coding groups are highly likely to be disjoint [28], and the number of groups is equal to: $G = \frac{N\cdot S}{k+r}$.

We call this placement strategy the *EC-Cache scheme*, as it produces a random coding group placement used by the prior state-of-the-art in-memory erasure coding system, EC-Cache [76]. In this scheme, with even a modest number of slabs per server, a high number of combinations of $r+1$ machines will be a copyset. In other words, even a small number of simultaneous node failures in the cluster will result in data loss. When the number of slabs per server is high, almost

every combination of only $r+1$ failures across the cluster will cause data loss. Therefore, to reduce the probability of data loss, we need to minimize the number of copysets, while achieving sufficient load balancing.

CodingSets: Reducing Copysets for Erasure Coding To this end, we propose CodingSets, a novel load-balancing scheme, which reduces the number of copysets for distributed erasure coding. Instead of having each node participate in several coding groups like in EC-Cache, in our scheme, each server belongs to a single, *extended* coding group. At time of coding, $(k+r)$ slabs will still be considered together, but the nodes participating in the coding group are chosen from a set of $(k+r+l)$ nodes, where l is the load-balancing factor. The nodes chosen within the extended group are the least loaded ones. While extending the coding group increases the number of copysets (instead of $\binom{k+r}{r+1}$ copysets, now each extended coding group creates $\binom{k+r+l}{r+1}$ copysets, while the number of groups is $G = \frac{N}{k+r+l}$), it still has a significantly lower probability of data loss than having each node belong to multiple coding groups. Hydra uses CodingSets as its load balancing and slab placement policy. We evaluate it in Section 7.2.

Tradeoff Note that while CodingSets reduces the probability of data loss, it does not reduce the expected amount of data lost over time. In other words, it reduces the number of data loss events, but each one of these events will have a proportionally higher magnitude of data loss (i.e., more slabs will be affected) [28]. Given that our goal with Hydra is high availability, we believe this is a favorable trade off. For example, providers often provide an availability SLA, that is measured by the service available time (e.g., the service is available 99.9999% of the time). CodingSets would optimize for such an SLA, by minimizing the frequency of unavailability events.

6 Implementation

Resilience Manager is implemented as a loadable kernel module for Linux kernel 4.11 or later. Kernel-level implementation facilitates its deployment as an underlying block device for different remote-memory systems [18, 50, 81]. We integrated Hydra with two remote-memory systems: Infiniswap, a disaggregated VMM and Remote Regions, a disaggregated VFS. All I/O operations (e.g., slab mapping, memory registration, RDMA posting/polling, erasure coding) are independent across threads and processed without synchronization. All RDMA operations use RC and one-sided RDMA verbs (RDMA WRITE/READ). Each Resilience Manager maintains one connection for each active remote machine. For erasure coding, we use x86 AVX instructions and the ISA library [8] that achieves over 4 GB/s encoding throughput per core for (8+2) configuration in our evaluation platform.

Resource Monitor is implemented as a user-space program. It uses RDMA SEND/RECV for all control messages.

7 Evaluation

We evaluate Hydra on a 50-machine 56 Gbps InfiniBand CloudLab cluster against Infiniswap [50], Leap [65] (disaggregated VMM) and Remote Regions [18] (disaggregated VFS). Our evaluation addresses the following questions:

- Does it improve the resilience of cluster memory? (§7.1)
- Does it improve the availability? (§7.2)
- What is its overhead and sensitivity to parameters? (§7.3)
- How much TCO savings can we expect? (§7.4)
- What is its benefit over a persistent memory setup? (§7.5)

Methodology Unless otherwise specified, we use k=8, r=2, and Δ=1, targeting 1.25× memory and bandwidth overhead. We select r=2 because late binding is still possible even when one of the remote slab fails. The additional read Δ=1 incurs 1.125× bandwidth overhead during reads. We use 1GB *SlabSize*, The additional number of choices for eviction $E' = 2$. Free memory headroom is set to 25%, and the control period is set to 1 second. Each machine has 64 GB of DRAM and 2× Intel Xeon E5-2650v2 with 32 virtual cores.

We compare Hydra against the following alternatives:

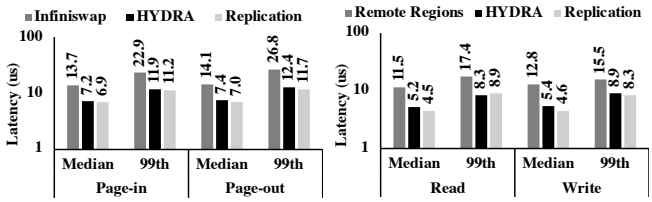
- **SSD Backup**: Each page is backed up in a local SSD for the minimum 1× remote memory overhead. We consider both disaggregated VMM and VFS systems.
- **Replication**: We directly write each page over RDMA to two remote machines' memory for a 2× overhead.
- **EC-Cache w/ RDMA**: Implementation of the erasure coding scheme in EC-Cache [76], but implemented on RDMA.

Workload Characterization Our evaluation consists of both micro-benchmarks and cluster-scale evaluations with real-world applications and workload combinations.

- We use TPC-C [16] on VoltDB [17]. We perform 5 different types of transactions to simulate an order-entry environment. We set 256 warehouses and 8 sites and run 2 million transactions. Here, the peak memory usage is 11.5 GB.
- We use Facebook's ETC, SYS workloads [21] on Memcached [12]. First, we use 10 million SETs to populate the Memcached server. Then we perform another 10 million operations (for ETC: 5% SETs, 95% GETs, for SYS: 25% SETs, 75% GETs). The key size is 16 bytes and 90% of the values are evenly distributed between 16–512 bytes. Peak memory usages are 9 GB for ETC and 15 GB for SYS.
- We use PageRank on PowerGraph [48] and Apache Spark/GraphX [49] to measure the influence of Twitter users on followers on a graph with 11 million vertices [56]. Peak memory usages are 9.5 GB and 14 GB, respectively.

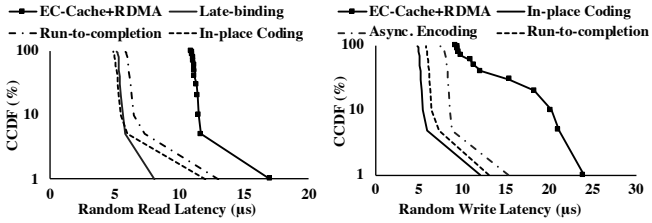
7.1 Resilience Evaluation

We evaluate Hydra both in the presence and absence of failures with microbenchmarks and real-world applications.



(a) *Disaggregated VMM Latency* (b) *Disaggregated VFS Latency*

Figure 9: ***Hydra provides better latency characteristics during both disaggregated VMM and VFS operations.***



(a) *Remote Read* (b) *Remote Write*

Figure 10: ***Hydra latency breakdown through CCDF.***

7.1.1 Latency Characteristics

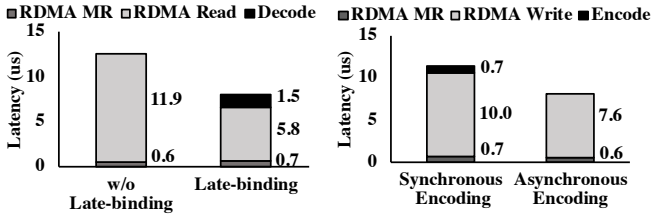
First, we measure Hydra's latency characteristics with micro-benchmarks in the absence of failures. Then we analyze the impact of its design components.

Disaggregated VMM Latency We use a simple application with its working set size set to 2GB. It is provided 1GB memory to ensure that 50% of its memory accesses cause paging. While using disaggregated memory for remote page-in, Hydra improves page-in latency over Infiniswap with SSD backup by 1.79× at median and 1.93× at the 99th percentile. Page-out latency is improved by 1.9× and 2.2× over Infiniswap at median and 99th percentile, respectively. Replication provides at most 1.1× improved latency over Hydra, while incurring 2× memory and bandwidth overhead (Figure 9a).

Disaggregated VFS Latency We use `fio` [5] to generate one million random read/write requests of 4 KB block I/O. During reads, Hydra provides improved latency over Remote Regions by 2.13× at median and 2.04× at the 99th percentile. During writes, Hydra also improves the latency over Remote Regions by 2.22× at median and 1.74× at the 99th percentile. Replication has a minor latency gain over Hydra, improving latency by at most 1.18× (Figure 9b).

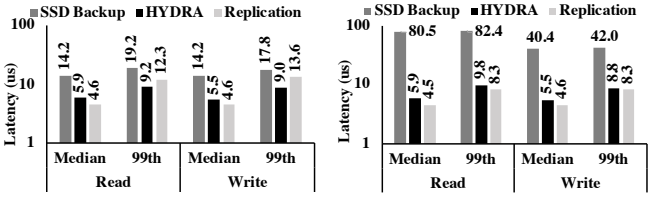
Benefit of Data Path Components Erasure coding over RDMA (i.e., EC-Cache with RDMA) performs worse than disk backup due to its coding overhead. Figure 10 shows the benefit of Hydra's data path components to reduce the latency.

1. Run-to-completion avoids interruptions during remote I/O, reducing the median read and write latency by 51%.
2. In-place coding saves additional time for data copying, which substantially adds up in remote-memory systems, reducing 28% of the read and write latency.



(a) *Read breakdown* (b) *Write breakdown*

Figure 11: ***Hydra latency breakdown at the 99th percentile.***



(a) *Background network flow* (b) *Remote Failures*

Figure 12: ***Latency in the presence of uncertainty events.***

3. Late binding specifically improves the tail latency during remote read by 61% by avoiding stragglers. The additional read request increases the median latency only by 6%.
4. Asynchronous encoding hides erasure coding overhead during writes, reducing the median write latency by 38%.

Tail Latency Breakdown The latency of Hydra consists of the time for (i) RDMA Memory Registration (MR), (ii) actual RDMA read/write, and (iii) erasure coding. Even though decoding a page takes about 1.5μs, late binding effectively improves the tail latency by 1.55× (Figure 11a). During writes, asynchronous encoding hides encoding latency and latency impacts of straggling splits, improving tail latency by 1.34× w.r.t. synchronous encoding (Figure 11b). At the presence of corruption ($r = 3$), accessing extra splits increases the tail latency by 1.51× and 1.09× for reads and writes, respectively.

7.1.2 Latency Under Failures

Background Flows We generate RDMA flows on the remote machine constantly sending 1 GB messages. Unlike SSD backup and replication, Hydra ensures consistent latency due to late binding (Figure 12a). Hydra's latency improvement over SSD backup is 1.97–2.56×. It even outperforms replication at the tail read (write) latency by 1.33× (1.50×).

Remote Failures Both read and write latency are disk-bound when it's necessary to access the backup SSD (Figure 12b). Hydra reduces latency over SSD backup by 8.37–13.6× and 4.79–7.30× during remote read and write, respectively. Furthermore, it matches the performance of replication.

7.1.3 Application-Level Performance

We now focus on Hydra's benefits for real-world memory-intensive applications and compare it with that of SSD backup and replication. We consider container-based application de-

		TPS/OPS (thousands)		Latency (ms)			
				50th		99th	
		HYD	REP	HYD	REP	HYD	REP
VoltDB	100%	39.4	39.4	52.8	52.8	134.0	134.0
	75%	36.1	35.3	56.3	56.1	142.0	143.0
	50%	32.3	34.0	57.8	59.0	161.0	168.0
ETC	100%	123.0	123.0	123.0	123.0	257.0	257.0
	75%	119.0	125.0	120.0	121.0	255.0	257.0
	50%	119.0	119.0	118.0	122.0	254.0	264.0
SYS	100%	108.0	108.0	125.0	125.0	267.0	267.0
	75%	100.0	104.0	120.0	125.0	262.0	305.0
	50%	101.0	102.0	117.0	123.0	257.5	430.0

Table 2: ***Hydra (HYD) provides similar performance to replication (REP) for VoltDB and Memcached workloads (ETC and SYS). Higher is better for throughput; Lower is better for latency.***

	Apache Spark/GraphX Completion Time (s)			PowerGraph Completion Time (s)		
	100%	75%	50%	100%	75%	50%
Hydra	77.91	105.41	191.93	73.10	66.90	68.00
Replication	77.91	91.89	195.54	73.10	73.30	73.70

Table 3: ***Hydra also provides similar completion time to replication for graph analytic applications.***

ployment [82] and run each application in an `lxc` container with a memory limit to fit 100%, 75%, 50% of the peak memory usage for each application. For 100%, applications run completely in memory. For 75% and 50%, applications hit their memory limits and performs remote I/O via Hydra.

We present Hydra's application-level performance against replication (Table 2 and Table 3) to show that it can achieve similar performance with a lower memory overhead even in the absence of any failures. For brevity, we omit the results for SSD backup, which performs much worse than both Hydra and replication – albeit with no memory overhead.

For VoltDB, when half of its data is in remote memory, Hydra achieves 0.82× throughput and almost transparent latency characteristics compared to the fully in-memory case.

For Memcached, at 50% case, Hydra achieves 0.97× throughput with read-dominant ETC workloads and 0.93× throughput with write-intensive SYS workloads compared to the 100% scenario. Here, latency overhead is almost zero.

For graph analytics, Hydra could achieve almost transparent application performance for PowerGraph; thanks to its optimized heap management. However, it suffers from increased job completion time for GraphX due to massive thrashing of in-memory and remote memory data – the 14 GB working set oscillates between paging-in and paging-out. This causes bursts of RDMA reads and writes. Even then, Hydra outperforms Infiniswap with SSD backup by 8.1×. Replication does not have significant gains over Hydra.

Performance with Leap Hydra's drop-in resilience mechanism is orthogonal to the functionalities of remote-memory frameworks. To observe Hydra's benefit even with faster in-kernel lightweight remote-memory data path, we integrate it to Leap [65] and run VoltDB and PowerGraph with 50% remote-memory configurations.

Leap waits for an interrupt during a 4KB remote I/O, whereas Hydra splits a 4KB page into smaller chunks and performs asynchronous remote I/O. Note that RDMA read for 4KB-vs-512B is 4μs-vs-1.5μs. With self-coding and run-to-completion, Hydra provides competitive performance guarantees as Leap for both VoltDB (0.99× throughput) and PowerGraph (1.02× completion time) in the absence of failures.

7.1.4 Application Performance Under Failures

Now we analyze Hydra's performance in the presence of failures and compare against the alternatives. In terms of impact on applications, we first go back to the scenarios discussed in Section 2.2 regarding to VoltDB running with 50% memory constraint. Except for the corruption scenario where we set r=3, we use Hydra's default parameters. At a high level, we observe that Hydra performs similar to replication with 1.6× lower memory overhead (Figure 13).

Next, we start each benchmark application in 50% settings and introduce one remote failure while it is running. We select a Resource Monitor with highest slab activities and kill it. We measure the application's performance while the Resilience Manager initiates the regeneration of affected slabs.

Hydra's application-level performance is transparent to the presence of remote failure. Figure 14 shows Hydra provides almost similar completion times to that of replication at a lower memory overhead in the presence of remote failure. In comparison to SSD backup, workloads experience 1.3–5.75× lower completion times using Hydra. Hydra provides similar performance at the presence of memory corruption. Completion time gets improved by 1.2–4.9× w.r.t. SSD backup.

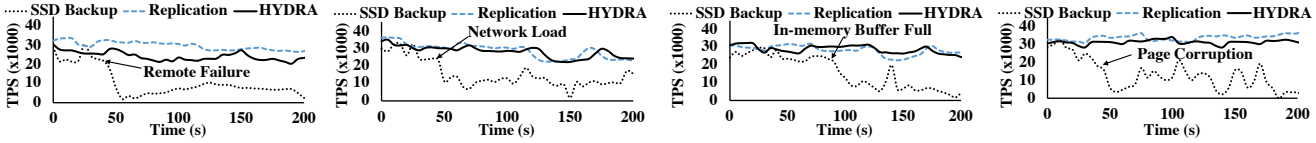
7.2 Availability Evaluation

In this section, we evaluate Hydra's availability and load balancing characteristics in large clusters.

7.2.1 Analysis of CodingSets

We compare the availability and load balancing of Hydra with EC-Cache and power-of-two-choices [67]. In CodingSets, each server is attached to a disjoint coding group. During encoded write, the $(k+r)$ least loaded nodes are chosen from a subset of the $(k+r+l)$ coding group at the time of replication. EC-Cache simply assigns slabs to coding groups comprising of random nodes. Power-of-two-choices finds two candidate nodes at random for each slab, and picks the less loaded one.

Probability of Data Loss Under Simultaneous Failures To evaluate the probability of data loss of Hydra under different scenarios in a large cluster setting, we compute the probability of data loss under the three schemes. Note that, in terms of data loss probability, we assume EC-Cache and power of two choices select random servers, and are therefore equivalent. Figure 15 compares the probabilities of loss for different parameters on a 1000-machine cluster. Our baseline comparison is against the best case scenario for EC-Cache and power-of-two-choices, where the number of slabs per



(a) *Remote failure* (b) *Remote Network Load* (c) *Request Burst* (d) *Page Corruption*

Figure 13: ***Hydra throughput with the same setup in Figure 3.***

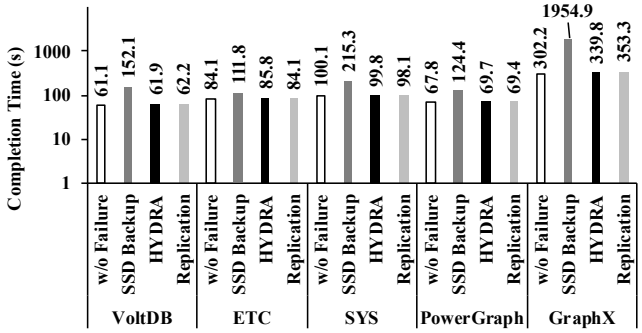
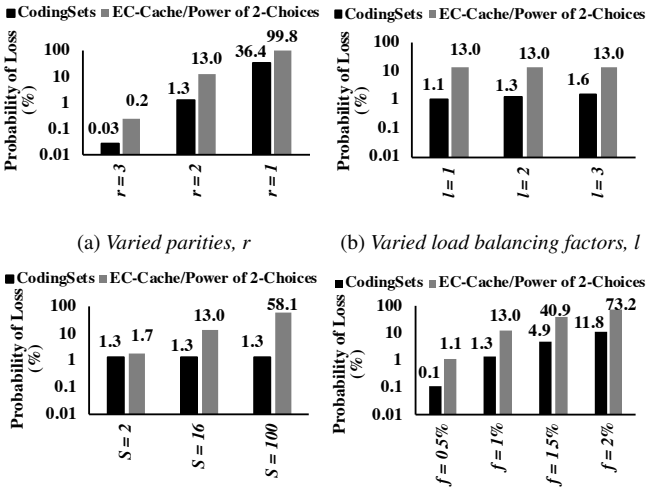


Figure 14: ***Hydra provides transparent completions in the presence of failure. Note that the Y-axis is in log scale.***



(a) *Varied parities, r* (b) *Varied load balancing factors, l*

(c) *Varied slabs per machine, S* (d) *Varied failure rate, f*

Figure 15: ***Probability of data loss at different scenarios (base parameters k=8, r=2, l=2, S=16, f=1%) on a 1000-machine cluster.***

server is low (1 GB slabs, with 16 GB of memory per server).

Even for a small number of slabs per server, Hydra reduces the probability of data loss by *an order of magnitude*. With a large number of slabs per server (e.g., 100) the probability of failure for EC-Cache becomes very high during correlated failure. Figure 15 shows that there is an inherent trade-off between the load-balancing factor (l) and the probability of data loss under correlated failures.

Load Balancing of CodingSets Figure 16 compares the load balancing of the three policies. EC-Cache's random selection of $(k+r)$ nodes causes a higher load imbalance, since some nodes will randomly be overloaded more than others. As a result, CodingSets improves load balancing over EC-Cache scheme by 1.1× even when $l = 0$, since CodingSets'

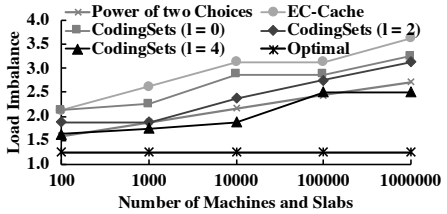


Figure 16: ***CodingSets enhances Hydra with better load balancing across the cluster (base parameters k=8, r=2).***

Latency (ms)		50th			99th		
		SSD	HYD	REP	SSD	HYD	REP
VoltDB	100%	55	60	48	179	173	177
	75%	60	57	48	217	185	225
	50%	78	61	48	305	243	225
ETC	100%	138	119	118	260	245	247
	75%	148	113	120	9912	240	263
	50%	167	117	111	10175	244	259
SYS	100%	145	127	125	249	269	267
	75%	154	119	113	17557	271	321
	50%	124	111	117	22828	452	356

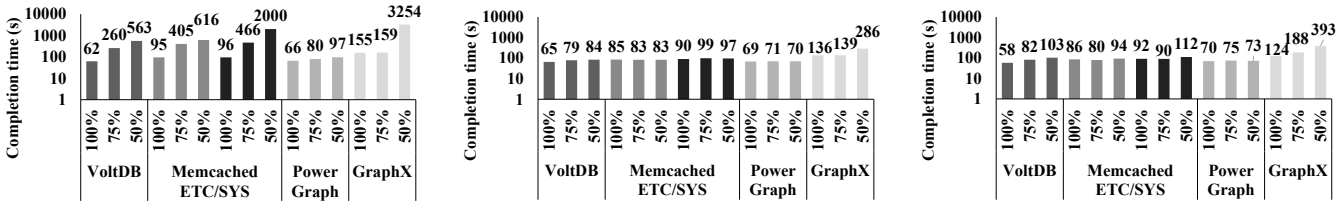
Table 4: ***VoltDB and Memcached (ETC, SYS) latencies for SSD backup, Hydra (HYD) and replication (REP) in cluster setup.***

coding groups are non-overlapping. For $l = 4$, CodingSets provides with 1.5× better load balancing over EC-Cache at 1M machines. The power of two choices improves load balancing by 0%-20% compared CodingSets with $l = 2$, because it has more degrees of freedom in choosing nodes, but suffers from an order of magnitude higher failure rate (Figure 15).

7.2.2 Cluster Deployment

We run 250 containerized applications across 50 machines. For each application and workload, we create a container and randomly distribute it across the cluster. Here, total memory footprint is 2.76 TB; our cluster has 3.20 TB of total memory. Half of the containers use 100% configuration; about 30% use the 75% configuration; and the rest use the 50% configuration. There are at most two simultaneous failures.

Application Performance We compare application performance in terms of completion time (Figure 17) and latency (Table 4) that demonstrate Hydra's performance benefits in the presence of cluster dynamics. Hydra's improvements increase with decreasing local memory ratio. Its throughput improvements w.r.t. SSD backup were up to 4.87× for 75% and up to 20.61× for 50%. Its latency improvements were up to 64.78× for 75% and up to 51.47× for 50%. Hydra's performance benefits are similar to replication (Figure 17c), but with lower memory overhead.



(a) *SSD Backup* (b) *Hydra* (c) *Replication*

Figure 17: ***Median completion times (i.e., throughput) of 250 containers on a 50-machine cluster.***

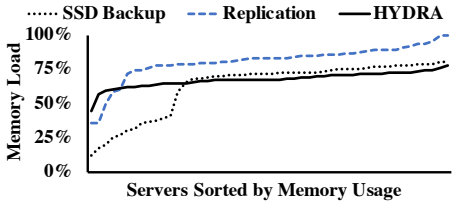
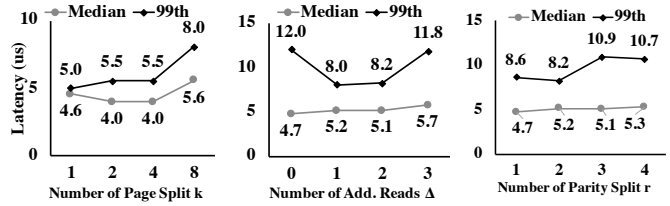


Figure 18: ***Average memory usage across 50 servers.***



(a) *r=4, Δ=1* (b) *k=8, r=4* (c) *k=8, Δ=1*

Figure 19: ***Impact of page splits (k), additional reads (Δ) on read latency, and parity splits (r) on write latency.***

Impact on Memory Imbalance and Stranding Figure 18 shows that Hydra reduces memory usage imbalance w.r.t. coarser-grained memory management systems: in comparison to SSD backup-based (replication-based) systems, memory usage variation decreased from 18.5% (12.9%) to 5.9% and the maximum-to-minimum utilization ratio decreased from 6.92× (2.77×) to 1.74×. Hydra better exploits unused memory in under-utilized machines, increasing the minimum memory utilization of any individual machine by 46%. Hydra incurs about 5% additional total memory usage compared to disk backup, whereas replication incurs 20% overhead.

7.3 Sensitivity Evaluation

Impact of (k, r, Δ) Choices Figure 19a shows read latency characteristics for varying k. Increasing from k=1 to k=2 reduces median latency by parallelizing data transfers. Further increasing k improves space efficiency (measured as $\frac{r}{k+r}$) and load balancing, but latency deteriorates as well.

Figure 19b shows read latency for varying values of Δ. Although just one additional read (from Δ=0 to Δ=1) helps tail latency, more additional reads have diminishing returns; instead, it hurts latency due to proportionally increasing communication overheads. Figure 19c shows write latency variations for different r values. Increasing r does not affect the median write latency. However, the tail latency increases from $r = 3$ due to the increase in overall communication overheads.

Monthly Pricing	Google	Amazon	Microsoft
Standard machine	$1,553	$2,304	$1,572
1% memory	$5.18	$9.21	$5.92
Hydra	6.3%	8.4%	7.3%
Replication	3.3%	4.8%	3.9%
PM Backup	3.5%	7.6%	4.9%

Table 5: ***Revenue model and TCO savings over three years for each machine with 30% unused memory on average.***

Resource Overhead We measure average CPU utilization of Hydra components during remote I/O. Resilience Manager uses event-driven I/O and consumes only 0.001% CPU cycles in each core. Erasure coding causes 0.09% extra CPU usage per core. As Hydra uses one-sided RDMA, remote Resource Monitors do not have CPU overhead in the data path.

In cluster deployment, Hydra increases CPU utilization by 2.2% on average and generates 291 Mbps RDMA traffic per machine, which is only 0.5% of its 56 Gbps bandwidth. Replication has negligible CPU usage but generates more than 1 Gbps traffic per machine.

Background Slab Regeneration To observe the overall latency to regenerate a slab, we manually evict one of the remote slabs. When it is evicted, Resilience Manager places a new slab and provides the evicted slab information to the corresponding Resource Monitor, which takes 54 ms. Then the Resource Monitor randomly selects k out of remaining remote slabs and read the page data, which takes 170 ms for a 1 GB slab. Finally, it decodes the page data to the local memory slab within 50 ms. Therefore, the total regeneration time for a 1 GB size slab is 274 ms, as opposed to taking several minutes to restart a server after failure.

To observe the impact of slab regeneration on disaggregated VMM, we run the micro-benchmark mentioned in §7.1. At the half-way of the application's runtime, we evict one of the remote slabs. Background slab regeneration has a minimal impact on the remote read – remote read latency increases by 1.09×. However, as remote writes to the victim slab halts until it gets regenerated, write latency increases by 1.31×.

7.4 TCO Savings

We limit our TCO analysis only to memory provisioning. The TCO savings of Hydra is the revenue from leveraged unused memory after deducting the TCO of RDMA hardware. We consider capital expenditure (CAPEX) of acquiring RDMA

System	Year	Deployability	Fault Tolerance	Load Balancing	Latency Tolerance
Memory Blade [61]	'09	HW Change	Reprovision	None	None
RamCloud [73]	'10	App. Change	Remote Disks	Power of Choices	None
FaRM [42]	'14	App. Change	Replication	Central Coordinator	None
EC-Cache [76]	'16	App. Change	Erasure Coding	Multiple Coding Groups	Late Binding
Infiniswap [50]	'17	Unmodified	Local Disk	Power of Choices	None
Remote Regions [18]	'18	App. Change	None	Central Manager	None
LegoOS [81]	'18	OS Change	Remote Disk	None	None
Compressed Far Memory [58]	'19	OS Change	None	None	None
Leap [65]	'20	OS Change	None	None	None
Kona [22]	'21	HW Change	Replication	None	None
Hydra		Unmodified	Erasure Coding	CodingSets	Late Binding

Table 6: ***Selected proposals on remote memory in recent years.***

hardware and operational expenditure (OPEX) including their power usage over 3 years. An RDMA adapter costs $600 [10], RDMA switch costs $318 [11] per machine, and the operating cost is $52 over 3 years [50] – overall, the 3-year TCO is $970 for each machine. We consider the standard machine configuration and pricing from Google Cloud Compute [7], Amazon EC2 [2], and Microsoft Azure [2] to build revenue models and calculate the TCO savings for 30% of leveraged memory for each machine (Table 5). For example, in Google, the savings of disaggregation over 3 years using Hydra is (($5.18*30*36)/1.25-$970)/($1553*36)*100% = 6.3%.

7.5 Disaggregation with Persistent Memory Backup

To observe the impact of persistent memory (PM), we run all the micro-benchmarks and real-world applications mentioned earlier over Infiniswap with local PM backup. Unfortunately, at the time of writing, we cannot get hold of a real Intel Optane DC. We emulate PM using DRAM with the latency characteristics mentioned in prior work [34].

Replacing SSD with local PM can significantly improve Infiniswap's performance in a disaggregated cluster. However, for the micro-benchmark mentioned in §7.1, Hydra still provides 1.06× and 1.09× better 99th percentile latency over Infiniswap with PM backup during page-in and page-out, respectively. Even for real-world applications mentioned in §7.1.3, Hydra almost matches the performance of local PM backup – application-level performance varies within 0.94–1.09× of that with PM backup. Note that replacing SSD with PM throughout the cluster does not improve the availability guarantee in the presence of cluster-wide uncertainties. Moreover, while resiliency through unused remote DRAM is free, PM backup costs $11.13/GB [14]. In case of Google, the additional cost of $2671.2 per machine for PM reduces the savings of disaggregation over 3 years from 6.3% to (($5.18*30*36)-$970-$2671.2)/($1553*36)*100% = 3.5% (Table 5).

8 Related Work

Remote-Memory Systems Many software systems tried leveraging remote machines' memory for paging [1,22,26,43,45,50,58,59,64,65,71,79], global virtual memory abstraction [15, 44, 55], and to create distributed data stores [3, 29, 30, 42, 54, 60, 73, 78]. Hardware-based remote access to memory using PCIe interconnects [61] and extended NUMA fabric [72] are also proposed. Table 6 compares a selected few.

Cluster Memory Solutions With the advent of RDMA, there has been a renewed interest in cluster memory solutions. The primary way of leveraging cluster memory is through key-value interfaces [42,52,66,73], distributed shared memory [70,75], or distributed lock [85]. However, these solutions are either limited by their interface or replication overheads. Hydra, on the contrary, is a transparent, memory-efficient, and load-balanced mechanism for resilient remote memory.

Erasure Coding in Storage Erasure coding has been widely employed in RAID systems to achieve space-efficient fault tolerance [80,90]. Recent large-scale clusters leverage erasure coding for storing *cold* data in a space-efficient manner to achieve fault-tolerance [51,69,83]. EC-Cache [76] is an erasure-coded in-memory cache for 1MB or larger objects, but it is highly susceptible to data loss under correlated failures, and its scalability is limited due to communication overhead. In contrast, Hydra achieves resilient erasure-coded remote memory with single-digit μs page access latency.

9 Conclusion

Hydra leverages online erasure coding to achieve single-digit μs latency under failures, while judiciously placing erasure-coded data using CodingSets to improve availability and load balancing. It matches the resilience of replication with 1.6× lower memory overhead and significantly improves latency and throughput of real-world memory-intensive applications over SSD backup-based resilience. Furthermore, CodingSets allows Hydra to reduce the probability of data loss under simultaneous failures by about 10×. Overall, Hydra makes resilient remote memory practical.

Acknowledgments

We thank the anonymous reviewers, our shepherd, Danyang Zhuo, and SymbioticLab members for their insightful comments and feedback that helped improve the paper. This work was supported in part by National Science Foundation grants (CNS-1845853, CNS-2104243) and a gift from VMware.

References

[1] Accelio based network block device. https://github.com/accelio/NBDX.

[2] Amazon EC2 Pricing. https://aws.amazon.com/ec2/pricing. Accessed: 2019-08-05.

[3] ApsaraDB for POLARDB: A next-generationrelational database - alibaba cloud. https://www.alibabacloud.com/products/apsaradb-for-polardb .

[4] Facebook announces next-generation Open Rack frame. https://engineering.fb.com/2019/03/15/data-center-engineering/open-rack/.

[5] Fio - Flexible I/O Tester. https://github.com/axboe/fio.

[6] Google Cloud Networking Incident 20005. https://status.cloud.google.com/incident/cloud-networking/20005.

[7] Google Compute Engine Pricing. https://cloud.google.com/compute/pricing. Accessed: 2019-08-05.

[8] Intel Intelligent Storage Acceleration Library (Intel ISA-L). https://software.intel.com/en-us/storage/ISA-L.

[9] Intel Rack Scale Design Architecture Overview. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf.

[10] Mellanox InfiniBand Adapter Cards. https://www.mellanoxstore.com/categories/adapters/infiniband-and-vpi-adapter-cards.html.

[11] Mellanox Switches. https://www.mellanoxstore.com/categories/switches/infiniband-and-vpi-switch-systems.html.

[12] Memcached - A distributed memory object caching system. http://memcached.org.

[13] Open Compute Project : Open Rack Charter. https://github.com/facebookarchive/opencompute/blob/master/open_rack/charter/Open_Rack_Charter.pdf.

[14] Pricing of Intel's Optane DC Persistent Memory. https://www.anandtech.com/show/14180/pricing-of-intels-optane-dc-persistent-memory-modules-leaks.

[15] The Versatile SMP (vSMP) Architecture. http://www.scalemp.com/technology/versatile-smp-vsmp-architecture/.

[16] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc/.

[17] VoltDB. https://github.com/VoltDB/voltdb.

[18] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *USENIX ATC*, 2018.

[19] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *SoCC*, 2017.

[20] M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN*, 2005.

[21] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[22] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, 2021.

[23] I. Calciu, I. Puddu, A. Kolli, A. Nowatzyk, J. Gandhi, O. Mutlu, and P. Subrahmanyam. Project pberry: Fpga acceleration for remote memory. HotOS, 2019.

[24] A. Carbonari and I. Beschasnikh. Tolerating faults in disaggregated datacenters. In *HotNets*, 2017.

[25] J. C. W. Chan, Q. Ding, P. P. C. Lee, and H. H. W. Chan. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *FAST*, 2014.

[26] H. Chen, Y. Luo, X. Wang, B. Zhang, Y. Sun, and Z. Wang. A transparent remote paging model for virtual machines. In *International Workshop on Virtualization Technology*, 2008.

[27] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *USENIX ATC*, 2015.

[28] A. Cidon, S. M. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *USENIX ATC*, 2013.

[29] Alexandre Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases . In *SIGMOD*, 2017.

[30] Anuj Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs . In *OSDI*, 2016.

[31] Daniel Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems . In *OSDI*, 2010.

[32] Feng Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA . In *SIGMOD*, 2016.

[33] Jeffrey Dean. Evolution and future directions of large-scale storage and computation systems at google . In *SoCC*, 2010.

[34] Joseph Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the intel optane DC persistent memory module . *arXiv preprint arXiv:1903.05714*, 2019.

[35] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. Rack-scale disaggregated cloud data centers: The dReDBox project vision . In *DATE*, 2016.

[36] Kimberly Keeton. The machine: An architecture for memory-centric computing . In *ROSS*, 2015.

[37] Krste Asanović. FireBox: A hardware building block for 2020 warehouse-scale computers . In *FAST*. USENIX Association, 2014.

[38] Robert J. Chansler. Data availability and durability with the hadoop distributed file system . *login Usenix Mag.*, 37, 2012.

[39] Wolf Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks . In *VLDB*, 2015.

[40] Yiwen Zhang, J. Gu, Y. Lee, M. Chowdhury, and K. G. Shin. Performance Isolation Anomalies in RDMA . In *KBNets*, 2017.

[41] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[42] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[43] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *IPPS/SPDP*, 1999.

[44] M. J. Feeley, W. E. Morgan, E. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath. Implementing global memory management in a workstation cluster. In *SOSP*, 1995.

[45] E. W. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, University of Washington, Mar 1991.

[46] M. D. Flouris and E. P. Markatos. The network RamDisk: Using remote memory on heterogeneous NOWs. *Journal of Cluster Computing*, 2(4):281–293, 1999.

[47] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[48] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[49] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[50] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

[51] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.

[52] A. K. M. Kaminsky and D. G. Andersen. Using rdma efficiently for key-value services. In *SIGCOMM*, 2014.

[53] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: Measurements and analysis. In *IMC*, 2009.

[54] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *SOSP*, 2017.

[55] Y. Kuperman, J. Nider, A. Gordon, and D. Tsafrir. Paravirtual Remote I/O. In *ASPLOS*, 2016.

[56] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[57] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. Coordinated and efficient huge page management with Ingens. In *OSDI*, 2016.

[58] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In *ASPLOS*, 2019.

[59] S. Liang, R. Noronha, and D. K. Panda. Swapping to remote memory over Infiniband: An approach using a high performance network block device. In *Cluster Computing*, 2005.

[60] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, 2014.

[61] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009.

[62] K. Lim, Y. Turner, J. Chang, J. Santos, and P. Ranganathan. Disaggregated memory benefits for server consolidation. 2011.

[63] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *HPCA*, 2012.

[64] E. P. Markatos and G. Dramitinos. Implementation of a reliable remote memory pager. In *USENIX ATC*, 1996.

[65] H. A. Maruf and M. Chowdhury. Effectively Prefetching Remote Memory with Leap. In *USENIX ATC*, 2020.

[66] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX ATC*, 2013.

[67] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. *Handbook of Randomized Computing*, pages 255–312, 2001.

[68] J. C. Mogul and J. Wilkes. Nines are not enough: Meaningful metrics for clouds. In *HotOS*, 2019.

[69] S. Muralidhar, W. Lloyd, S. California, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. Facebook's Warm BLOB Storage System. In *OSDI*, 2014.

[70] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, 2015.

[71] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel. Nswap: A network swapping module for Linux clusters. In *Euro-Par*, 2003.

[72] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *ASPLOS*, 2014.

[73] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[74] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high performance storage entirely in DRAM. *SIGOPS OSR*, 43(4), 2010.

[75] R. Power and J. Li. Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[76] 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*, 2016.

[77] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[78] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*, 2020.

[79] A. Samih, R. Wang, C. Maciocco, T.-Y. C. Tai, R. Duan, J. Duan, and Y. Solihin. Evaluating dynamics and bottlenecks of memory collaboration in cluster systems. In *CCGrid*, 2012.

[80] M. Sathiamoorthy, M. Asteris, D. S. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In *VLDB*, 2013.

[81] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.

[82] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with Borg. In *EuroSys*, 2015.

[83] 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 *OSDI*, 2006.

[84] M. M. T. Yiu, H. H. W. Chan, and P. P. C. Lee. Erasure coding for small objects in in-memory kv storage. In *SYSTOR*, 2017.

[85] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed lock management with rdma: decentralization without starvation. In *SIGMOD*, 2018.

[86] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *FAST*, 2016.

[87] Q. Zhang, Y. Cai, X. Chen, S. Angel, A. Chen, V. Liu, and B. T. Loo. Understanding the effect of data center resource disaggregation on production dbmss. In *VLDB*, 2020.

[88] Q. Zhang, M. F. Zhani, S. Zhang, Q. Zhu, R. Boutaba, and J. L. Hellerstein. Dynamic energy-aware capacity provisioning for cloud computing environments. In *ICAC*, 2012.

[89] Y. Zhang, Y. Tan, B. Stephens, and M. Chowdhury. Justitia: Software multi-tenancy in hardware kernel-bypass networks. In *USENIX NSDI*, 2022.

[90] Z. Zhang, Z. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does erasure coding have a role to play in my data center? Technical Report May, Microsoft Research Technical Report MSR-TR-2010-52, May 2010, 2010.

MT^2: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms

Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

Non-volatile memory (NVM) has emerged as a new memory media, resulting in a hybrid NVM/DRAM configuration in typical servers. Memory-intensive applications competing for the scant memory bandwidth can yield degraded performance. Identifying the noisy neighbors and regulating the memory bandwidth usage of them can alleviate the contention and achieve better performance. This paper finds that bandwidth competition is more severe on hybrid platforms and can even significantly degrade the total system bandwidth. Besides the absolute bandwidth, the competition is also highly correlated with the bandwidth type. Unfortunately, NVM and DRAM share the same memory bus, and their traffic is mixed together and interferes with each other, making memory bandwidth regulation a challenge on hybrid NVM/DRAM platforms.

This paper first presents an analysis of memory traffic interference and then introduces MT^2 to regulate memory bandwidth among concurrent applications on hybrid NVM/DRAM platforms. Specifically, MT^2 first detects memory traffic interference and monitors different types of memory bandwidth of applications from the mixed traffic through hardware monitors and software reports. MT^2 then leverages a dynamic bandwidth throttling algorithm to regulate memory bandwidth with multiple mechanisms. To expose such a facility to applications, we integrate MT^2 into the cgroup mechanism by adding a new subsystem for memory bandwidth regulation. The evaluation shows that MT^2 can accurately identify the noisy neighbors, and the regulation on them allows other applications to improve performance by up to 2.6× compared to running with unrestricted noisy neighbors.

1 Introduction

Emerging fast, byte-addressable NVM, such as phase-change memory (PCM) [41, 52], STT-MRAM [25, 37], Memristor [56], and Intel's 3D-XPoint [55], are promising to be employed to build fast cloud data centers. Intel Optane DC Persistent Memory, the first commercially available NVM product, has been released in 2019 [28, 33] and deployed in cloud environments, such as Google Cloud [13].

NVM has attracted many research efforts on exploring its usage scenarios. Consequently, an increasing number of NVM-aware file systems [19–21, 38, 53, 54, 60, 62, 63], NVM programming libraries [9, 70], NVM data structures [26, 42, 46, 64, 73] and NVM-based databases [5, 10, 45] have been proposed and studied, which in turns accelerates the widespread deployment of NVM. NVM is being deployed in data centers as fast byte-addressable storage or large-volume runtime memory that lies side-by-side with the volatile DRAM, resulting in hybrid NVM/DRAM platforms.

However, the hybrid NVM/DRAM platforms exacerbate the noisy neighbor problem. In cloud environments, a physical platform may be shared by many users. Applications, containers, or VMs of different users inevitably share the same memory bus on the platform. Some applications may overutilize memory bandwidth, either accidentally or intentionally, and become the noisy neighbors that significantly affect the performance of other applications. On hybrid NVM/DRAM platforms, both NVM and DRAM are attached to the memory bus. As a result, different applications compete for the limited memory bandwidth, and different kinds of memory traffic interfere with each other, reducing the overall performance of all applications on the hybrid NVM/DRAM platform.

Memory bandwidth regulation is one common approach that reduces the interference of memory bandwidth usage to mitigate the noisy neighbor problem. With the commercial use of NVM in cloud data centers, the need for memory bandwidth regulation on hybrid platforms is imminent. However, several significant challenges hinder memory bandwidth regulation on NVM/DRAM hybrid platforms.

The first challenge is memory bandwidth asymmetry. On NVM/DRAM hybrid platforms, different memory accesses (i.e., DRAM reads, DRAM writes, NVM reads and NVM writes) yield different maximal memory bandwidth. The actual available memory bandwidth heavily depends on the proportions of different kinds of memory accesses in the workload. Thus, it is no longer appropriate to assume a static maximal memory bandwidth and disregard the difference between different memory accesses like in prior work [67–69].

Especially, the maximal NVM bandwidth is usually relatively smaller than DRAM bandwidth. Besides, we find that different types of memory accesses interfere with each other differently, making it even more difficult to estimate the available memory bandwidth under various workloads. Thus, the assumption that all memory accesses are equal (as in prior work [67–69]) does not hold anymore.

The second challenge stems from the fact that NVM shares the memory bus with DRAM on existing NVM/DRAM hybrid platforms [3]. On existing hybrid platforms, NVM traffic and DRAM traffic are inevitably mixed and difficult to separate. With the mixed memory traffic, monitoring different kinds of memory bandwidth on a per-process basis become almost impossible [49], which invalidates existing hardware and software regulation approaches designed for DRAM.

The third challenge is inadequate hardware and software mechanisms for memory regulation. As both NVM and DRAM are directly accessible by CPU load/store instructions, counting and throttling each memory access is impractical for the sake of performance. CPU vendors, such as Intel, provide hardware mechanisms to regulate the memory bandwidth. However, the bandwidth restriction is coarse-grained and qualitative, which is insufficient for precise memory bandwidth regulation. Some other approaches, such as frequency scaling and CPU scheduling, may provide relatively finer-grained bandwidth adjustment. However, they are also qualitative and slow down both computation and memory accesses, thus inefficient for the overall platform performance.

In this paper, we reveal severe bandwidth interference problems in hybrid memory platforms and propose MT^2 (short for Memory Traffic Throttle) to address the above challenges. MT^2 collaboratively leverages several hardware and software techniques to monitor real-time bandwidth of different types of memory accesses. To regulate memory bandwidth with non-static maximal memory bandwidth, MT^2 proposes a dynamic memory bandwidth throttling framework, combining both hardware and software techniques to provide efficient memory bandwidth regulation.

We have implemented MT^2 as a new subsystem in the existing Linux control groups (cgroups) and applied MT^2 to mitigate the noisy neighbor problem and demonstrate MT^2's effectiveness in two more scenarios: memory bandwidth allocation and cloud SLO guarantee. Performance evaluation shows that MT^2 can effectively regulate memory bandwidth on hybrid platforms with nearly zero performance overhead.

In summary, the contributions of this paper include:

- A survey uncovering the problem of memory bandwidth interference that leads to notable performance churn for memory-intensive applications on hybrid NVM/DRAM platforms (§2);
- The first study on existing hardware and software memory bandwidth regulation mechanisms on hybrid NVM/DRAM platforms (§3.3.1);
- The design and implementation of MT^2, the first comprehensive system that efficiently and effectively regulates memory bandwidth on hybrid NVM/DRAM platforms with thread-level granularity (§3 and §4);
- Detailed evaluation of MT^2 in noisy neighbor suppression and other two scenarios (§5) on Intel Optane PM to illustrate MT^2's effectiveness and overhead (§6).

2 Background

2.1 Noisy Neighbors

In complex modern multi-tenant cloud environments, memory bandwidth can significantly impact applications' overall performance. In a cloud data center, some applications may over-utilize memory bandwidth, which will affect the performance of other applications. These applications that over-utilize memory bandwidth are usually called *noisy neighbors*, and the affected applications are the *victims*.

Two strategies can mitigate the noisy neighbor problem. The *prevention* strategy proactively sets bandwidth limits for applications to keep anyone from being a potential noisy neighbor. The *remedy* strategy monitors the system for the presence of noisy neighbors and identifies then limits the bandwidth usage of appeared noisy neighbors. Both strategies require a system to monitor applications' bandwidth usage and/or the system-wide traffic interference level and provide effective mechanisms to limit applications' memory traffic.

2.2 NVM

The release of Intel Optane DC Persistent Memory (Optane PM) marks the widespread commercial deployment of NVM [28, 33]. With Intel's proprietary DDR-T protocol [33], Optane PM can be directly accessed via CPU load/store instructions. However, the actual bandwidth of NVM is still far below DRAM [34].

Before the public release of Optane PM, NVM has been widely studied in academia and industry. Some NVM-aware file systems, such as PMFS [53], NOVA [62,63], SoupFS [21], Strata [38], SplitFS [35] and ZoFS [20], are proposed to provide file abstraction over NVM. Applications can create files on these file systems and map the files using `mmap` to access NVM directly. For example, Marathe et al. [45] modify Memcached [7], a popular high-performance memory object caching system, to run upon NVM using files and `mmap`.

However, managing persistent files by hand can be laborious. Intel develops Persistent Memory Development Kit (PMDK) [9], which is a suite of open-source libraries to simplify the programming model of NVM. With PMDK, programmers do not need to manage persistent files by themselves. Instead, they can utilize PMDK abstractions, such as objects, transactions and simple persistent data structures, to develop NVM-aware applications more easily. Many in-memory or storage systems are ported to NVM using PMDK, such as PmemKV [5] and Pmem-RocksDB [10].

NVM is increasingly deployed in data centers. For example, SAP HANA has deployed Optane PM in its data plat-

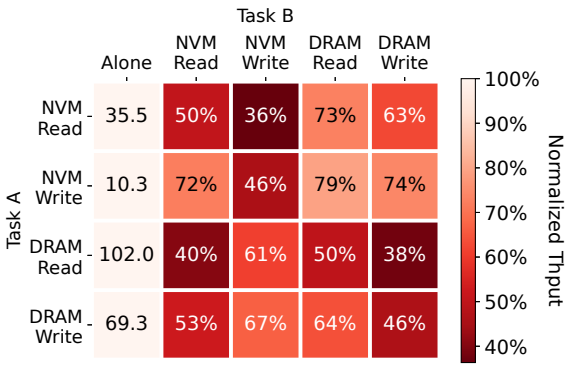


Figure 1: The impact of memory interference of two tasks. The first column is the bandwidth of Task A when it runs alone (in GB/s) and the last 4 columns are the bandwidth of Task A as a percentage of the first column when two tasks run simultaneously and compete for the bandwidth. Task B decrease Task A' throughput by 21 to 64%. The darkest block at the top row shows that NVM write bandwidth affect NVM read bandwidth significantly. Different types of bandwidth affect others differently.

forms [14]. Google Cloud has deployed Optane PM on its virtual machines in public clouds [13].

2.3 Memory Bandwidth Interference

Despite the advantages NVM has brought to the data center, the use of NVM on the hybrid NVM/DRAM platforms increases the complexity of the memory bandwidth interference due to the fact that NVM and DRAM share the memory bus.

To illustrate the impact between different types of bandwidth, we conducted an experiment in which two tasks run different kinds of workloads simultaneously. In the experiment, we run two Flexible I/O tester [2] (`fio`) workloads for the tasks to compete for the memory bandwidth. We test four workloads, namely *NVM Read*, *NVM Write*, *DRAM Read*, and *DRAM Write*, and use the mmap engine for the DRAM workloads and libpmem for the NVM workloads. The experiment setup is described in §6. To fully utilize the memory bandwidth, we use fourteen cores to run each workload, except for *NVM Write*. We use six cores to run *NVM Write* workload because its bandwidth drops significantly with more cores due to its own bandwidth competition.

We first run Task A alone and then run two tasks together with different workload combinations to illustrate the impact. To avoid the contention of CPU cache, we also leverage Intel CAT [4] to make each task run on different cache partitions in the experiment. Thus, the performance degradation in the figure is simply caused by memory bandwidth interference.

Figure 1 shows the results. For Task A (i.e., each row in the figure), the throughput (GB/s) of running alone is used as the baseline, as listed in the first column, and the throughput running simultaneously with Task B is normalized to the baseline. Thus, smaller numbers (i.e., the darker blocks) indicate a more significant impact of Task B.

We make two observations from the results.

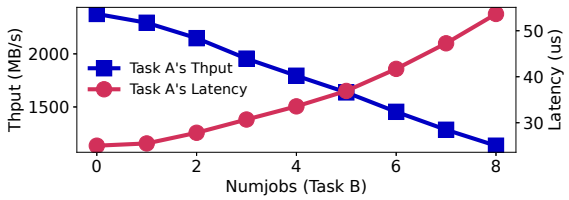


Figure 2: Relationship between bandwidth and latency of Task A (one-job NVM-Write `fio`) when running simultaneously with different number of Task B (NVM-Write `fio`). The bandwidth and latency of task A are negatively correlated. Notice that the Y-axis does not start from zero.)

1. *The impact of memory interference is closely related to the type of memory access.* Tasks that occupy a smaller bandwidth may have a more significant impact on other tasks than those with a larger bandwidth. A 102GB/s DRAM-read Task B can only reduce the bandwidth of an NVM-read Task A to 73% of the original, while an NVM-write Task B with only 10.3GB/s can bring it down to 36%. This observation indicates that the ability to distinguish between different bandwidth types is vital on hybrid platforms.
2. *NVM accesses affect other tasks more severely than DRAM accesses.* When Task B runs a 35.5GB/s NVM read workload, it drops the bandwidth of different Task A to 40%-72% (the second column in the figure) of what it would have been running alone. However, when Task B runs a 102GB/s DRAM read workload, Task A's bandwidth drops to only 50%-79% (the fourth column in the figure) of the original bandwidth. In particular, a 10.3GB/s NVM write task B can severely degrade the performance of other tasks. NVM writes can lead to severe interference with minimal absolute bandwidth, followed by NVM reads and finally DRAM accesses. In other words, applications that write NVM a lot are more likely to become the noisy neighbors and affect others.

While investigating the memory bandwidth interference, we also check the relationship between a task's throughput and latency. Figure 2 shows the throughput and latency of a one-job `fio` with the NVM-Write workload (Task A) when running simultaneously with Task B (NVM-Write `fio` with variable numjobs). As the numjobs of Task B grows, the bandwidth of Task A gradually decreases (due to the growth of bandwidth interference), while the latency of Task A increases. Together with the evaluation of other memory access type combinations, the results lead to another observation that *the memory access latency is negatively correlated to the bandwidth usage*, which indicates that we can detect the memory bandwidth interference by measuring the latency of different types of memory accesses.

2.4 Memory Bandwidth Monitoring (MBM)

Intel Memory Bandwidth Monitoring (MBM) [48] is a feature that allows monitoring bandwidth from the L3 cache to the next level of the memory hierarchy system, which can be

DRAM or NVM. It provides a hardware-level measurement of memory bandwidth on each logical core.

Each logical core can be assigned with a resource monitoring ID (RMID), and a group of logical cores can be assigned with the same RMID. The underlying hardware tracks memory bandwidth with the RMID and groups the memory bandwidth of processors with the same RMID. On a platform with the non-uniform memory access (NUMA) architecture, the MBM hardware on each NUMA node tracks two types of memory bandwidth for each RMID: the local external bandwidth and the total external bandwidth, indicating the memory traffic to the local NUMA node and all NUMA nodes respectively. System programmers can access model-specific registers (MSRs) to get the tracked bandwidth. To get the system-wide memory bandwidth of an RMID, programmers need to read the tracked total bandwidth from all NUMA nodes and add them together.

2.5 Memory Bandwidth Allocation (MBA)

Intel Memory Bandwidth Allocation (MBA) [29] is a hardware feature that provides indirect and approximate control over memory bandwidth with negligible overhead. MBA introduces a programmable request rate controller between each physical core and the shared L3 cache. The controller throttles the memory bandwidth usage by inserting delays to the memory requests. MBA defines throttling values to indicate how much delay is imposed. Due to the delay mechanism, the same throttling value might behave differently on applications with different memory access patterns [29]. The specific throttling values vary on different platforms. On our platform, the throttling values range from 10 to 100, with a precision of 10. For MBA, Intel also exposes a set of Classes of Service (CLOS) [29] into which threads can be assigned. To use MBA, administrators need to set a throttling value to a CLOS, after which all threads in the CLOS will be throttled.

3 MT^2 Design

3.1 Overview

To regulate bandwidth efficiently on a hybrid NVM/DRAM platform, we design a hybrid bandwidth regulation system called MT^2. Figure 3 shows the architecture of MT^2, which is designed to work in the kernel space. Though some functionalities of MT^2 can be implemented in user space, the kernel space environment makes it much easier and more efficient for MT^2 to access hardware features, cooperate with other kernel components, and put constraints on user-space threads.

System administrators communicate with MT^2 via exposed pseudo-filesystem interfaces in user space. Administrators can classify threads into different groups (same as cgroups) and specify a policy to regulate each group's bandwidth. We call these groups TGroups (i.e., Throttling Groups), which are the target of bandwidth monitoring and restriction in MT^2.

MT^2 consists of two parts: the monitor and the regulator. With data collected from VFS, PMU (Performance Monitoring Unit), and MBM, the monitor divides it into four types and forwards them and the interference information to the regulator. According to the monitored data and the regulation policy, the regulator makes decisions to limit the bandwidth with two mechanisms: adjusting the MBA throttling values and changing CPU quotas. MT^2 adopts a dynamic bandwidth throttling algorithm that constantly monitors and adjusts restrictions based on the real-time bandwidth and the interference level.

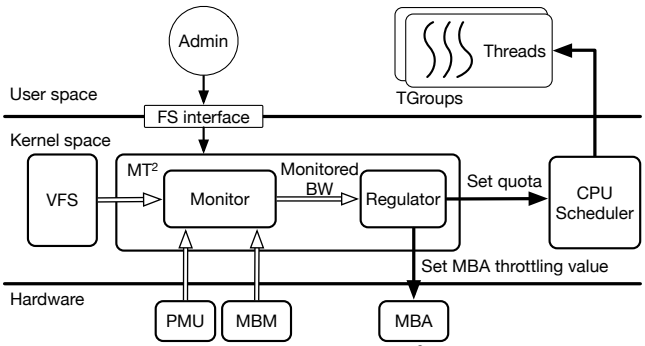


Figure 3: The overview architecture of MT^2. Threads are classified into TGroups (the unit of NVM bandwidth monitoring and restriction). The monitor computes NVM bandwidth with the data from VFS, MBM and PMU, and then pass the result to the regulator, who is responsible for restricting a TGroup's bandwidth with different mechanisms.

MT^2 provides two strategies to mitigate the noisy neighbor problem (*prevention* and *remedy* in § 2.1) to cope with different scenarios. For prevention, administrators are asked to set bandwidth caps for each TGroup. MT^2 monitors the precise real-time bandwidth and enforces all groups not to use more bandwidth than the caps. However, several TGroups that do not exceed the caps together may still cause strong bandwidth interference, which can be identified and restricted by the remedy strategy. The two strategies are orthogonal; thus, when and how to use the two strategies depends on the specific scenarios.

3.2 The Monitor

The monitor distinguishes different types of bandwidth with process granularity and detects the current memory interference level of the system. Unfortunately, existing hardware technology cannot achieve this directly [49]. For example, Intel MBM cannot distinguish between NVM bandwidth and DRAM bandwidth. The IMC performance counter [22] can help get different types of real-time bandwidth, but only with memory channel granularity rather than process granularity. Thus, the MT^2 monitor leverages various hardware and software techniques jointly.

3.2.1 Bandwidth Estimation

The monitor needs to get accurate or estimated bandwidth of each access type (i.e., BW_{DR} for DRAM reads, BW_{NR} for NVM reads, BW_{DW} for DRAM writes and BW_{NW} for NVM writes), so that the regulator can use these information to

decide whether and how to restrict each TGroup's memory bandwidth usage to avoid or suppress noisy neighbors.

MT^2 calculates the precise BW_{NR} of each process by retrieving the number of local NVM reads via the `ocr.all_data_rd.pmm_hit_local_pmm.any_snoop` PMU event counter and multiplying the value by the cache line size (64B). MT^2 calculates BW_{DR} similarly via the `ocr.all_data_rd.l3_miss_local_dram.any_snoop` PMU counter for DRAM reads of each process [30].

However, MT^2 cannot get BW_{DW} and BW_{NW} via PMU, since no similar performance events exist for write instructions. Fortunately, we can leverage MBM to monitor each TGroup's total memory access bandwidth, which is the sum of BW_{DR}, BW_{NR}, BW_{DW} and BW_{NW}. Given that we can calculate the precise value of BW_{DR} and BW_{NR} via PMU, we only need to know one of BW_{DW} and BW_{NW} or the ratio between them to calculate the two values via simple arithmetic.

We choose to calculate BW_{NW} of a TGroup by collecting the amount of NVM writes periodically since user-space applications can write to NVM in only two ways: the file APIs (such as `write`) and the CPU `store` instructions after memory mapping the file. For file APIs, MT^2 hooks the VFS in the kernel and tracks the amount of NVM writes for each TGroup. For memory-mapped accesses, we propose two different approaches according to whether the applications on the platform are trusted.

Trusted applications. Many cloud applications (such as those in private clouds) are from trusted users or cooperations; thus, we can rely on these trusted applications to collect and report to MT^2 its amount of writes to memory-mapped NVM. To facilitate the process, we provide a modified PMDK [9], which is the official and most popular library for NVM programming on Intel's NVM. Specifically, we hook the PMDK APIs that explicitly flush cache lines to NVM or perform non-temporal memory writes (e.g., `movnt`), by calculating and adding the amount of NVM writes to per-thread counters. To report the counters to MT^2, each process sets up a shared page with the kernel, and each thread in the process writes its per-thread counter value to a different slot in the page. MT^2 in the kernel checks the counters periodically and calculates the bandwidth of each TGroup.

To collaborate with MT^2, applications built on PMDK can directly link to our modified PMDK without source code modification; other applications are required to collect and report NVM writes by themselves, which should be a simple task since our modification to PMDK is merely 43 lines.

With the reported writes to mapped NVM and the NVM writes via file APIs, MT^2 calculates BW_{NW} for each TGroup. With the BW_{NW}, MT^2 further calculates the BW_{DW} by $BW_{DW} = BW_{Total} - BW_{DR} - BW_{NR} - BW_{NW}$.

Untrusted applications. Untrusted applications may not report their NVM write bandwidth faithfully. Thus, we provide another approach to roughly distinguish NVM writes and DRAM writes without the collaboration of applications.

We leverage Processor Event Based Sampling (PEBS) [31], an efficient sampling feature in modern Intel processors, to sample each TGroup's memory writes (`mem_inst_retired.all_stores`) with the target addresses. By comparing the sampled addresses to the address ranges of NVM, we can figure out the ratio of sampled writes to NVM and DRAM, with which we calculate BW_{NW} and BW_{DW} roughly.

Note that the BW_{NW} and BW_{DW} we calculated via PEBS are not precise due to the shadow effect [65]. But it would be sufficient for MT^2 to identify which TGroup is more likely to be the noisy neighbors.

3.2.2 Interference Detection

Even given the accurate bandwidth usage of four types of memory accesses, it is difficult to determine whether the bandwidth interference occurs and its severity, since both the decrease of memory access demand and the presence of noisy neighbors can cause an application to utilize less bandwidth.

Instead of detecting memory interference via memory bandwidth, MT^2 proposes to detect the interference level by measuring the latency of different kinds of memory accesses, which is supported by the observation that the memory access latency is negatively correlated to the bandwidth (§2.3).

We measure four types of memory accesses separately. For reads, we derive the latency from four performance events, `unc_m_pmm_rpq_occupancy.all` (RPQ_O), `unc_m_pmm_rpq_inserts` (RPQ_I), `unc_m_rpq_occupancy`, and `unc_m_rpq_inserts`. The latency of NVM reads can be calculated by RPQ_O/RPQ_I. The DRAM read latency can be obtained similarly. For writes, MT^2 periodically issues a few NVM and DRAM write requests and measures their completion time to obtain the latency of both types of write requests.

We set a threshold to determine whether bandwidth interference occurs. When the latency of a certain access request exceeds the corresponding threshold, relatively severe interference occurs on the platform and affects this type of memory access (as shown in Listing 1). The threshold can be tuned across different platforms by measuring the relationship between bandwidth and latency under different interference levels. On our platform, we use the latency at a 10% reduction in throughput as the threshold (THRESHOLD in Listing 1).

Listing 1: Interference detection

```
def detect_interference():
  for bt in bandwidth_type:
    if latency[bt] > THRESHOLD[bt]:
      return true
  return false
```

3.3 The Regulator

Monitoring the bandwidth information is the first step towards bandwidth regulation. The following step is to restrict the bandwidth a TGroup can occupy, which is handled by the regulator. The regulator takes the interference level and the

Table 1: Performance events

Performance Event Name	Description	Where we use (if not, why)
ocr.all_data_rd.pmm_hit_local_pmm.any_snoop	per-core: local NVM read	Bandwidth Estimation (§3.2.1)
mem_load_retired.local_pmm	per-core: memory load instructions retired that hit local NVM	No, the results are not precise without disabling the hardware prefetcher
ocr.all_data_rd.l3_miss_local_dram.any_snoop	per-core: local dram read	Bandwidth Estimation (§3.2.1)
mem_inst_retired.all_stores	per-core: all memory store instruction retired	Bandwidth Estimation (§3.2.1)
unc_m_pmm_rpq_occupancy.all	per-socket: NVM read pending queue occupancy	Interference Detection (§3.2.2)
unc_m_pmm_rpq_inserts	per-socket: NVM read pending queue inserts	Interference Detection (§3.2.2)
unc_m_pmm_wpq_occupancy.all	per-socket: NVM write pending queue occupancy time	No, wpq_occupancy/wpq_inserts is not inversely proportional to bandwidth
unc_m_pmm_wpq_inserts	per-socket: NVM write pending queue insert count	No, wpq_occupancy/wpq_inserts is not inversely proportional to bandwidth

monitored bandwidth as the input and decides what actions to take to adjust the bandwidth of the TGroup according to the regulation policy from system administrators.

3.3.1 Memory Regulation Mechanisms

MBA. To illustrate the effect of MBA, we use `fio` to generate different workloads under different MBA throttling values. Throttling value 100 means that there are no restrictions, while 10 represents the maximum MBA limit. The configuration of `fio` is the same as in §2.3.

The red lines in Figure 4 show the following phenomena. 1) MBA only supports limited throttling values, and not all throttling values are effective to workloads. This means that we cannot precisely control the bandwidth of threads with MBA. 2) MBA can restrict DRAM-intensive workloads better than NVM-intensive workloads. MBA is almost completely ineffective for NVM writes. Therefore, MBA alone is insufficient for controlling memory bandwidth. We must employ other techniques to restrict the NVM bandwidth.

CPU scheduling. An effective mechanism to control memory bandwidth is to reduce the number of cores allocated to an application [43]. We take a finer-grained approach by changing the CPU time (or CPU quota) of a thread with the help of the existing Linux CPU cgroup [6] controls. CPU quota in MT^2 defines an upper bound on CPU time allocated to the threads of a TGroup within a given period. TGroups with lower CPU quota take less CPU time, so it consumes less memory bandwidth. Since CPU quota leverages the CPU scheduler, it can provide a more fine-grained adjustment of memory bandwidth.

Effectiveness and comparison. CPU scheduling is a mechanism that supplements MBA. We repeat the same experiment with CPU scheduling as we do with MBA to compare these two mechanisms to decide how to cooperate better. Figure 4 shows the results. Take reading DRAM in figure 4(a) as an example (Read MBA and Read CPU two lines in the figure). When we don't enforce any limits on the workload (i.e., when the horizontal coordinate is 100), the throughput of 14 DRAM read `fio` workloads reachs 102GB/s on our platform. When the MBA throttling value keeps decreasing to 60, the throughput of the DRAM read does not change much. When this value decreases to 30, there is a significant drop in throughput. After we enforce the maximum limit via MBA, the throughput drops to about 28GB/s. For CPU scheduling, the throughput is proportional to the available CPU time. Thus, CPU scheduling can restrict memory bandwidth better than MBA.

Table 2: Execution time and max bandwidth of pagerank under different restrictions

	No limit	50% CPU	10% MBA
Execution Time(s)	56.459	118.179	78.662
Max BW_{read}(GB/s)	3.61	1.89	1.45
Max BW_{write}(GB/s)	4.84	2.53	1.87

The previous experiment is only for the effectiveness of bandwidth restriction. Table 2 gives some data when we run a real-world application, pagerank. We run the same task in three different situations. When this task runs without any limits, it takes 56 seconds to complete, of which the maximum read and write bandwidth is 3.61GB/s and 4.84GB/s, respectively. When we only allow it to use 50% of the CPU time, the task consumes 118 seconds, with the read and write bandwidth dropping to 1.89GB/s and 2.53GB/s. It seems that bandwidth usage is indeed changed to half while spending almost twice the original time. After we apply the maximum limit with MBA (10% MBA), its peak bandwidth is lower than that in 50% CPU, but it performs faster. This is because CPU scheduling will reduce the amount of CPU time the program can use. In contrast, MBA slows down the memory access operations and does not affect other operations, such as computation operations. We can conclude that the MBA mechanism is more efficient than CPU scheduling in restricting memory bandwidth.

Table 3 summarizes the characteristics of these two memory restriction mechanisms. As MBA has limited throttling values, it can only provide discrete restrictions on memory bandwidth. CPU scheduling can adjust memory bandwidth continuously, which could restrict the bandwidth finer. However, CPU scheduling is not as efficient as MBA since it is not friendly to the overall performance. At last, these mechanisms have different favor in memory access types. According

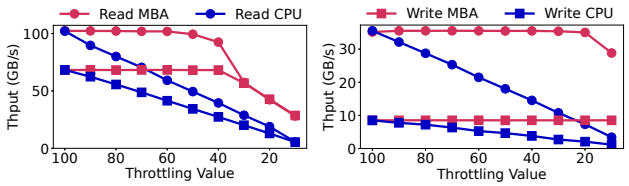


Figure 4: The effect of MBA and CPU scheduling on limiting DRAM (a) and NVM (b) bandwidth. Compared with MBA, CPU scheduling is more effective on restricting NVM bandwidth.

Table 3: Comparison of different memory restriction mechanisms

Mechanism	Granularity	Efficiency	Favor
MBA	Discrete	High	DRAM
CPU scheduling	Continuous	Low	Both

to Figure 4, MBA is good at restricting DRAM bandwidth, while CPU scheduling can cope with both DRAM and NVM bandwidth because it treats DRAM and NVM equally.

3.3.2 Dynamic Bandwidth Throttling

To ensure a relatively stable bandwidth for TGroups, we adopt an algorithm called dynamic bandwidth throttling that combines all mechanisms above. The algorithm first identifies noisy neighbors according to the information provided by the monitor and then takes actions to restrict the noisy neighbors' memory bandwidth.

Identifying the noisy neighbors. The algorithm will identify the noisy neighbors according to the enabled strategies. In the *prevention* strategy, the algorithm treats all TGroups that exceed their administrator-configured bandwidth limits as noisy neighbors. In the *remedy* strategy, the algorithm first checks whether there is severe memory interference on the platform according to the memory interference information provided by the monitor. If severe memory interference presents, the algorithm identifies noisy neighbors by each TGroup's current bandwidth use. According to the observations in the previous analysis (§2.3), a small amount of NVM writes can lead to severe bandwidth interference and NVM accesses affect others more severely than DRAM accesses. Thus, TGroups with the most NVM writes are more likely to become the noisy neighbors, followed by TGroups with most NVM reads, and finally the TGroups with more DRAM accesses. The algorithm picks the TGroup that is the most likely to be a noisy neighbor in the above order.

Regulating the memory bandwidth. The algorithm then chooses the memory regulation mechanism according to the types of memory bandwidth to restrict. To restrict NVM access bandwidth, the algorithm takes the CPU scheduling mechanism since MBA is almost ineffective for NVM. To restrict only DRAM access bandwidth, the algorithm chooses to decreases the MBA value of the target TGroups. If the MBA is already set to the lowest value, the algorithm uses CPU scheduling for further restriction.

Relaxing the memory regulation. Once the memory interference disappears, the algorithm attempts to relax the enforced bandwidth restrictions. The procedure is opposite to the way we add and enforce the restriction.

After the regulator finishes a single step of the algorithm (i.e., identifying then regulating/relaxing), it continues to wait for the next period in which another step will be taken according to the new information provided by the monitor. The step-by-step approach reduces the uncertainty of platform memory bandwidth changes and prevents unnecessary performance jitters for applications.

4 Implementation

As control groups (cgroups) [6] is an existing Linux kernel feature that manages resource usage of a collection of threads, we modify the Linux kernel 5.3.11 to add TGroup as a subsystem of cgroups. MT^2 is implemented as a kernel module that cooperates closely with the TGroup subsystem.

Cgroup interface. Cgroups exposes its interfaces via files in a pseudo-filesystem called cgroupfs. MT^2 follows the same approach as other cgroups subsystems. Specifically, an administrator first mounts the subsystem and creates a new directory in the subsystem mount point (i.e., creating a new TGroup). Then the administrator writes the pid of the process to the *cgroup.procs* file (i.e., adding the process to the TGroup). Three more files in this directory can be read/written to manage the TGroup:

1. The *priority* file is used to get and set the priority of a TGroup. Two priorities are currently supported. TGroups with *high* priority will not be restricted by the regulator, while the *low*-priority TGroups will be limited when interferences occur in the system.
2. The *bandwidth* file is read-only and returns the bandwidth of a TGroup for the last second.
3. The *limit* file is used to get and set the absolute bandwidth limit of a TGroup. Four comma-separated numbers can be written to this file as upper bandwidth limits of four types of memory accesses. When any one of the limits is exceeded, the TGroup will be throttled. A zero value indicates no limit, and the values take effect immediately.

When the write bandwidth cannot be separated accurately, only the first interface is valid. When the measured bandwidth is accurate, the last two can be used for prevention (§ 2.1). In this case, MT^2 allows the administrator to set four caps for four types of bandwidth for each TGroup. Once the real-time bandwidth used by one TGroup exceeds its limit, MT^2 enforces restrictions on that group, ensuring that each group does not use more bandwidth than the preset cap.

Thread creation. All the child processes are put in the same TGroup as their parent when created unless the administrator puts them manually into another TGroup. To achieve this, we also add a hook to the process/thread creation routine, which is the fork routine in the Linux kernel.

MBA. The MBA hardware supports ten MBA throttling values. However, there are only eight CLOS available on our platform. To support as many TGroups as possible, we do not assign a dedicated CLOS for each TGroup. Instead, we assign eight MBA throttling values to eight CLOS, respectively. We omit MBA throttling values 70 and 80 because the effect of the MBA throttling values 70, 80, and 90 are very similar across all workloads in Figure 4. As a result, each CLOS presents a different MBA throttling value. To restrict the bandwidth of a TGroup to an MBA throttling value, we assign all threads of the TGroup to the corresponding CLOS. Thus, by changing the CLOS of a TGroup, we can change its MBA throttling value. Since the MBA limits the memory bandwidth by adjusting the request rate between the physical core and the shared LLC, TGroups with the same CLOS get the same request rate without interference.

Context switches. To set up the MT^2 context for each thread, including setting the PMU related context, writing the MSR registers related to MBA and setting CPU quota, we add a hook to the scheduler. Each time a context switch happens, we set up the corresponding MT^2 context for the new thread that is going to run on this CPU core.

PMU. PMU is used to count read instructions that miss all caches and access the NVM and DRAM respectively. Using these data we are able to accurately calculate the DRAM and NVM read bandwidth. The latency of both types of read operations is also obtained through the PMU.

PEBS. We set the PEBS sample frequency to 10,007; thus, PEBS will record one linear address for every 10,007 events. As later evaluated in §6.2.2, this sample frequency is large enough to avoid noticeable overhead.

During context switches or PEBS interrupts occur, MT^2 reads all the samples in the PEBS buffer and filters out addresses in the kernel and the user stacks to mitigate the interference of irrelevant accesses and the CPU cache. MT^2 then translates the addresses to physical addresses and counts the number of NVM accesses and DRAM accesses, respectively. Finally, MT^2 stores the numbers in per-thread data structures, which will be used to estimate NVM bandwidth usage in the untrusted environment.

Listing 2: Kernel thread main loop

```
def kthread_main():
  start = current_time()
  interference = detect_interference()
  for group in TGroups:
    group.aggregate_bandwidths()
    group.adjust_bandwidths(interference)
  sleep(INTERVAL - (current_time() - start))
```

The dedicated kernel thread. A kernel thread is created at the initialization phase of MT^2 kernel module to periodically detect the interference, track the bandwidth and take actions generated by the dynamic bandwidth throttling algorithm. When interference is detected, the kernel thread calculates all types of bandwidth of all TGroups via information from MBM, VFS, and PMU. It then invokes the dynamic bandwidth throttling algorithm to adjust the bandwidth. The kernel thread runs at a configurable frequency (INTERVAL in Listing 2), which is once per 100ms in our implementation.

5 Other Use Cases

In addition to being used to prevent severe bandwidth interference caused by the noisy neighbors, MT^2 can also be used in more scenarios, such as memory bandwidth allocation, and cloud SLO guarantee.

5.1 Memory Bandwidth Allocation

For prevention, choosing and setting the maximum bandwidth for each application is a practical problem. A more reasonable solution in practice is the bandwidth guarantee, which assigns a minimum guarantee bandwidth to each task. As long as there is such a guarantee, a task will be able to use more bandwidth than this minimum guarantee when a task needs to use bandwidth, regardless of how much bandwidth other tasks are using at the same time.

Bandwidth allocation is essentially the same as bandwidth limiting since bandwidth resources are finite. The only method to reserve a minimum bandwidth for a program is to ensure that no other programs can consume excessive bandwidth resources. However, since the bandwidth resources in a hybrid system are not fixed, it is very difficult to give such a guarantee. We build an empirical model of four kinds of bandwidths on our platform to help us solve this problem. The input is the four bandwidths without interference, while the output is the actual bandwidths running simultaneously.

In this use case, each group of programs needs to predeclare their demand for each kind of bandwidth. Then we add up the demanded bandwidths of all the programs and pass to the empirical model to calculate the intensity of bandwidth competition. If the bandwidth competition is sufficiently intense, the minimum bandwidth for these programs cannot be guaranteed at the same time, and MT^2 will report an incident. If the bandwidth competition is low, we look for a point in the model where the system's bandwidth resources can be more fully utilized without excessive bandwidth competition (in our implementation, 90% of the desired value is considered to be no excessive bandwidth competition). Then MT^2 allocates the extra bandwidth resources proportionally to all programs, in such a way that each group of programs is guaranteed to use more than 90% of its own declared bandwidth.

5.2 Cloud SLO Guarantee

Service Level Objective (SLO) assurance is important for cloud users [18, 51]. For example, for users deploying KV-store applications, which primarily use memory bandwidth resources, the latency and the throughput of GET and PUT requests are what they value most. However, the request pattern of latency-critical (LC) tasks may not be fixed. There may not be any requests for a while, but at the next moment, the

requests become very intensive.

Cloud service providers want to make their devices as highly utilized as possible. When KV-store requests are not frequent, we can run some other best-effort (BE) background tasks simultaneously to make full use of the physical machine's bandwidth resources. When the foreground requests are dense, we can dynamically reduce the background tasks' bandwidth to ensure the SLO of the foreground tasks.

For trusted environment, we can use the bandwidth limiting (prevention) to prevent the BE applications from becoming noisy neighbors. We can divide the tasks into two TGroups: a high-priority TGroup for tasks that require a guaranteed SLO and a low-priority TGroup for other tasks. MT^2 first gives the foreground tasks a relatively smaller bandwidth guarantee. When the foreground tasks are about to run out of the allocated bandwidth, MT^2 allocates more bandwidth for them while reclaiming the bandwidth resources owned by the background BE tasks.

For untrusted environment, the two types of write bandwidth cannot be precisely separated. We can assign high priority to LC applications and low priority to the BE applications to mitigate the interference when the BE applications overuse memory bandwidth. This case is then transformed into remedy in noisy neighbor suppression.

6 Evaluation

In this section, we comprehensively evaluate MT^2 from multiple dimensions, including effectiveness for all use cases,the performance overhead, and the accuracy when the environment is trusted.

Experiment setup. Experiments are conducted on a server with two 28-core Intel® Xeon® Gold 6238R CPUs with hyper-threading disabled. The server has two NUMA nodes, and each is equipped with 6*32GB DDR4 DRAM and 6*128GB Optane™ PM configured in interleaved app-direct mode. All experiments are conducted on a single NUMA node.

6.1 Effectiveness

In this part, we evaluate the effectiveness of our three use cases: noisy neighbor suppression, memory bandwidth allocation and cloud SLO guarantee.

6.1.1 Noisy Neighbor Suppression

Effect of restrictions on noisy neighbors. We first re-conduct the experiment in §2 (results are shown in Figure 1) with and without MT^2 respectively to show the effect of MT^2 in micro-benchmarks. In this experiment, we run two `fio` simultaneously, one is marked as the noisy neighbor and the other as the victim. The throughputs when the victim runs alone are used as the baselines.

The results are shown in Figure 5. Generally, the columns with MT^2 have a much lighter color than the columns without MT^2, which indicates that MT^2 can effectively reduce noisy neighbors' interference by restricting their bandwidth

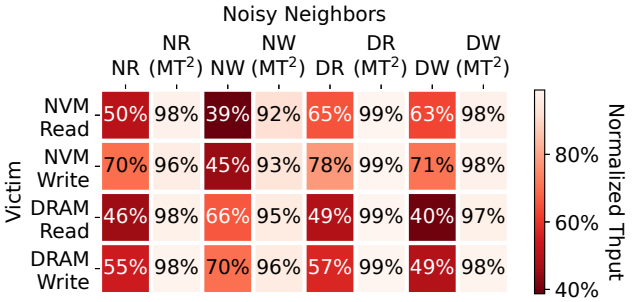


Figure 5: The normalized throughput of `fio` with/without MT^2. Noisy neighbors decrease the victim's throughput heavily while MT^2 can benefit the victim by restricting noisy neighbors' bandwidth.

usage. NVM Read noisy neighbor is abbreviated to NR in the following figures, and the others are similar. Take the NVM-Read workload as an example, four kinds of noisy neighbors decrease `fio`'s throughput to 50%, 39%, 65%, and 44% of the baseline. By restricting noisy neighbors' bandwidth with MT^2, `fio`'s throughput recovers to 98%, 92%, 99%, and 98% of the baseline. The other workloads present similar phenomena. The victim can run with a nearly maximal throughput with the help of MT^2.

Applications We evaluate three real-world applications, Hadoop [1], Graphchi [39] and Pmem-RocksDB [10], to check the effectiveness of MT^2. The computing tasks of these applications are conducted on DRAM, while the data is stored on NVM. Consequently, these applications will access both NVM and DRAM at the same time. The number of the `fio` noisy neighbors are the same as the configuration in §2.

On Hadoop 2.10.0 and Graphchi, we run a page-rank job on Twitter [11] social graph with 81,306 nodes and 1,768,149 edges. The iteration count of the page-rank is set to 3. Figure 6 gives the results. We treat the execution time when the victim application runs alone as the baseline. For both applications, MT^2 can mitigate the victim's performance slowdown well by restricting the noisy neighbors' bandwidth.

YCSB [12] is used to evaluate the throughput of RocksDB [8]. Before running the benchmark, we load 500,000 records, each with the size of 1KB, into RocksDB. Besides `fio`, we also use the aforementioned Graphchi with eight long jobs as the noisy neighbors (denoted by Graph in the figure). Figure 7 shows the results. Generally, the throughput of RocksDB with noisy neighbors is 61% to 77% of that without any noisy neighbors. When the noisy neighbors are restricted, RocksDB's throughput rises to about 94% to 100% of the original throughput. This indicates that MT^2 effectively reduces or even eliminates the impact of noisy neighbors to improve the high-priority application's bandwidth.

Response of limit update. A fast and accurate bandwidth limiting is the foundation of prevention. Figure 8 shows the applications' response to the update of the bandwidth limit in MT^2 at runtime. In this experiment, we run a six-job NVM Write `fio` and a fourteen-job NVM Read `fio` simultaneously.

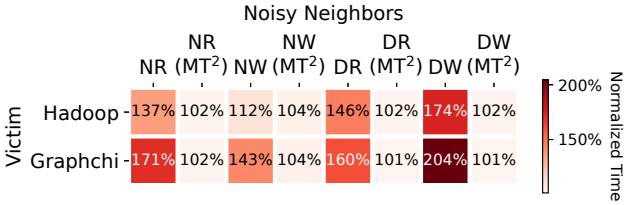


Figure 6: The normalized execution time of Hadoop and Graphchi when running page rank on Twitter social graph. Noisy neighbors slow down the execution and MT^2 mitigates the impact by restricting noisy neighbors' NVM bandwidth.

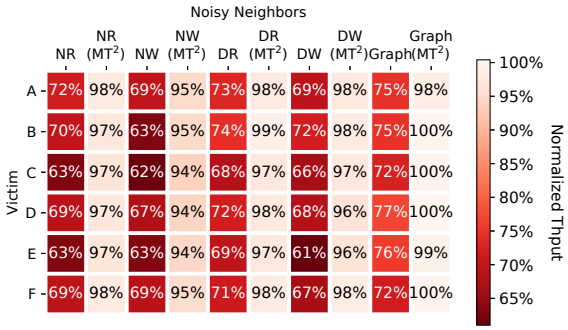


Figure 7: The throughput of YCSB's different workloads on RocksDB when running with/without MT^2 against eight noisy neighbors. Noisy neighbors decrease the throughput and MT^2 mitigates the impact by restricting noisy neighbors' bandwidth.

The concepts of victims and noisy neighbors are relative. We assume the former as the neighbor and the other as the victim. At first, the victim and neighbor run together without any restrictions. They can reach the throughput of 12.5GB/s and 7.2GB/s, respectively. After 5 seconds, we set the noisy neighbor's NVM write bandwidth limit to 5GB/s. It takes no more than one second that the noisy neighbor's throughput drops to 5GB/s, and the victim's throughput rises to 20GB/s due to less bandwidth interference. Similar results appear when noisy neighbors' NVM bandwidth limit is changed to 1GB/s after 15 seconds, 3GB/s after 25 seconds and unlimited after 35 seconds. The evaluation result shows that MT^2 can adjust a TGroup's throughput accurately and timely.

This also illustrates that it is not just applications that use plenty of bandwidth that can become noisy neighbors. An application that uses relatively small amounts of NVM bandwidth can have a significant impact on other applications. So the ability to distinguish between different types of bandwidth is critical in the hybrid NVM/DRAM platforms.

6.1.2 Memory Bandwidth Allocation

We run four `fio` tasks with different memory access patterns individually and record their throughputs. Then we run all four tasks simultaneously without MT^2 and with different guarantees with MT^2. As shown in Table 4, DRAM Write and NVM Read suffer the most severe bandwidth degradation when run together. DRAM Write has a whopping 66% drop in throughput (from 7.4GB/s to 2.5GB/s). We then assign

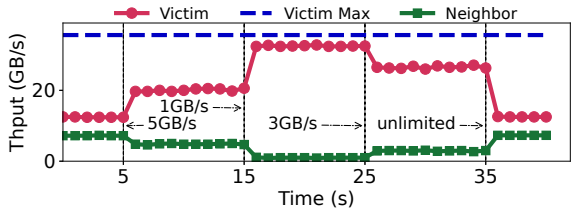


Figure 8: Response of the victim's and noisy neighbors' throughput when NVM-write intensive noisy neighbors' NVM write bandwidth limit is updated at run time. The limit is changed at 5s, 15s, 25s and 35s. When a lower limit is put on noisy neighbors, its throughput decreases and the victim's throughput increases and vice versa. The adjustment takes no more than 1 second and is very accurate.

Table 4: The throughput of `fio` tasks under BW allocation

Thput(GB/s)	Alone	w/o MT^2	Config 1	Config 2
DRAM Read	100	69.8	28.8(20)	11.5(10)
DRAM Write	7.4	2.5	5.3(5)	4.2(4)
NVM Read	7.2	3.4	4.2(4)	5.3(5)
NVM Write	5.0	3.8	4.5(4)	3.3(3)

different bandwidth guarantees (as indicated by the numbers in parentheses in the table) to these tasks, which are satisfied under the regulation of MT^2.

6.1.3 Cloud SLO Guarantee

We conduct three experiments to verify the effectiveness of the SLO guarantee. The first is a micro-benchmark that shows a breakdown of DRAM/NVM read/write bandwidth changes of both foreground tasks and background tasks. The second is a macro-benchmark which evaluates the 95th percentile latency of several LC tasks when running simultaneously with Graphchi [39] as the BE task. These two correspond to the first method in § 5.2 (similar to prevention). For the other method (like remedy), the third experiment is conducted. We run YCSB as the LC task with different types of memory accesses generated by `fio` to simulate different BE tasks.

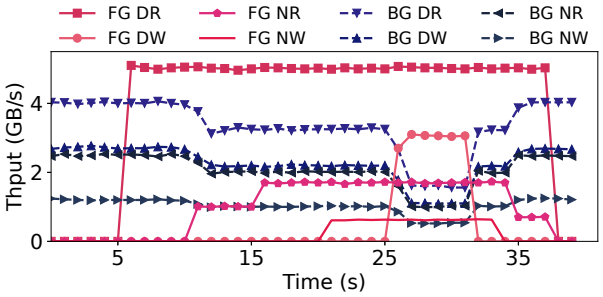


Figure 9: The throughput of the foreground and background (dashed lines) tasks. As foreground tasks use more and more bandwidth resources, if there are no sufficient bandwidth resources in the system, MT^2 will reduce the bandwidth for background tasks. When the foreground task reduce its memory usage, MT^2 will restore the bandwidth resources of the background tasks.

Table 5: The 95th percentile latency of several LC tasks

Workloads	w/o BEs	w/o MT^2	w/ MT^2
`img-dnn (ms)`	5.719	99.656	5.661
`masstree (ms)`	0.991	1.956	1.056
`YCSB-A-read (us)`	39	81	42
`YCSB-A-update (us)`	57	103	59

Breakdown of bandwidth changes. Figure 9 shows the result of dynamic bandwidth changes of foreground and background tasks. We use `fio` that read DRAM and NVM to act as the foreground workloads. For the background tasks, we choose all kinds of `fio` to show the impact to the hybrid bandwidth. The solid lines represent the bandwidth of foreground tasks, while the dashed lines represent background tasks. Before the beginning, MT^2 reserves 1GB/s bandwidth for each of the four types of memory accesses for the foreground tasks and allocate all the remaining bandwidth to the background tasks. After excluding the bandwidth reserved for foreground tasks, MT^2 lookups the empirical model and selects an appropriate bandwidth cap for background tasks to ensure that foreground tasks will not be affected until they use more bandwidth than reserved.

At first, the background tasks normally run with 4GB/s DRAM read, 2.7GB/s DRAM write, 2.5GB/s NVM read, and 1.2GB/s NVM write bandwidth consumption. After 5 seconds, a foreground task starts to read DRAM and takes 5GB/s DRAM read bandwidth. MT^2 increases the DRAM read bandwidth reservation for the foreground tasks to 6GB/s. As the existing four kinds of bandwidths do not exceed the limitation, MT^2 does not restrict the background tasks. Then the foreground tasks start to read NVM after 10 seconds. MT^2 finds that the foreground tasks occupy 1GB/s NVM read bandwidth, which exceeds 90% of the reservation, and assumes they may need extra NVM read bandwidth. So MT^2 lowers the NVM read bandwidth cap for the background tasks by 1GB/s, and then the NVM read bandwidth of background tasks exceeds the limit. As a result, MT^2 decides to tighten the restriction on background tasks. After 16 seconds, foreground tasks read NVM at 1.7GB/s, which is less than 90% of 2GB/s. Hence no additional NVM read bandwidth needs to be added to meet the SLO guarantee, i.e., MT^2 will not put more restrictions on background tasks.

The following bandwidth changes are all caused by the same reasons. With more bandwidth being taken by foreground tasks, the background tasks can use less bandwidth, and MT^2 puts more restrictions on them to ensure the foreground tasks' performance. After 32 seconds, the foreground tasks start to sleep one by one. Finally, all foreground tasks sleep, and background tasks occupy their original bandwidth.

Impact on latency. YCSB on RocksDB and two workloads from TailBench [36] (img-dnn [66] and masstree [44]) are selected as the latency-critical (LC) applications. First, we measure the 95th percentile latency of these LC tasks when running alone without any interference. Then we run the LC tasks together with 25 Graphchi (as the BE tasks) and measure their latency. We then group the LC tasks into one high-priority TGroup and the Graphchi into another (the BE TGroup) and repeat the same experiments. The results are shown in Table 5. Since YCSB's results are similar for all workloads, only the results for workload A are given.

Table 6: Tail latency of the LC task and throughput of the BEs

Workloads	alone	w/o MT^2	w/ MT^2
`95th YCSB-A-read (us)`	39	61	41
`95th YCSB-A-update (us)`	58	86	64
`99.9th YCSB-A-read (us)`	69	110	76
`99.9th YCSB-A-update (us)`	419	545	449
`FIO(DR) (GB/s)`	17.3	14.9	16.7
`FIO(DW) (GB/s)`	10.8	10.4	10.8
`FIO(NR) (GB/s)`	13.2	4.8	8.8
`FIO(NW) (GB/s)`	10.3	7.8	1.2

Without MT^2, the 95th percentile latency of `img-dnn` increases to 17.4x. MT^2 can restore all LC tasks' latency almost to the level when there is no interference at all. At the same time, the bandwidth of the BE tasks is limited to about 25% of the original. The performance of the BE tasks can be rapidly restored after the LC tasks are completed.

MT^2 in an untrusted environment. For this case, we use the hardware method (PEBS) to separate the two types of write bandwidth. We run YCSB as the LC application along with four fio tasks as the BE tasks. The four BE tasks perform read or write operations on NVM or DRAM respectively. MT^2 can optimize the latency of the LC application and improve the throughput of some BE applications by only restricting the bandwidth of the noisiest BE application as shown in Table 6. This thanks to the ability of distinguishing different types of memory bandwidth.

6.2 Performance Overhead

The performance overhead derives from three aspects: interference detection, bandwidth monitoring and restriction. The interference detection and monitoring overhead stems from setting and reading the PMU registers reading MBM data, and issuing write requests, while the restriction overhead comes from setting MT^2 context for threads, including MBA throttling value and CPU scheduling.

6.2.1 Trusted Environment

We measure the throughput or the execution time of some aforementioned test programs (`fio`, `graphchi`, `hadoop`, and `RocksDB`) with/without MT^2 to evaluate the performance overhead. As shown in Table 7, all overheads are less than 0.01%. The slight performance improvement in `graphchi` and `hadoop` is caused by noise. The overhead is negligible because most operations in MT^2 are performed by the dedicated kernel thread. For the interference detection, no overhead or additional bandwidth contention is introduced as only 400KB

Table 7: The performance overhead of MT^2

Thput/Time	w/o MT^2	w/ MT^2	Overhead
`fio`	31505 MB/s	31507 MB/s	< 0.01%
`graphchi`	321.64 s	321.55 s	< 0.01%
`hadoop`	54.93 s	54.93 s	< 0.01%
`RocksDB`	37770 ops/s	37767 ops/s	< 0.01%

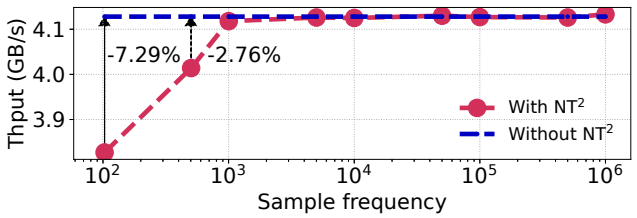


Figure 10: The throughput of `fio` under different sample frequencies. Frequencies no less than 10^3 introduce nearly no overhead. The Y-axis starts at 3.8GB/s to show the difference clearly.

data is written by the dedicated kernel thread for each period in our implementation, Others access the MSRs to use the hardware, which introduces little performance overhead.

We also run two applications on the same core to measure the overhead introduced in context switches. MT^2 increases an average of 900 cycles (less than 1 microsecond) in each context switch, which is insignificant compared to the millisecond-level scheduling period.

6.2.2 PEBS in the Untrusted Environment

In an untrusted environment, PEBS sampling is one of the main sources of performance overhead and is closely related to the sample frequency. We use `fio` to test the throughput under different sample frequencies and present the result in Figure 10. The overhead is negligible when the sample frequency is no less than 10^3.

6.3 Efficiency

We then split the regulation mechanisms to show the necessity of the two-stage algorithm design. First we run YCSB (as the victims) along with graphchi (as the noisy neighbors), and try using different techniques separately to restore the throughput of YCSB to 80% of the throughput it runs alone. We can not achieve the desired goal (restoring the throughput of YCSB to 80% of the initial) with MBA only. As shown in table 8, when we throttle the memory bandwidth only via CPU quota to restore the throughput of YCSB to 80%, the execution time of graphchi is 10m40s. The corresponding time is 9m56s when MT^2 is used. This indicates that MT^2 can control the bandwidth efficiently, consistent with our analysis in § 3.3.1.

Table 8: The efficiency of MT^2

Time	MT^2	CPU Scheduling
`graphchi`	9m56s	10m40s

Table 9: The deviation (in %) of monitored bandwidth in MT^2

Bandwidth(GB/s)	DR	DW	NR	NW
MT^2	10.51	4.19	3.84	2.79
PCM	10.69	4.22	3.89	2.81
Deviation	1.68%	0.71%	0.13%	0.71%

6.4 Accuracy

When the environment is trusted, applications faithfully report their NVM write bandwidth using the interface we provide. Although other bandwidths are obtained using reliable techniques, we still need to verify whether the results are accurate. PCM [58] is a software that can monitor different types of bandwidth of the whole system. When there is only one memory-intensive program in the system, the system-wide bandwidth reported by PCM is almost equal to the only program's bandwidth. So we run four `fio` simultaneously to generate different kinds of workloads and compare the bandwidths monitored by MT^2 with those of PCM since Intel also uses PCM's bandwidth as the baseline [32]. The results are shown in Table 9, where the results reported by MT^2 are very close to the ground-truth bandwidths.

7 Discussions

7.1 Limitations and Possible Mitigations

MT^2 brings the hybrid memory bandwidth regulation with several limitations. MT^2 relies on hardware mechanisms such as MBA, MBM, and PMU, which may conflict with applications that also depend on these techniques. The problem stems from the limited hardware resources. For example, there are only 79 RMIDs on our platform. It is possible to mitigate these limitations via virtualization or by more powerful hardware in the future.

Besides, the granularity of our empirical model is coarse, which may result in some bandwidth waste. Machine learning may be used to build more accurate models with a finer granularity in the future.

Currently, MT^2 is only able to accurately track the bandwidth of applications accessing the NVM via the file system and NVM programming libraries like PMDK. On trusted environments, MT^2 relies on applications to report their NVM write bandwidth honestly. Since many applications (such as PmemKV [5], Pmem-RocksDB [10], and Pangolin [71]) choose to use NVM programming libraries to manage the NVM, this can be easily achieved by slightly modifying the libraries. For untrusted environments, MT^2 can only monitor the write bandwidth roughly because of the hardware limitation, which can be addressed correctly by an update to the hardware mechanism.

7.2 NUMA

Currently, MT^2 does not support cross-NUMA bandwidth monitoring/regulation. MT^2 assumes that applications are bound to the same NUMA node where its NVM resides so

that it can monitor and regulate without cross-NUMA NVM accesses. The binding can be done manually (by the admins) or by the system's scheduler (e.g., in cloud environments). This assumption is reasonable and commonly stands in practice since applications and FS tend to access the local NUMA node to avoid degraded cross-NUMA accesses. For multi-socket machines, MT^2 can separate the monitoring and restriction policies for different sockets so that MT^2 will not regulate the TGroup in socket 1 when the bandwidth contention level is high in socket 0.

7.3 Future work

In some scenarios, cross-NUMA accesses are inevitable. A NUMA-and-NVM-aware scheduler can mitigate the memory bandwidth interference via more advanced scheduling policies, e.g., isolating DRAM-only applications and the NVM-intensive applications to different NUMA nodes. We leave the memory throttling in such scenarios as future work.

MT^2 prefers to limit TGroups with massive write NVM writes. However, these TGroups are not always the culprits of memory bandwidth contention. Accurately identifying the noisy neighbors remains a challenge and might require more hardware assistance. We leave the exploration of accurate bandwidth monitoring and regulation with hardware modifications as future work.

8 Related Work

DRAM bandwidth monitoring and regulation. MemGuard [67–69] is a DRAM bandwidth reservation system designed for real-time multi-core systems. It provides guaranteed and best-effort DRAM bandwidth for different applications. MemGuard monitors the DRAM traffic by accounting for the cache misses and suspends a task when it has exhausted its budgets in a given period.

Although both MemGuard and MT^2 aim to throttle memory bandwidth to avoid interference, MT^2 differs from MemGuard in several aspects. First, MemGuard is designed for DRAM in real-time systems, but MT^2 is proposed for hybrid NVM/-DRAM platforms. Second, MemGuard throttles DRAM bandwidth via a software budget-based throttling mechanism. MT^2 leverages both hardware and software mechanisms and proposes a dynamic bandwidth throttling algorithm to better regulate bandwidth for various applications.

LIKWID [59], Larysch [40], and Merlin [57] estimate memory bandwidth with L3 cache miss information collected from hardware performance counters. LibDistGen [17] estimates the memory bandwidth of applications based on stack reuse histograms. Mmbwmon [16] estimates the memory bandwidth consumption of applications by running benchmarks on other CPU cores of the system simultaneously. These techniques are proposed for DRAM and cannot be simply adopted to a system with both DRAM and NVM.

EMBA [61] models the relationship between performance and LLC occupancy and memory bandwidth and then proposes an algorithm with Intel MBA to restrict the memory bandwidth to improve the overall system performance in data centers. However, EMBA cannot control the memory bandwidth of a group of threads, and it cannot be used on hybrid NVM/DRAM platforms.

HyPart [50] consists of thread packing, clock modulation and Intel's MBA. MT^2 utilizes the CPU scheduler, thus providing finer-grained and precise control. Caladan [24] is a CPU scheduler that supports task monitoring and scheduling at the microsecond level. Some other studies [15, 23, 47, 72] also reduce resource contention with a modified scheduler.

None of these works can be used directly on hybrid memory NVM/DRAM platforms because the interference model is completely different from the DRAM-only platforms. They can only be used on hybrid platforms if they can separate the DRAM and NVM traffic at thread granularity as MT^2 does.

Hybrid NVM/DRAM bandwidth interference regulation. FairHym [27] will limit the frequencies of cores that perform NVM writes when the NVM write bandwidth exceeds a threshold to improve the inter-process fairness. It only concerns the bandwidth interference between NVM writes and DRAM accesses. It requires an impractical setup (installing DRAM and NVM on different NUMA nodes) to estimate the number of NVM writes. MT^2 has more flexible monitoring and allocation method that takes all types of bandwidth interference into account and can be used to meet the need of different user scenarios. Dicio [49] can control the bandwidth interference in a single LC and a single BE job situation. It blames and throttles the only BE job. In real-world scenarios, it cannot figure out which one to blame. In comparison, MT^2 targets a more practical setup (each NUMA node has both DRAM and NVM) and a more common scenario where multiple applications can run together.

9 Conclusions

This paper presents MT^2, the first comprehensive system to regulate memory bandwidth on the hybrid NVM/DRAM platforms. MT^2 first detects the bandwidth interference, monitors four types of memory bandwidth through various mechanisms and adjusts the bandwidth with a dynamic bandwidth throttling algorithm. Evaluation shows that MT^2 can effectively regulate the bandwidth among applications with nearly zero performance overhead and can be used in multiple use cases.

Acknowledgments

We sincerely thank our shepherd Sanidhya Kashyap and the anonymous reviewers from ATC '20, FAST '21, ATC '21, SoCC '21, and FAST '22 for constructive comments and insightful suggestions. This work is supported in part by the High-Tech Support Program from Shanghai Committee of Science and Technology (No. 19511121100), the National Natural Science Foundation of China (No. 61925206), and Huawei. Mingkai Dong (mingkaidong@sjtu.edu.cn) is the corresponding author.

References

[1] Apache hadoop. https://hadoop.apache.org/.

[2] Flexible i/o tester. https://fio.readthedocs.io/en/latest/index.html.

[3] Intel® optanetm dc persistent memory quick start guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf.

[4] Introduction to cache allocation technology in the intel® xeon® processor e5 v4 family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology.

[5] Key/value datastore for persistent memory. https://github.com/pmem/pmemkv.

[6] Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html.

[7] Memcached. https://memcached.org/.

[8] A persistent key-value store for fast storage environments. https://rocksdb.org.

[9] Persistent memory programming. https://pmem.io/pmdk/.

[10] Rocksdb on persistent memory. https://github.com/pmem/pmem-rocksdb.

[11] Twitter socail graph. http://snap.stanford.edu/data/ego-Twitter.html.

[12] Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.

[13] Google Cloud. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory, 2019.

[14] SAP HANA. https://www.sap.com/products/hana.html, 2019.

[15] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, page 1, USA, 2011. USENIX Association.

[16] Jens Breitbart, Simon Pickartz, Stefan Lankes, Josef Weidendorfer, and Antonello Monti. Dynamic co-scheduling driven by main memory bandwidth utilization. *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 400–409, 2017.

[17] Jens Breitbart, Josef Weidendorfer, and Carsten Trinitis. Automatic co-scheduling based on main memory bandwidth usage. In *JSSPP*, 2015.

[18] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[19] 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.

[20] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and protection in the zofs user-space nvm file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 478–493, New York, NY, USA, 2019. ACM.

[21] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 719–731, Berkeley, CA, USA, 2017. USENIX Association.

[22] Intel Xeon Processor Scalable Memory Family. Uncore performance monitoring reference manual. *Intel Corporation, July*, 2017.

[23] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, page 25–38, USA, 2007. IEEE Computer Society.

[24] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 281–297, 2020.

[25] Yiming Huai et al. Spin-transfer torque mram (stt-mram): Challenges and prospects. *AAPPS bulletin*, 18(6):33–40, 2008.

[26] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency

in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, page 187–200, USA, 2018. USENIX Association.

[27] S. Imamura and E. Yoshida. Fairhym: Improving interprocess fairness on hybrid memory systems. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, 2020.

[28] Intel. Intel optane dc persistent memory readies for widespread deployment. https://newsroom.intel.com/news/intel-optane-dc-persistent-memory-readies-widespread-deployment/, 2018.

[29] Intel. Intel 64 and ia-32 architectures software developer's manual. *Volume 3: System Programming Guide*, pages Vol. 3B 17–64, 2019.

[30] Intel. Intel 64 and ia-32 architectures software developer's manual. *Volume 3: System Programming Guide*, 2019.

[31] Intel. Intel 64 and ia-32 architectures software developer's manual. *Volume 3: System Programming Guide*, pages Vol. 3B 18–19, 2019.

[32] Intel. Intel resource director technology (intel rdt) on 2nd generation intel xeon scalable processors reference manual. *Intel Resource Director Technology Reference Manual*, 2019.

[33] Intel. Intel(R) Optane(TM) DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2019.

[34] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[35] 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.

[36] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[37] Takayuki Kawahara. Scalable spin-transfer torque ram technology for normally-off computing. *IEEE Des. Test*, 28(1):52–63, January 2011.

[38] 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.

[39] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX.

[40] Florian Larysch. Fine-grained estimation of memory bandwidth utilization. Master thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, March13 2016.

[41] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[42] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. Wort: Write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, page 257–270, USA, 2017. USENIX Association.

[43] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In Deborah T. Marr and David H. Albonesi, editors, *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pages 450–462. ACM, 2015.

[44] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[45] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[46] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and fast persistence for volatile

data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 789–806, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 153–166, New York, NY, USA, 2010. Association for Computing Machinery.

[48] Khang T Nguyen. Introduction to memory bandwidth monitoring in the intel(r) xeon(r) processor e5 v4 family. https://software.intel.com/en-us/articles/introduction-to-memory-bandwidth-monitoring, 2016.

[49] Jinyoung Oh and Youngjin Kwon. Persistent memory aware performance isolation with dicio. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '21, page 97–105, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[51] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206, 2020.

[52] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[53] Dulloor Subramanya Rao, Sanjay Kumar, Anil S. Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, pages 15:1–15:15, 2014.

[54] Yujie Ren, Changwoo Min, and Sudarsun Kannan. Crossfs: A cross-layered direct-access file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154. USENIX Association, November 2020.

[55] Smith Ryan. Intel announces optane storage brand for 3d xpoint products. https://www.anandtech.com/show/9541/intel-announces-optane-storage-brand-for-3d-xpoint-products, 2015.

[56] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *nature*, 453(7191):80, 2008.

[57] Priyanka Tembey, Ada Gavrilovska, and Karsten Schwan. Merlin: Application- and platform-aware resource allocation in consolidated server systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[58] Thomas Willhalm and Roman Dementiev. Intel(r) performance counter monitor - a better way to measure cpu utilization. https://software.intel.com/content/www/us/en/develop/articles/intel-performance-counter-monitor.html, 2012.

[59] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, page 207–216, USA, 2010. IEEE Computer Society.

[60] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[61] Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. Emba: Efficient memory bandwidth allocation to improve performance on intel commodity processor. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[62] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[63] 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 Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.

[64] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 167–181, USA, 2015. USENIX Association.

[65] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 98–105, 2020.

[66] Xingdi (Eric) Yuan. A deep network handwriting classifier. https://github.com/xingdi-eric-yuan/multi-layer-convnet, 2014.

[67] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In Robert Davis, editor, *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pages 299–308. IEEE Computer Society, 2012.

[68] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 55–64. IEEE Computer Society, 2013.

[69] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Trans. Computers*, 65(2):562–576, 2016.

[70] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 897–911, USA, 2019. USENIX Association.

[71] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 897–912, 2019.

[72] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, page 129–142, New York, NY, USA, 2010. Association for Computing Machinery.

[73] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 461–476, USA, 2018. USENIX Association.

Aurogon: Taming Aborts in All Phases for Distributed In-Memory Transactions

Tianyang Jiang, Guangyan Zhang*, Zhiyue Li, Weimin Zheng
Department of Computer Science and Technology, BNRist, Tsinghua University

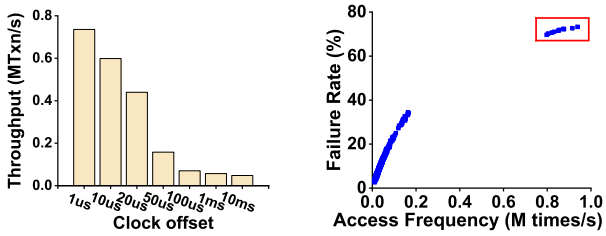
Abstract

Flourishing OLTP applications promote transaction systems to scale out to datacenter-level clusters. Benefiting from high scalability, timestamp ordering (T/O) approaches tend to win out from a number of concurrency control protocols. However, under workloads with skewed access patterns, transaction systems based on T/O approaches still suffer severe performance degradation due to frequent transaction aborts.

We present Aurogon, a distributed in-memory transaction system that pursues taming aborts in all execution phases of a T/O protocol. The key idea of Aurogon is to mitigate request reordering, the major cause of transaction aborts in T/O-based systems, in all phases: in the timestamp allocation phase, Aurogon uses a clock synchronization mechanism called 2LClock to provide accurate distributed clocks; in the request transfer phase, Aurogon adopts an adaptive request deferral mechanism to alleviate the impact of nonuniform data access latency; in the request execution phase, Aurogon pre-attaches certain requests to target data in order to prevent these requests from being issued late. Our evaluation shows that Aurogon increases throughput by up to 4.1× and cuts transaction abort rate by up to 73%, compared with three state-of-the-art distributed transaction systems.

1 Introduction

Many online transaction processing (OLTP) applications such as Web service and e-commerce scale out to massive servers with the growing computational and storage demands. Distributed transaction systems pursue high throughput and low latency to meet the requirements of OLTP applications. OLTP applications have two characteristics. First, OLTP workloads feature severe skew in data access frequency, making data hotspots common [4, 6, 18, 35, 39, 51, 56]. Second, requests in transactions usually contain data dependency [3, 11, 12] such as read-modify-write (RMW) operations, which are more complex to handle than overwrite operations due to the dependency of write on read. Multiple RMWs accessing data hotspots concurrently will bring about high contention, which triggers numerous transaction aborts. Aborts will decrease the throughput of distributed transaction systems and increase transaction latency due to retrying aborted transactions.



(a) (b)

Figure 1: (a) Throughput of distributed transaction system with varying clock accuracy. (b) Relationship between request failure rate and access frequency (hotspots are in red box).

Different concurrency control protocols are used in plenty of transaction systems. These protocols are mainly categorized into two-phase locking (2PL) [10, 49], optimistic concurrency control (OCC) [8, 13, 44], and timestamp ordering (T/O) [2, 26]. Under OLTP workloads with the aforementioned characteristics, 2PL systems suffer performance degradation due to low concurrency levels of locks on data hotspots, while OCC systems encounter frequent aborts since numerous RMWs interrupt the execution of read requests. Compared with them, T/O systems show better performance [19] since they can support flexible concurrency so long as multiple requests are served in the timestamp order.

Transaction execution in T/O systems undergoes four phases: allocating timestamps, transferring requests, executing requests and committing transactions. Existing T/O systems can be categorized into *clock-driven approach* [7, 15, 29, 40, 47, 55] and *data-driven approach* [53, 54], both of them still have deficiencies in addressing the problem of transaction aborts. The former only optimizes one of the first three phases. The latter seeks opportunities in the commit

*Corresponding author: gyzh@tsinghua.edu.cn

phase to save transactions that will be aborted, but aborts are often already inevitable at that time. To this end, this paper explores how to tame aborts in all phases of a T/O protocol when designing a distributed transaction system.

A transaction will abort once the execution of any request in this transaction fails. The key reason for request execution failures in T/O systems is that **requests are not executed in the order of their timestamps**. We call this phenomenon *request reordering*. We examine the aforementioned four phases of T/O systems and find that request reordering only occurs in the first three execution phases. Here we introduce three reasons for request reordering in different phases briefly.

1) Inaccurate distributed clocks in the timestamp allocation phase. The inaccurate timestamps allocated for transactions degrade the performance of T/O systems [33, 54]. To verify this, we adjust the clock offset among servers from 1µs to 10ms manually in a T/O system and see a throughput reduction of up to 94% (Figure 1(a)) under the YCSB workload [9].

2) Nonuniform data access latency in the request transfer phase. The nonuniformity of data access latency aggravates request reordering. The latency of accessing remote servers via the network (≈2µs) [22] is at least one order of magnitude larger than that of accessing local memory (≈100ns) [50]. This perhaps makes the order of request arrivals at the destination mismatch the order of request timestamps.

3) Requests with data dependency in the request execution phase. Requests that depend on the previous requests' results in the same transaction are called *dependent requests* in the rest of this paper. In the request execution phase, dependent requests will be issued late due to dependency wait, leading to an execution failure with high probability.

We present *Aurogon*[1], an **all-phase reordering-resistant** distributed in-memory transaction system ensuring serializability. To avoid request reordering, we devise three key techniques to address the three issues, respectively.

To design the clock synchronization for transaction systems, one key challenge is how to achieve high accuracy of distributed clocks under serious CPU interference from foreground transaction processing. With the growing capability of high-speed networks (*e.g.*, RDMA [13, 43]), CPU resources nowadays are becoming the bottleneck in distributed systems [17, 52]. Both foreground transaction processing and background clock synchronization share and even contend for CPU resources. We have observed that *CPU-NIC clock synchronization and NIC-NIC clock synchronization are heterogeneous*. Inspired by this observation, we propose a *two-layer clock synchronization mechanism* called *2LClock*. 2LClock provides a distributed clock with an average accuracy of 41ns under foreground interference.

To mitigate the impact of nonuniform data access latency, one challenge is how to trade off between transaction processing latency and abort rate. Performing requests in a first-come-first-serve (FCFS) order violates the original timestamp order of requests [7], leading to potential transaction aborts. Thus, we have an opportunity to buffer those requests with larger timestamps and defer their execution to tolerate late arrivals of requests. But request deferral will increase the latency of the deferred requests. We further examine the request failure rate of individual data records (Figure 1(b)) and find that transaction aborts mostly arise from failures of requests on data hotspots. In other words, deferrals of requests performed on cold data hardly reduce aborts. So, Aurogon adopts an *adaptive request deferral mechanism* that detects data hotspots dynamically and only defers those requests performed on hotspots.

Finally, to prevent dependent requests from being issued late, we *pre-attach dependent requests to data* when transactions including these requests access the target data for the first time. Specifically, within a transaction, the metadata of dependent requests will be transferred to target servers together with the requests they depend on, and then the metadata will be attached to the corresponding data. Thus dependency wait will not lead to the late arrivals at the destination of these dependent requests.

We implement Aurogon on an RDMA-capable cluster. We compare Aurogon with three state-of-the-art distributed transaction systems (i.e., Sundial-CC[2] [54], DrTM+H [48], and DST [47]) by evaluating them under two typical OLTP workloads with different degrees of contention. Our experiments show that Aurogon achieves up to 4.1× higher throughput and up to 86% lower average latency than prior systems. Our further experiments also show that Aurogon decreases the abort rate by up to 73%.

2 Background and Motivation

In this section, we first show the characteristics of workloads we target (§2.1). Then we profile prior T/O systems and point out their deficiencies (§2.2). A main disadvantage is that existing timestamp allocation schemes are inaccurate, so we need to introduce clock synchronization approaches to solve this issue. One common concern is why existing clock synchronization approaches cannot work well in transaction systems, which is analyzed in §2.3.

2.1 Working Scenario

We conduct detailed profiling on typical OLTP workloads and characterize two important features below.

Dependent requests. There are two kinds of dependent requests in transactions: 1) *key-dependent requests*, determining which key to read or write with the indexing information from previous read requests, 2) *value-dependent requests*, determining the value that they update the records with using the return

[1] Aurogon is a god who controls day and night in Chinese mythology.

[2] We denote the work [54] as Sundial-CC and the work [28] as Sundial-Clock in this paper for distinction.

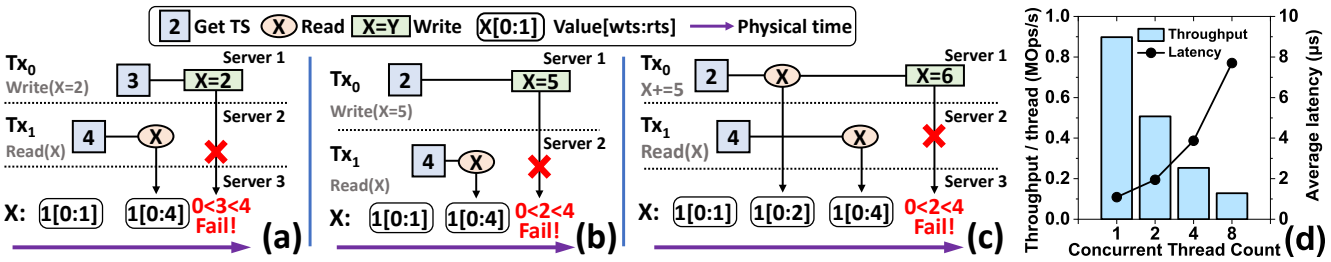


Figure 2: (a-c) Transaction aborts resulting from request reordering, which is caused by inaccurate distributed clocks (a), by nonuniform latency of data accesses (b), and by dependent requests (c), respectively. For brevity, we only plot one request for each transaction. (d) Per-thread throughput and latency when different numbers of threads query the clock of one NIC.

	smallbank	TPC-C	TPC-E
Ratio of transactions with RMWs	90%	92%	25%
Avg. RMW count per transaction	1.41	7.22	3.49

Table 1: Statistics of RMWs in typical OLTP workloads.

results of previous read requests. For instance, in "a[i]+=5", WRITE(a[i]) depends on the result of READ(i) so it is a key-dependent request, and in "a=b+5", WRITE(a) depends on the result of READ(b) so it is a value-dependent request.

With dependent requests, developers can implement complex business logic. For instance, RMWs, the most typical value-dependent requests, are widely used in OLTP applications (Table 1). However, it is complicated to execute dependent requests efficiently in a serializable transaction system. Taking a RMW as an example, any other write performed between the executions of the read and write in the RMW will make the whole RMW fail.

Skew in data access frequency. Data hotspots exist widely in real-world database workloads [4, 6, 18, 35, 39, 51, 56]. A small fraction of data is accessed frequently in a burst with the occurrence of hot events while other data remains cold. For instance, the analysis [5] shows that the top Twitter users had a disproportionate amount of influence indicated by a power-law distribution, so their tweets become hotspots.

2.2 Limitations of T/O Transaction Systems

There is a great deal of prior work [2, 7, 15, 29, 40, 47, 53–55] studying T/O transaction systems. The core idea of T/O systems is to take the partial order relation revealed by timestamps as the serializable order in transaction systems. All transactions in the system reach a consensus on the order that timestamps reveal. A transaction attaches its timestamp to all requests it issues, and requests are performed in the timestamp order on each data record. According to the sources of their timestamps, existing T/O systems lie in two categories:

1) Clock-driven approach. In these systems [7, 15, 29, 40, 47, 55], timestamps allocated for transactions are obtained from local clocks. **Those timestamp allocation approaches pursue scalability but relax the requirement for timestamp accuracy**. We study three typical systems [7, 29, 47] and conclude that they utilize a kind of loosely synchronized clocks, called *chasing clocks*.

Specifically, requests record their timestamps on data during execution and return the largest timestamp recorded on data. These returned timestamps help servers to update their lagging clocks if servers find their clock values are smaller than the returned timestamp. The process makes it seem that all clocks *chase* the fastest one in the system. Although all clocks will catch up with the fastest one eventually, clocks are not accurate instantaneously. Therefore, chasing clocks decrease the performance of transaction systems although they do not violate the correctness.

2) Data-driven approach. Transactions' timestamps are determined by their read or write dependency of committed transactions in these systems [53, 54]. A transaction first obtains the dependencies when accessing data and uses them to determine its timestamp in the commit phase. Data-driven approach seeks opportunities in the commit phase to save transactions that will be aborted by reordering their commit timestamps without violating the known dependency. However, a majority of aborts have been inevitable at that time due to request reordering during execution.

Implications. Existing T/O systems hardly alleviate aborts due to either inaccurate timestamps caused by chasing clocks or inevitable request reordering during execution. So our design adopts high-accuracy clock synchronization to allocate timestamps and prevent request reordering in each phase of execution.

Request reordering in each phase. T/O systems execute a transaction via four phases: 1) allocating a timestamp for each transaction, 2) transferring requests to the servers containing required data, 3) executing requests in their timestamp order and returning execution results, and 4) committing the transaction after receiving all acknowledgements successfully.

After analyzing request reordering phase by phase, we list three key reasons: inaccurate distributed clock, nonuniform latency of data accesses, and requests with data dependency. Figure 2(a-c) shows three cases of the reasons, respectively. There are two transactions (Tx_0, Tx_1), which issue read/write/RMW requests to one record (X). [wts, rts] repre-

sents the live period of a certain record version (more details in §4.1). For a version, a write will fail if its timestamp ts satisfies wts ≤ ts < rts, and a successful read may increase rts.

In Figure 2(a), inaccurate distributed clocks cause the order of two transaction timestamps to not be consistent with the physical time order in which they get timestamps. If it takes the same time for Tx_0 and Tx_1 to access X, Tx_0 starting later will abort due to its smaller timestamp.

Figure 2(b) illustrates that nonuniform latency of data accesses incurs request reordering though clocks are accurate. Tx_1 starting later accesses X earlier than Tx_0 since X and Tx_1's execution reside on the same server. So Tx_0 aborts due to the late arrival of its request. The case occurs frequently in a large heterogeneous cluster because the latency of accessing data from different servers is usually nonuniform due to different network hop counts and uneven traffic distribution.

Figure 2(c) shows that dependent requests also lead to reordering. In this case, "X=6" in Tx_0 is such a request since it depends on the result of the previous read. The event "X=6" arrives at X later than Tx_1's read results in Tx_0's abort because "X=6" must wait for the return of Tx_0's read to calculate the value to be written.

2.3 Clock Synchronization Approaches

Clock synchronization dedicated to transaction systems should meet three requirements: 1) high accuracy, 2) low calling overhead, and 3) resistant to CPU interference. To achieve the three requirements, clock synchronization should adopt differentiated methods on CPU-NIC synchronization (CPU-NIC sync) and NIC-NIC synchronization (NIC-NIC sync). Prior work [16, 28, 32, 40] ignore the heterogeneity between CPU-NIC sync and NIC-NIC sync, so they cannot be applied to transaction systems directly.

Observation. *CPU-NIC sync probes show a larger latency fluctuation than NIC-NIC sync ones in transaction systems.*

The large fluctuation of probe latency will impair the accuracy of synchronization [16]. A longer execution time of a probe suggests the readings of two probed clocks can differ, leading to inaccurate clock synchronization. Prior work [16, 28] believes network traffic is the main reason for inaccurate probes so they focus on eliminating the effect of link noise. However, that does not cover all cases especially when foreground applications are CPU-intensive instead of network-intensive. Table 2 illustrates the latencies of both CPU-NIC sync and NIC-NIC sync probes under DrTM+H [48], a CPU-intensive distributed transaction system which saturates RDMA networks to accelerate transaction processing. The P999/median ratio of network probe latency only rises by 37% when adding the DrTM+H load. It suggests that peak network traffic from this advanced distributed transaction system does not affect the stability of network probes seriously. On the other hand, this ratio reaches 6.75 for CPU-NIC sync probes. This happens because getting CPU timestamps suffers from latency spikes when the CPU is preempted by foreground transaction processing.

	CPU-NIC sync		NIC-NIC sync	
	w/o load	w/ load	w/o load	w/ load
median (µs)	1.26	1.86	0.95	1.05
P999 (µs)	1.33	12.56	0.97	1.47
P999/median ratio	1.06	6.75	1.02	1.40

Table 2: Execution time of two kinds of probes without load or under the DrTM+H load.

Based on the heterogeneity we observed, existing clock synchronization approaches have four limitations. First, they consume precious CPU resources in transaction systems when using a dedicated core [40] to poll requests via user-level network interfaces, decreasing transaction throughput. Second, without separating CPU-NIC sync and NIC-NIC sync [32, 40], high accuracy cannot be achieved due to highly variable software stack latencies [16]. Third, the synchronized NIC clocks (*e.g.*, PTP [21], HUYGENS [16]) cannot serve intensive queries from transaction systems. Figure 2(d) illustrates that as the number of threads querying the clock of one NIC increases, the average query latency rises sharply and the per-thread throughput also decreases. Finally, some of them [27, 28] rely on customized modification to NICs, making their extensive deployment in datacenters difficult.

3 System Overview

We propose Aurogon, an all-phase reordering-resistant distributed in-memory transaction system, with solutions targeting the three issues discussed in §2.2.

Design rationale. The key idea of alleviating request reordering is to maintain the timestamp order for requests in all phases of transaction execution.

First, in the timestamp allocation phase, we propose a clock synchronization mechanism to improve the accuracy of timestamps obtained by transactions. The clock exploits a two-layer architecture, divided into CPU-NIC sync and NIC-NIC sync. The high accuracy achieved by 2LClock resists the CPU interference from foreground transaction processing.

Second, in the request transfer phase, Aurogon buffers received requests and defers the execution to mitigate the impact of nonuniform data access latency. To cut the latency rise caused by request deferrals, Aurogon only defers the requests performed on hotspots and shortens the request deferral time heuristically.

Third, in the request execution phase, to prevent dependent requests from being issued late, Aurogon pre-attaches the metadata of dependent requests to data when their transactions issue requests to the same data at the first time. This mechanism not only diminishes execution failures of dependent requests, but also saves one network communication for dependent requests.

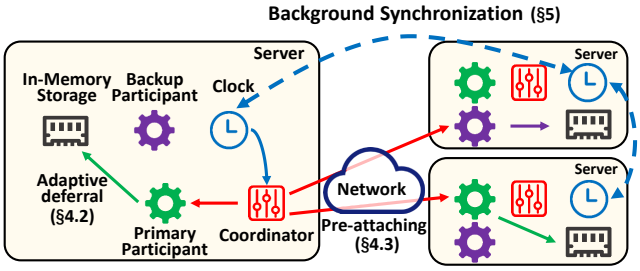


Figure 3: Aurogon architecture.

Aurogon Architecture. Figure 3 illustrates the Aurogon architecture. Servers are connected with a fast network. Data is partitioned into multiple shards and each server maintains one of data shards in its memory. All servers are homogeneous and transactions can start to execute in any server. Each server has four modules: a coordinator, a primary participant, a backup participant and a clock.

- The coordinator is responsible for coordinating transactions. It sends read/write/commit requests to primary participants and it decides to commit or abort transactions based on the replies. The coordinator also *pre-attaches the metadata of dependent requests to data* (§4.3).
- The primary participant processes requests in their timestamp order by accessing local in-memory storage, and finally sends their execution results to coordinators. Meanwhile, the primary participant conducts the method of *adaptive request deferral* to cut aborts.
- The backup participant replicates the commit messages from a coordinator before transactions are committed to the primary participant to tolerate a server failure.
- Clocks provide accurate timestamps for transactions and are synchronized via *2LClock* (§5).

4 Concurrency Control Protocol Design

4.1 Protocol Specifications

We first introduce the data management in Aurogon. Aurogon leverages a multi-version mechanism to store historical versions of data records and distinguishes data versions with timestamps obtained from 2LClock. Each data version has an active range of timestamps bounded by the write timestamp (*wts*) and the read timestamp (*rts*). Specifically, *wts* is the timestamp of the transaction that has created this version and *rts* indicates the maximum timestamp of transactions that read this version. The versions of a record are organized by a linked list sorted by their *wts* in descending order. Each version also contains a *status* that indicates whether this version is committed.

We now describe the rules for handling requests in Aurogon. Read requests always succeed while write requests may fail during the execution if they conflict with the committed reads. Reads are regarded as `COMMITTED` upon finishing execution so they do not demand an extra commit message, but writes require an extra communication with participants to ensure that all writes of the transaction are performed successfully.

Algorithm 1 shows the protocol of transaction processing in Aurogon. The coordinator performs a transaction T beginning with getting a timestamp (line 3) from local 2LClock (§5). The timestamp indicates the serial order of the transaction and is attached to all requests issued by T. Aurogon encodes the acquired timestamp together with the server and thread IDs of the coordinator into a 64-bit timestamp to make it globally unique.

In the request transfer phase, the coordinator traverses T's read and write sets dynamically, and sends its read requests to the participants[3] maintaining the required data (line 5-7). If a write depends on the return of the read accessing the same data record (*e.g.*, the write in a RMW), the read will piggyback the metadata of these *dependent writes*. Meanwhile, *independent writes*, which do not contain data dependencies, are issued to corresponding participants directly (line 8-9). It should be noted that transactions in Aurogon do not require knowing read and write sets before their executions. Specifically, transactions can add a key-dependent request into the read or write set while running, and then the coordinator will issue such a key-dependent request.

The coordinator receives return messages of T's requests from participants and aborts T once it obtains the failure return of any request (line 10-13). If T is aborted, the coordinator will notify the participants which have performed T's requests successfully to rollback the changes brought by T's requests. When T's all requests are returned successfully, the coordinator will start the commit phase, issuing commit messages to the participants containing the records in T's write set (line 17-20). Finally T can be committed to users.

`Tx_read` shows the execution of read requests in participants. If a read request R is issued together with a dependent write W_d, the participant will first install W_d into the data list (line 24-26). The installation process is illustrated in `Tx_write` later. Then the participant determines whether the execution of R will be deferred according to the hotness of data R accesses (line 27-28).

After a possible deferral, R starts to read by searching for the correct version based on its timestamp: it traverses the data list in the descending timestamp order (line 29) and chooses the first version v which satisfies $v.wts < R.ts$. In other words, v has the largest *wts* among versions whose *wts* are smaller than $R.ts$ (line 31). The participant will extend $v.rts$ to $R.ts$ if $v.rts < R.ts$ to show the existence of R (line 32). If $v.status$ is `PENDING`, which indicates that v is not yet committed, R will wait until $v.status$ turns to `COMMITTED` (line 33-34). Aurogon chooses to block R instead of reading uncommitted data [26] to prevent the high overhead caused by cascading aborts. Finally, R returns the correct data version to the coordinator.

[3] Participants refer to primary participants in the rest of this paper unless explicitly stated otherwise.

Algorithm 1: Pseudo code of Aurogon's concurrency control protocol.

```
1  Function Coordinate(txn)
2      // timestamp allocation phase
3      ts = get_local_clock_ts()
4      // request transfer phase
5      for read,record in txn do
6          send(read.dest_node, Tx_read, read, record)
7          //piggyback the metadata of dependent writes
8      for independent_write,record in txn do
9          send(write.dest_node, Tx_write, write, record)
10     while receiving an ACK do
11         if ACK.status == failure then
12             abort txn
13             return
14         if key-dependent requests exist then
15             update read and write sets and issue requests
16         // commit phase
17         if all ACKs are received then
18             replicate updates to backup participants
19             for write,record in txn do
20                 send(write.dest_node, Tx_commit, write, record)
21             commit txn // notify upper users
22 // request execution phase
23 Function Tx_read(record, req)
24     if req.dependent_write exists then
25         if install(record, req.dependent_write) fails then
26             return dependent_write failure // reply to coordinator
27     if record.is_hot == True then
28         wait for deferred_interval
29     for version in record.list do
30         // in descending order of version timestamp
31         if version.wts < req.ts then
32             version.rts = max(version.rts, req.ts)
33             if version.status == PENDING then
34                 wait until version.status == COMMITTED
35             reply version's data to coordinator
36             return success
37 Function Tx_write(record, req)
38     if install(record, req) fails then
39         return failure
40     else
41         return success
42 Function Tx_commit(record, req)
43     find req's version in record.list
44     version.status = COMMITTED
```

`Tx_write` shows the execution of write requests in participants. Before a write request W is processed to update the record r, the participant needs to ensure that no version v of r satisfies $v.wts \leq W.ts < v.rts$. If such v exists, W will conflict with requests which read v and each have a timestamp larger than $W.ts$. It is because these reads should return the updates of W but they have already returned v. W has to fail if such a conflict occurs as it is expensive or even impossible to change these reads' return and abort relevant transactions.

The participant validates W and searches for the correct installation position in this way: it traverses the version list and chooses the first version v_f which satisfies $v_f.wts \leq W.ts$, followed by validating whether $v_f.rts \leq W.ts$. The conflict would only occur on v_f since all versions are sorted in the descending order of wts. If W passes the validation, the participant installs a new version v' in front of v_f, setting the status of v' to `PENDING` and $v'.wts = v'.rts = W.ts$. Now the installation of W succeeds. The same process is also used to install a dependent write in `Tx_read`. Finally the participant returns the success of W to the coordinator.

`Tx_commit` shows the commit process of writes in participants. If the coordinator commits the transaction T, the participant finds the versions of all pending writes issued from T and turns the status of these writes to `COMMITTED`. Then the participant returns the pending reads blocked by these writes.

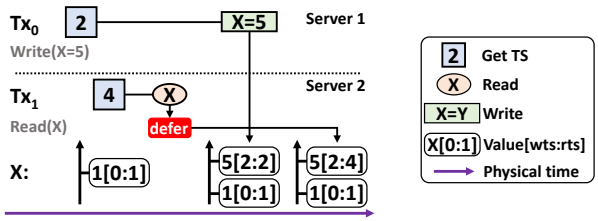


Figure 4: An example of request deferral.

4.2 Adaptive Request Deferral

To decrease the number of transaction aborts, we focus on reducing the failures of write requests since reads never fail in Aurogon. As depicted in Figure 2(a) and 2(b), writes will fail if they conflict with committed reads. If a read R is committed and returns the version v to the coordinator, an incoming write W with $v.wts \leq W.ts < R.ts$ has to fail in order to ensure serializability. Besides inaccurate distributed clocks, nonuniform data access latency causes that W with a smaller timestamp arrives later than R. The issue results from both the data location and the network queuing on transmitting links.

To solve this problem, we propose an adaptive request deferral mechanism to defer the execution of reads until the straggling writes arrive and finish. Figure 4 shows such a deferral case. Compared to Figure 2(b), Aurogon buffers Tx_1's read and defers its execution. "X=5" in Tx_0 benefits from the deferral and can be installed successfully when it arrives at X. Then the deferral ends, and Tx_1's read is performed, returning the latest value 5.

However, a raised challenge is that a read's deferral will increase the transaction latency inevitably. We tackle the difficulty by two methods: reducing the number of unnecessary deferred reads and shortening the request deferral time.

Cutting the deferred read count. We examine the failure rate of individual data records and observe that transaction aborts usually arise from request reordering on hotspots. The hotspots account for a small portion of user data but encounter enormous request failures. Figure 1(b) illustrates that the

failure rate rises sharply when data is hot. So the request deferral on hotspots can be much more effective.

To this end, we propose a lightweight hotspot detection scheme. The scheme counts the number of receiving requests for each record individually in last 10 milliseconds using accurate timestamps allocated for requests. If the throughput of a particular record exceeds a preset threshold, we consider that the record is being accessed frequently and at great risk of request failures. Then the request deferral mechanism starts to defer the execution of reads on high-contention records. The space overhead of the detection scheme is quite modest (only an 8-byte counter for each data record). Moreover, the detection scheme can discover the change of hotspots easily so that we will conduct deferrals on the new hotspots.

Shortening request deferral time. Aurogon exploits a heuristic method to increase the deferral time instead of setting an overlong deferral time in advance. The deferral time of each record is determined individually. Aurogon calculates the request failure rate of each record dynamically based on the hotspot detection scheme. The deferral time will be increased heuristically to tolerate late arrivals of more writes if the failure rate exceeds a preset threshold. If the request failure rate of the hotspot remains low, Aurogon could cut the deferral time conservatively to avoid the waste of deferral time. Our evaluation shows that a 20-microsecond deferral for reads on hotspots cuts 50% write failures in most cases.

Note that the request deferral mechanism is a best-effort method which cannot eliminate the write failures completely. However, we could trade a moderate latency rise of a handful of transactions for reducing aborts.

4.3 Pre-attaching Dependent Requests to Data

A dependent request may fail during execution since it has to wait for the results of requests it depends on within the same transaction (Figure 2(c)). Dependent requests are classified into key-dependent requests and value-dependent requests (§2.1) and we first target value-dependent requests.

Value-dependent requests are writes depending on the previous reads' results and the write set of value-dependent requests are deterministic. We first consider the situation that writes are issued to the same records with previous reads. such as an auto-increment counter. Taking RMW as an example, Figure 5 illustrates that three network communications are consumed in T/O systems [2, 19] without pre-attaching between the coordinator and the participant: 1) a read R is issued to the record X and obtains the correct data, 2) a preparing write request "X=8", PW, is sent to the record carrying the updated data, and 3) a commit or abort message C notifies the participant to commit or rollback the updates from PW. It leaves a *vulnerable interval* from the time point R finishes to the time point PW arrives, in which any arriving write W with $W.ts > PW.ts$ will break the integrity of RMW's execution and

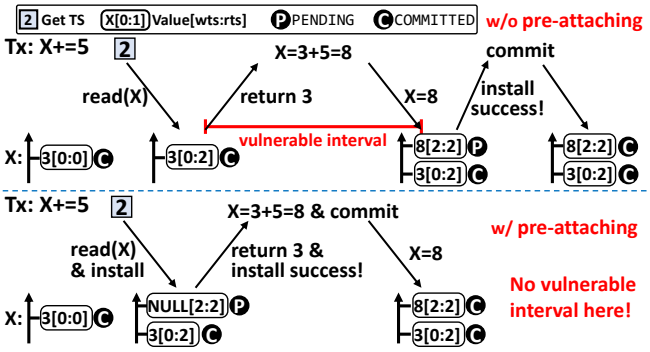


Figure 5: Dependent request handling with or without pre-attaching.

cause PW's failure.

To this end, Aurogon piggybacks the metadata of dependent requests to target data while transferring requests they depend on, and pre-attaches these dependent requests to data records for the purpose of eliminating the vulnerable interval (Figure 5). Specifically, the metadata of PW is issued to the participant together with R in the first communication, and the participant first tries to install PW to the data list before performing R (line 25 in Algorithm 1). If the installation fails, there is no need to perform R and the participant can return a failure message to the coordinator, resulting in an early abort of the transaction which saves CPU resources (line 26).

After installing the version of PW, the participant sets the version's status to `PENDING` and leaves its value NULL since the coordinator has not obtained R's return and could not have calculated PW's new value. Note that leaving the version's value NULL does not violate the correctness as the `PENDING` versions will block incoming reads. The participant performs R and returns the result immediately after installing PW, which eliminates the vulnerable interval. In the commit phase, the participants will receive the message C and commit the version of PW with new data if it is determined the transaction will commit.

Besides eliminating the vulnerable interval, pre-attaching dependent requests brings two extra advantages. First, it saves one network communication compared to prior T/O systems. Second, it assists the transaction in finding early aborts if the installation fails in the first network communication, which cuts unnecessary network bandwidth usage compared to encountering PW's failure after sending PW's data to the participant in Figure 5. Meanwhile, since the metadata piggybacked is just a boolean variable to show the existence of a dependent write, the latency of transferring requests are not affected.

We now discuss other situations of dependent requests. For other value-dependent requests (*e.g.*, PW accesses the records different from R), Aurogon supports issuing R followed by transferring the metadata of PW to its corresponding record for pre-attaching, which shortens the vulnerable interval greatly as well. For key-dependent requests, it is inevitable to wait for previous reads' results to determine which data

to access. Aurogon introduces the modification of adaptive request deferral mechanism to benefit these requests. Previous reads' executions are accelerated by disabling their possible deferral in participants selectively. Furthermore, for performing key-dependent requests, a key-dependent read can be performed directly since its dependency wait in the coordinator is regarded as an implicit deferral, while a key-dependent write may benefit from read deferrals on hotspots.

4.4 Aurogon's Isolation Level

Aurogon ensures serializability and here we only give a simple proof sketch. Each transaction has a globally unique timestamp. Requests are executed in the order of their transactions' timestamps. So each transaction can achieve a view of the system and update the system's status consistently with its globally unique timestamp. Therefore, transactions in Aurogon are serializable, and the timestamp order is the serial order of transactions.

Aurogon also ensures strong partition serializability (SPS) [1, 7, 41] as an extension. SPS is an isolation level slightly weaker than strict serializability [20, 38]. In a SPS system, for any two transactions T_1 and T_2, T_1 must precede T_2 in the serial order if two conditions are satisfied: 1) T_2 starts after T_1 finishes, 2) T_1 and T_2 access the same record. One can achieve strict serializability easily in SPS systems by adding explicit out-of-band dependencies with a cross-record read, solving the anomaly usually called "causal reverse".

Two main anomalies which can happen in serializable Aurogon are "stale reads" and "immortal writes" [1] compared to SPS. The key reason for both anomalies is that timestamps 2LClock allocates cannot match the physical time completely in a distributed system. A transaction could obtain a smaller timestamp so it cannot see the newest updates.

We solve the two anomalies by adding some modifications to request processing in participants. To prevent stale reads, reads cannot return a version v if there exists a `COMMITTED` version $v^{'}$ that $v^{'}.wts > v.wts$. To prevent immortal writes, a write W cannot be inserted if there exists a `COMMITTED` version v that $v.wts > W.ts$. Besides, coordinators should not commit a transaction T to users until receiving all of its commit acknowledgement messages. An early commit of T may cause that transactions starting later will find the status of T's updates is `PENDING`, which should have been `COMMITTED`, leading to the incorrect execution of two aforementioned modifications.

4.5 Fault Tolerance

To tolerate the server-level crash, Aurogon leverages the widely-used primary-backup replication similar to prior work [14,24,48]. Aurogon does not require to replicate coordinators since their commit decisions and states can be recovered from the states of primary participants and backup participants. If a transaction T is determined to be committed, its coordinator first replicates T's updates to backup participants. Backup participants perform transactions' updates asynchronously to survive primary participant failures. After receiving all ACKs from backup participants, T's updates can be committed to primary participants and T can be returned to users simultaneously.

Failures of coordinators. The approach for handling the coordinator failure is to finish transactions having been committed to users and abort running ones it coordinates. If primary participants detect the failure of a coordinator with periodic heartbeats, they need to judge whether the transactions coordinated by this coordinator has been committed to users. They first check if updates of the transaction exist in corresponding backup participants. If all backup participants retain the updates, it means the transaction is allowed to return to users so that primary participants will search for the transaction's updates in memory and commit them. Otherwise, primary participants will discard these `PENDING` updates and inform backup participants to roll back the possible changes.

Failures of participants. The updates of a transaction will first be replicated to backup participants followed by primary participants, so coordinators can select a new primary participant from backups with a lightweight consensus [25, 37] after detecting a primary participant failure. The correctness is guaranteed since backup participants have retained all transactions' updates that corresponding primary participants received. The failure of a backup participant can also be recovered by the consensus protocol.

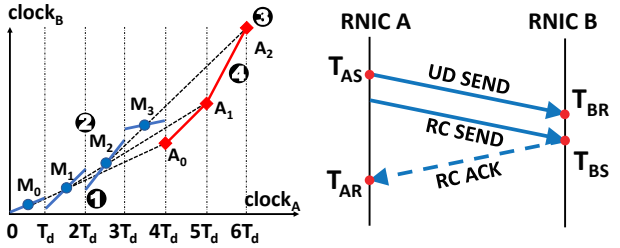
5 2LClock Design and Implementation

Inspired by the observation in §2.3, we propose *2LClock*, a clock synchronization mechanism using a two-layer mapping scheme to provide a global clock for each server. We further implement 2LClock with the help of an emerging network technique, RDMA [23, 31, 34, 43, 46]. 2LClock meets three requirements of clock synchronization proposed in §2.3.

- To achieve high accuracy, 2LClock separates CPU-NIC sync and NIC-NIC sync to alleviate the latency fluctuation from the software stack, and issues synchronization probes frequently to resist clock drift.
- To reduce the overhead of querying timestamps for transactions, 2LClock implements CPU-NIC sync to enable transactions to avoid querying NIC clocks directly.
- To resist the CPU interference from transaction processing, 2LClock filters inaccurate synchronization probes and cuts the CPU usage of synchronization with the help of RDMA.

5.1 Clock Mapping Functions

In 2LClock, all CPU clocks in the cluster synchronize with one preset "reference" NIC clock. Specifically, for a server, its CPU clock is synchronized with its local NIC clock, using the mapping function F_1. Then the local NIC clock uses the mapping function F_2 to synchronize with the reference NIC



(a) Clock mapping function (b) Probing of NIC-NIC sync

Figure 6: Mapping and probing in clock synchronization.

clock. The principle of constructing a mapping function F_i (F_1 or F_2) is to ensure their increasing monotonicity since distributed transaction systems cannot tolerate individual clocks going backwards. We now present how to construct F_i.

The synchronization accuracy of F_i will be influenced by the fluctuation of individual probes if we update the F_i upon receiving a new probe. So 2LClock divides the time into multiple successive timeslices with a fixed-length T_d and filters out "bad" probes collected in each timeslice to achieve high accuracy. As shown in Figure 6(a), 2LClock derives F_i in four steps. First, 2LClock generates a linear function for each timeslice (❶), which is different between F_1 and F_2 and is elaborated in §5.2. Second, for each timeslice $[i \cdot T_d, (i+1) \cdot T_d)$, 2LClock gets the middle point M_i of the line segment (❷). The third step is to get the set of *anticipated points* (❸). By drawing a line across two middle points M_i and M_{i+1}, 2LClock gets an anticipated point A_i. Finally, by connecting each two successive anticipated points (❹), 2LClock gets a mapping clock function (red lines with diamond symbols in Figure 6(a)).

One may wonder why 2LClock exploits an extension method to obtain a piecewise mapping function. It is because the linear function generated in the first step may not be successive at the junction of two timeslices, which may violate the monotonicity of allocated timestamps.

5.2 Synchronization in a Timeslice

As depicted in the first step in §5.1, to construct either F_1 or F_2, 2LClock first needs to generate a linear function for a timeslice. Here we introduce the ways to generate such linear functions for F_1 and F_2 via synchronization.

5.2.1 CPU-NIC Synchronization

In a timeslice, 2LClock first produces a set of mapping pairs between a server's CPU clock and its local NIC clock by probing and measurement. Then, it fits those mapping pairs with linear regression, generating a linear function for F_1.

To produce a mapping pair dataset, a CPU-NIC sync thread periodically queries its local NIC to get *mapping pairs* $< t_c, t_n >$, which gives an estimation that the time in local NIC is t_n when time in CPU is t_c. Each NIC query generates three timestamps: a NIC timestamp (T_n) from the query reply, and two CPU timestamps (T_{c1} and T_{c2}) that the host CPU records when CPU begins and finishes the NIC query request. So we get the mapping pair $< t_c, T_n >$, where $t_c = \alpha \cdot T_{c1} + (1-\alpha) \cdot T_{c2}$. α is a parameter in $[0,1]$ and we set it offline[4].

After collecting enough mapping pairs within one timeslice, we discard those outlier pairs whose execution duration, $T_{c2} - T_{c1}$, deviates far from the normal value. Resource contention between clock synchronization and transaction processing can occasionally make the execution duration of some mapping pairs to millisecond level. So filtering out outlier pairs can effectively enhance the accuracy of CPU-NIC mapping.

5.2.2 NIC-NIC Synchronization

For NIC-NIC sync, 2LClock uses three steps to generate a linear function in each timeslice for F_2. First, 2LClock produces a set of mapping pairs between local NIC and the reference NIC. The reference NIC is on a randomly picked server in the cluster. 2LClock first gets a quadruple $<T_{AS}, T_{BR}, T_{BS}, T_{AR}>$ by issuing probes from local NIC to the target NIC, and then obtains a mapping pair $<t_n, t_g>$ from such a quadruple. Here we adopt the link symmetric assumption, widely used in prior work [16, 27, 28, 32]. Second, 2LClock uses supported vector machine to filter the mapping pairs deduced from those outlier probes as prior work [16] does. This is because a probe with a finishing time beyond the normal range of measurement encounters network fluctuation with high probability [16], which will violate the link symmetric assumption. Finally, 2LClock generates a linear function for F_2 by fitting mapping pairs with linear regression.

Tree structure of NIC-NIC sync. 2LClock builds a K-ary tree to organize the synchronization links, whose root is the reference NIC. Each NIC issues probes to its parent NIC, so that each NIC can ultimately get a mapping pair synchronized with the reference NIC clock. Each tier of the tree will introduce an additional synchronization error [28]. Thus we set K to 10 intuitively to limit the synchronization error. For example, only a four-tier tree is required to enable 2LClock to work in a 1000-server cluster ($1+10+10^2+10^3=1111$).

RDMA characteristics. To reduce CPU usage, 2LClock exploits two characteristics of RDMA when obtaining the clock mapping pair between a local NIC and its parent NIC with probes. First, RDMA *ibv_cq_ex* can automatically record accurate times when a network request leaves and arrives at an RDMA NIC (RNIC). This interface allows buffering these NIC timestamps in both sides' host memory via direct memory access without CPU involvement. Second, RDMA has multiple transport modes [24, 43] including Reliable Connected (RC) and Unreliable Datagram (UD). Specifically, RC mode uses acknowledgment packets to ensure reliable transmission, while UD mode removes these packets. We have

[4] Theoretically, α is the ratio between NIC query uplink latency and the sum of uplink and downlink latencies. In a homogeneous cluster, α's values on different servers are the same so the relative time of CPU clocks is not affected by α's values.

found an interesting feature of *ibv_cq_ex* in RC mode and used it to probe target RNIC: *ibv_cq_ex* records the leaving and arriving times of RC request's ACK instead of RC request itself. Next we show how to generate a probe and calculate a mapping pair.

Using a combination of RDMA modes. Figure 6(b) plots how a probe gets a quadruple when local RNIC A synchronizes with its parent RNIC B. 2LClock first sends a UD request from A to B with *ibv_cq_ex* and records two timestamps, T_{AS} and T_{BR}, indicating when the UD request leaves A and arrives at B, respectively. Then 2LClock sends an RC request from A to B with *ibv_cq_ex* as well and two timestamps, T_{BS} and T_{AR}, are recorded, indicating when the RC's ACK leaves B and arrives at A. If UD requests suffer packet loss, we discard the probes directly though it seldom happens [24].

Transferring timestamps asynchronously. Finally, the server that B resides on obtains existing timestamps T_{BS} and T_{BR} in a batch by polling local *ibv_cq_ex* [30, 36] periodically, and transfers them to A's server. The clock offset between B and A can be calculated as: $\texttt{offset}_{B-A} = ((T_{BR} - T_{AS}) - (T_{AR} - T_{BS}))/2$, and the mapping pair is $(T_{AS}, T_{AS} + \texttt{offset}_{B-A})$. We also measure the difference of one-way delay between RC and UD requests offline, and then subtract this part when calculating the offset to avoid violating the link symmetric assumption.

The design of probing in 2LClock reduces the CPU utilization of the parent nodes in the tree since we connect a parent node with K (10 by default) child nodes to alleviate additional errors. Specifically, it brings two advantages. First, using RDMA *ibv_cq_ex* to obtain accurate timestamps lowers CPU utilization compared with polling network interfaces frequently with a dedicated core [40]. Second, the combination of RC and UD requests offloads tasks of issuing probes from parent servers to child servers.

5.3 Fault Tolerance

2LClock uses a two-phase mechanism to tolerate a server failure. If a server detects a failure of its parent server with the heartbeat mechanism, it will turn to a new parent to synchronize with it. The selection of the new parent cannot violate the requirements in the child count and the height of the K-ary tree. Such new parent and child servers will perform a two-phase recovery since it will take a while for them to build new connections and warm up their clock mapping functions.

Figure 7 depicts how a child server A safely switches from parent server B to a new parent server C, without breaking its increasing monotonicity guarantee. For brevity, here we consider F_1 and F_2 as a whole, and denote their composite mapping function as F. When A detects that B fails at t_1, A will immediately connect to a new parent C and probe it to construct a new mapping function F_C synchronized with the clock in C (green line in Figure 7).

Given that 2LClock finishes warming up the new mapping

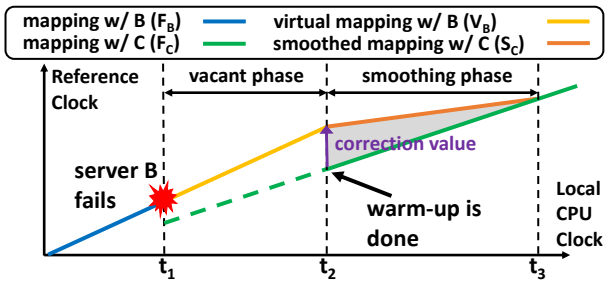


Figure 7: Server failure handling in 2LClock.

F_C at t_2, a new problem is that no mapping is available in A during the *vacant phase* from t_1 to t_2. So 2LClock provides a temporary clock by extrapolating the old mapping function synchronized with failed server B during the vacant phase. We call this mapping as virtual mapping V_B, as it may be inaccurate after a while.

To prevent the time from going backwards, a correction value is added when switching from V_B to F_C. However, different servers add different correction values, leading to inaccuracy of 2LClock. To eliminate the impact of correction value, we further add a smoothing phase from t_2 to t_3 (corresponding to the mapping S_C), during which the correction value decreases gradually till it becomes zero. After t_3, 2LClock comes back to the normal state again.

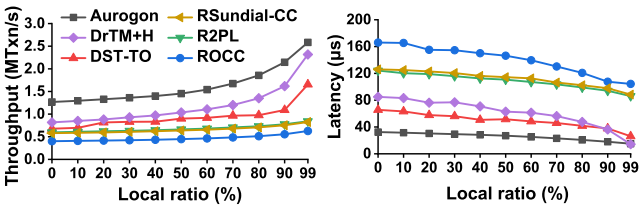
6 Performance Evaluation

6.1 Experiment Setup

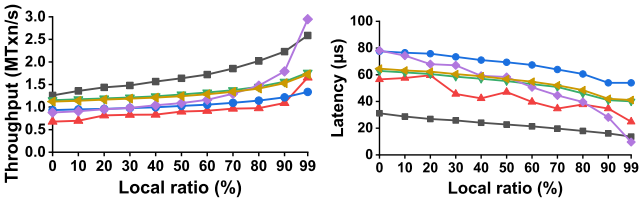
Transaction system setup. We compare Aurogon with five distributed transaction systems. First, we pick the state-of-the-art T/O systems from *clock-driven approach* and *data-driven approach*, respectively. For clock-driven approach, we integrate DST [47] into our system and build a compared system called *DST-TO*. In DST-TO, we replace 2LClock with DST and turn off Aurogon's reordering-resistant techniques. For data-driven approach, we choose Sundial-CC and replace the original TCP network with RDMA for a fair comparison based on an open-sourced implementation [45], called *RSundial-CC*. Second, we compare with *DrTM+H* [48], which saturates RDMA networks to accelerate distributed transactions. Third, we compare with systems based on traditional concurrency control protocols. We use the implementation [45] to evaluate an RDMA version of 2PL and OCC, called *R2PL* and *ROCC*, respectively.

Workloads. We use two workloads, TPC-C and YCSB.

TPC-C [11] is the industry standard benchmark for evaluating OLTP transaction systems, which simulates a warehouse-centric order processing application and partitions all data based on their warehouse IDs. The warehouse count determines the contention degree of TPC-C. We adopt two contention configurations: one warehouse per server to model high contention and one warehouse per thread to model



(a) Under high contention.



(b) Under medium contention.

Figure 8: Throughput and average latency of TPC-C/N.

medium contention. We implement two kinds of TPC-C, named TPC-C/NP and TPC-C/N. TPC-C/NP contains two representative transactions, `NewOrder` and `Payment`, comprising 88% of the default TPC-C mix. TPC-C/N only includes the most complicated transaction `NewOrder` since some compared systems do not implement `Payment`.

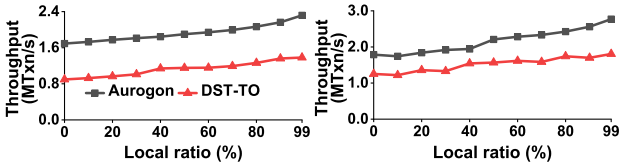
YCSB is a benchmark commonly used for key-value store evaluation as well as transaction system evaluation [29, 42]. We list two main configurable parameters in YCSB: the RMW request ratio `RMW_ratio` and the skew factor θ. Each transaction contains multiple requests (8 in our evaluation) and each request is either read or RMW determined by `RMW_ratio`. Each request accesses a random record based on the Zipf distribution and the θ shows the degree of skew in data access (a larger value means more skew). The YCSB benchmark uses 2 M records in total, uniformly partitioned among servers.

Testbed. All experiments were conducted on a cluster with 5 servers. Each server has two 10-core Intel Xeon Silver 4210R processors and 64GB DRAM, running CentOS 7.6. Each server is equipped with a ConnectX-5 MCX556A 100Gbps Infiniband NIC connected to a Mellanox SB7890 Infiniband Switch.

6.2 TPC-C Results

Figure 8 and Figure 9 illustrate the aggregated throughput and average latency of evaluated distributed transaction systems under TPC-C/N and TPC-C/NP, respectively. We vary the `local_item_ratio` from 0% to 99% to adjust the ratio of records that `NewOrder` transactions access in remote servers.

Under high contention. Figure 8(a) reveals the performance results in a high contention scenario using TPC-C/N with one warehouse per server. When `local_item_ratio` is small, most of accessed records reside on remote servers, resulting in more network communications. Aurogon improves throughput by 1.55×-3.16× compared with other systems



(a) Under high contention (b) Under medium contention

Figure 9: Throughput and average latency of TPC-C/NP.

when `local_item_ratio` is 0%, which is attributed to Aurogon's capability to reduce aborts caused by accesses from different servers.

The runner-up of performance is DrTM+H. Although it enumerates the RDMA primitive combinations to accelerate network accesses, it suffers throughput degradation as RMWs on hotspots lead to many transaction aborts. Each `NewOrder` contains 11 RMWs on average and the accessed records are nonuniform. Concurrent RMW operations result in a high possibility of transaction aborts in the validation phase.

Besides, DST-TO is a clock-driven transaction system utilizing DST to achieve high scalability. Trading accuracy for scalability in DST decreases the throughput under high contention as inaccurate distributed clocks trigger request reordering. Two reasons lead to the inaccuracy of DST. First, the timestamps that DST uses to update lagging clocks are inaccurate since DST does not consider one-way network delay when obtaining these timestamps. Second, the frequency drift among distributed clocks cannot be corrected in DST.

With increasing `local_item_ratio`, the throughput of Aurogon rises due to the growing local accesses. Note that DST-TO improves the throughput a lot since local accesses use the same clock to get timestamps, easing the reordering. However, Aurogon still outperforms the runner-up (DrTM+H) in throughput by 11% when all transactions are local.

Consequently, Aurogon increases throughput by 1.11×-4.12× compared with other systems, reaching 1.27M transactions per second (0% `local_item_ratio`) and 2.58M (99% `local_item_ratio`, default configuration in standard TPC-C). Meanwhile, Aurogon cuts average latency by up to 86%.

When mixing `NewOrder` and `Payment`[5], the aggregated throughput of Aurogon further expands since `Payment` accesses less records. Figure 9(a) shows that Aurogon outperforms DST-TO by 70%-99% in throughput and reduces average latency by 62%-67%.

Under medium contention. Figure 8(b) plots the systems' performance using TPC-C/N with one warehouse per thread. When each warehouse is bound to a dedicated thread, the contention is alleviated since some variables in `NewOrder` (*e.g.*, `next_o_id`) are never shared among threads.

When `local_item_ratio` is 50%, Aurogon improves throughput by 28%-82% compared to peer systems. The throughput of all systems rises gradually with the growing

[5]We only compare Aurogon with DST-TO under TPC-C/NP and YCSB because other peer systems do not implement these workloads.

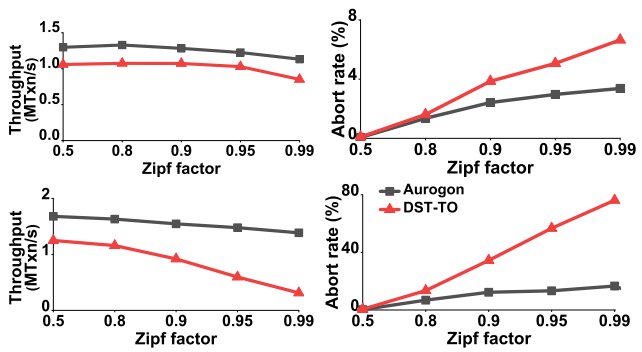


Figure 10: Performance under YCSB (100% `RMW_ratio` in the top two and 50% `RMW_ratio` in the bottom two).

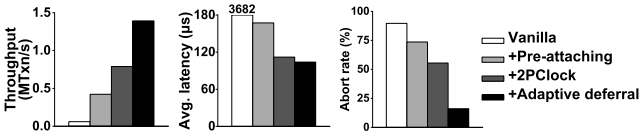


Figure 11: Incremental impact of proposed techniques.

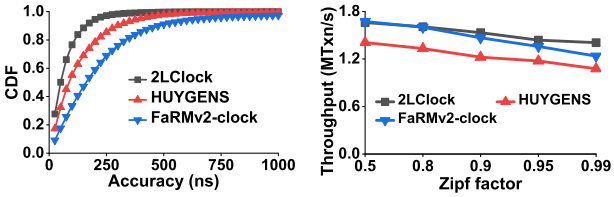
`local_item_ratio` because the contention shrinks. Aurogon shows a moderate slowdown by 1.14× compared to DrTM+H when `local_item_ratio` is 99%. This is because recycling stale versions in Aurogon consumes extra CPU resources when memory consumption rises due to the increase of warehouse count.

Moreover, Figure 9(b) shows that Aurogon outperforms DST-TO by 19%-42% in throughput and reduces average latency by 24%-34% under TPC-C/NP.

6.3 YCSB Results

Figure 10 shows the performance comparison of Aurogon and DST-TO under YCSB with skew factor θ varying from 0.5 to 0.99. When all requests are RMWs, Aurogon increases throughput by 19%-33% compared to DST-TO.

When `RMW_ratio` decreases to 50%, the throughput of both systems rises under low contention (θ is 0.5). Read requests are executed faster than RMWs since reads only require one network communication and never fail. Unfortunately, DST-TO suffers a 4× throughput slowdown when θ rises from 0.5 to 0.99. It is because the throughput improvement increases the running request count in the system. Although conflicts never occur between reads, more reads will make incoming RMWs' execution fail since reads may extend old data version's *rts* larger than RMW's timestamp. So the abort rate of DST-TO reaches up to 76%. On the contrary, Aurogon still maintains a low abort rate (16%) when θ is 0.99, since adaptive request deferral tolerates late arrivals of RMWs. If `RMW_ratio` further decreases, the performance improvement of Aurogon will shrink because Aurogon does not target read-dominant workloads.



(a) CDF of accuracy (b) Throughput in YCSB

Figure 12: (a) CDF of accuracy under CPU interference from foreground transaction system. (b) Aurogon's performance with different clocks.

6.4 Impact of Individual Techniques

To isolate the improvement brought by Aurogon's key techniques, we implement a vanilla version of Aurogon and add three techniques into Aurogon in turn. Here we take the performance under YCSB as an example, setting θ to 0.99 and `RMW_ratio` to 50%.

Figure 11 illustrates that pre-attaching increases the throughput by 7.01× and cuts the abort rate by 16% compared to vanilla Aurogon. Pre-attaching is significant in high contention workload as it not only saves one network communication but also avoids RMW's execution being interrupted.

2LClock further brings a 87% throughput improvement and reduces the average latency by 33%. The decrease of average latency mainly comes from reducing the P99 latency. Specifically, the overhead of retrying transactions after aborts incurs long tail latency since the high contention extends the transaction's execution time and increases the abort rate. Aurogon benefits from 2LClock's high accuracy (41ns) and boosts performance.

Finally, Aurogon improves the throughput by 70% after adding adaptive request deferral. One may wonder whether request deferral is compatible with 2LClock. In fact, 2LClock helps to shorten the required deferral time. To achieve the same abort rate, our test shows that 2LClock requires a deferral time of 24.1µs while DST requires 38.4µs.

6.5 Performance and Impact of 2LClock

We first evaluate the accuracy of 2LClock under foreground CPU interference. Here we compare 2LClock with two state-of-the-art clock synchronization approaches: HUYGENS [16] and FaRMv2-clock [40]. HUYGENS synchronizes the NIC clock of distributed servers with limited CPU involvement. FaRMv2-clock obtains timestamps directly from CPU to synchronize clocks. We remove FaRMv2-clock's guarantee of global increasing monotonicity by skipping its uncertainty wait and support it to provide timestamps directly.

We adopt a common way [16] to test the clock accuracy: two clocks (C_1, C_2) are started on the same NUMA node of one server, and synchronized with the clock C_3 on another server, respectively. The discrepancy of C_1 and C_2 would be 0 since they use the same clock. We take the absolute value of measured discrepancy between C_1 and C_2 as the clock error.

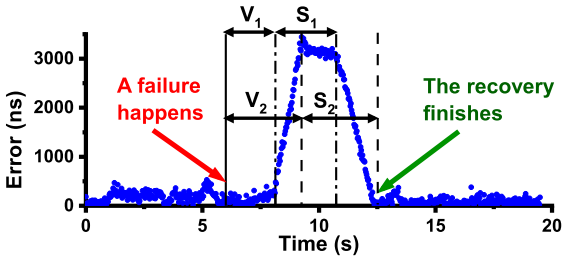


Figure 13: 2LClock failure handling process.

Figure 12(a) shows error CDFs of three clocks under foreground transaction load. The average clock error of 2LClock is 41ns, achieving a cut of 51% and 87% compared to HUYGENS and FaRMv2-clock, respectively. Furthermore, 2LClock reduces the P99 errors by 2.3× and 31× compared to HUYGENS and FaRMv2-clock. Note that FaRMv2-clock encounters a large accuracy fluctuation since CPU timestamps are inaccurate under foreground interference.

We further replace 2LClock with HUYGENS and FaRMv2-clock in Aurogon followed by evaluating these systems under YCSB to show 2LClock's advantage. Figure 12(b) illustrates that 2LClock helps Aurogon to improve the throughput by up to 31% and 14% compared to HUYGENS and FaRMv2-clock, respectively. Aurogon with HUYGENS shows the lowest throughput since obtaining timestamps from HUYGENS takes a longer time (3.3µs) than 2LClock (200ns).

6.6 2LClock Failure Handling

We evaluate how 2LClock handles a server failure of the root clock. Figure 13 plots the clock accuracy in this process. We use the same configuration in §6.5 and simulate the situation where C_1 and C_2 turn to synchronize with a new reference clock C_4 after the crash of C_3.

As illustrated in Figure 13, 2LClock detects the crash of C_3 at time 6s. Then, C_1 and C_2 start their virtual phase (V_1 and V_2), in which they build connections with C_4 and warm up the new clock. Note that they still use the previous measured value of C_3 although C_3 crashes in this phase.

C_1 finishes virtual phase first, taking 2.1s and starts to use the new clock C_4. To smooth the correction value, C_1 begins the smoothing phase S_1, which generates a rising error up to 3.5µs. Then C_2 starts S_2 as well and the error becomes stable since two clocks smooth the correction value at the same speed. As C_1 finished S_1, we can see an error curve plunge before S_2 ends, after which 2LClock turns to normal state. This total process takes 6.4s in our experiment.

6.7 Discussion of Scalability

Here we discuss how Aurogon performs in a large cluster. The rising server count results in more network hops when accessing data. Meanwhile, the network heterogeneity leads to more nonuniform data access latency in the cluster.

Let us examine the three techniques in Aurogon when scaling to a large cluster. First, if distributed clocks aim at achieving high accuracy in a large cluster, the problem is the high CPU usage of parent servers in synchronization topological tree. 2LClock utilizes asynchronous transfers and a combination of two RDMA modes to solve this problem. Second, the more nonuniform data access latency makes the adaptive deferral more effective to tolerate straggling requests. Third, pre-attaching method, saving one RTT for RMWs, is more effective when the network delay rises.

7 Related Work

T/O transaction systems. T/O systems [2,7,15,29,40,47,53–55] lie in two categories. 1) Clock-driven approach: DAST [7] determines timestamps of transactions with a two-phase protocol to anticipate the best execution timing. Cicada [29] separates read and write timestamps during allocation to accelerate read-only transactions. 2) Data-driven approach: TicToc [53] first obtains the data dependencies and determine the transactions' timestamps in commit phase.

Clock synchronization. HUYGENS [16] exploits the network effect to synchronize NIC clocks. Spanner [10], FaRMv2 [40] and Sundial-Clock [28] utilize uncertainty wait to ensure increasing monotonicity of global clocks. The monotonic guarantee is orthogonal to 2LClock since 2LClock can provide it with moderate modifications. Furthermore, 2LClock increases the accuracy of distributed clocks.

RDMA-enabled transaction systems. FaRM [13], DrTM [49], FaSST [24], and DrTM+H [48] exploit RDMA networks to accelerate distributed transaction processing. Their core target is to fully utilize CPU resources to saturate RDMA's high bandwidth. Aurogon proposes a new idea that RDMA can help to reduce transaction aborts.

8 Conclusion

In this work, we propose, implement and evaluate Aurogon, an all-phase reordering-resistant distributed in-memory transaction system. We alleviate request reordering in all phases by three techniques: high-accuracy clock synchronization 2LClock, adaptive request deferral, and pre-attaching dependent requests to data. Aurogon reduces distributed transaction aborts significantly and boosts the performance. The source code of Aurogon is available at `https://github.com/THU-jty/Aurogon.git`.

Acknowledgements

We thank all reviewers for their insightful comments and helpful suggestions, and especially our shepherd Florentina Popovici for her guidance during our camera-ready preparation. We also thank Xingda Wei and Kezhao Huang for helping us run peer systems. This work was supported by the National Natural Science Foundation of China under Grant 62025203.

References

[1] Daniel Abadi. Correctness Anomalies Under Serializable Isolation. http://dbmsmusings.blogspot.com/2019/06/correctness-anomalies-under.html, 2019.

[2] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[3] Michael J Cahill, Uwe Röhm, and Alan D Fekete. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)*, 34(4):1–42, 2009.

[4] 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.

[5] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *Proceedings of the Fourth International Conference on Weblogs and Social Media (ICWSM'10)*, volume 4, 2010.

[6] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. Hotring: A hotspot-aware in-memory key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, pages 239–252, 2020.

[7] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys'21)*, pages 210–227, 2021.

[8] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*, pages 1–17, 2016.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[10] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[11] The Transaction Processing Council. TPC Benchmark C. http://www.tpc.org/tpcc/, 2010.

[12] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 401–414, 2014.

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

[15] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 173–184. IEEE, 2013.

[16] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 81–94, 2018.

[17] Hossein Golestani, Amirhossein Mirhosseini, and Thomas F Wenisch. Software data planes: You can't always spin to win. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 337–350, 2019.

[18] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD'21)*, pages 658–670, 2021.

[19] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment*, 10(5):553–564, 2017.

[20] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[21] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. IEEE Standard 1588 (2008).

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

[23] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC'16)*, pages 437–450, 2016.

[24] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, pages 185–201, 2016.

[25] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[26] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 5(4):298–309, 2011.

[27] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 454–467, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 1171–1186, 2020.

[29] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*, pages 21–35, 2017.

[30] Linux. ibv_create_cq_ex. https://man7.org/linux/man-pages/man3/ibv_create_cq_ex.3.html, 2016.

[31] 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, 2017.

[32] David L Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.

[33] Pulkit A Misra, Jeffrey S Chase, Johannes Gehrke, and Alvin R Lebeck. Enabling lightweight transactions with precision time. *ACM SIGARCH Computer Architecture News*, 45(1):779–794, 2017.

[34] 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, 2013.

[35] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 511–524, 2014.

[36] NVIDIA. Time-Stamping Service in RDMA. https://docs.mellanox.com/display/OFEDv502180/Time-Stamping#TimeStamping-EnablingTime-Stamping, 2021.

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

[38] Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[39] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM International Conference on Management of Data (SIGMOD'12)*, pages 61–72, 2012.

[40] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*, pages 433–448, 2019.

[41] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD'20)*, pages 1493–1509, 2020.

[42] Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. An analysis of concurrency control protocols for in-memory databases with ccbench. *Proceedings of the VLDB Endowment*, 13(13):3531–3544, 2020.

[43] Mellanox Technology. RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf, 2015.

[44] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 18–32, 2013.

[45] Chao Wang and Xuehai Qian. RDMA-enabled Concurrency Control Protocols for Transactions in the Cloud Era. *IEEE Transactions on Cloud Computing*, 2021.

[46] 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. Association for Computing Machinery, 2022.

[47] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. Unifying timestamp with transaction ordering for mvcc with decentralized scalar timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, pages 357–372, 2021.

[48] 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, 2018.

[49] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, pages 87–104, 2015.

[50] Wikipedia. DDR4 SDRAM. https://en.wikipedia.org/wiki/DDR4_SDRAM, 2014.

[51] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 191–208, 2020.

[52] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. Ship Compute or Ship Data? Why Not Both? In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, pages 633–651, 2021.

[53] 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 (SIGMOD'16)*, pages 1629–1642, 2016.

[54] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. Sundial: harmonizing concurrency control and caching in a distributed oltp database management system. *Proceedings of the VLDB Endowment*, 11(10):1289–1302, 2018.

[55] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment*, 10(6), 2017.

[56] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD'20)*, pages 511–526, 2020.

DedupSearch: Two-Phase Deduplication Aware Keyword Search

Nadav Elias*, Philip Shilane†, Sarai Sheinvald§, Gala Yadgar*
*Computer Science Department, Technion †Dell Technologies §ORT Braude College of Engineering

Abstract

Deduplication is widely used to effectively increase the logical capacity of large-scale storage systems, by replacing redundant chunks of data with references to their unique copies. As a result, the logical size of a storage system may be many multiples of the physical data size. The many-to-one relationship between logical references and physical chunks complicates many functionalities supported by traditional storage systems, but, at the same time, presents an opportunity to rethink and optimize others. We focus on the offline task of searching for one or more byte strings (keywords) in a large data repository.

The traditional, *naïve*, search mechanism traverses the directory tree and reads the data chunks in the order in which they are referenced, fetching them from the underlying storage devices repeatedly if they are referenced multiple times. We propose the *DedupSearch* algorithm that operates in two phases: a physical phase that first scans the storage sequentially and processes each data chunk only once, recording keyword matches in a temporary result database, and a logical phase that then traverses the system's metadata in its logical order, attributing matches within chunks to the files that contain them. The main challenge is to identify keywords that are split between logically adjacent chunks. To do that, the physical phase records keyword prefixes and suffixes at chunk boundaries, and the logical phase matches these substrings when processing the file's metadata. We limit the memory usage of the result database by offloading records of tiny (one-character) partial matches to the SSD/HDD, and ensure that it is rarely accessed.

We compare DedupSearch to the naïve algorithm on datasets of different data types (text, code, and binaries), and show that it can reduce the overall search time by orders of magnitude.

1 Introduction

Deduplication first appeared with backup storage systems holding weeks of highly redundant content [32, 43, 48], with the purpose of reducing the physical capacity required to store the growing amounts of logical backup data. This is achieved by replacing redundant chunks of data with references to their unique copies, and can reduce the total physical storage to 2% of the logical data, or even less [43]. Deduplication has recently become a standard feature of many storage systems, including primary storage systems that support high IOPS and low latency accesses [18, 41]. Even with the lower redundancy levels in such systems, deduplication may reduce the required physical capacity to 12%-50% of the original data's size [18].

Most storage architectures distinguish between the logical view of files and objects and the physical layout of blocks of data on the storage media. In deduplicated storage, this distinction further creates multiple logical pointers, often from different files and even users, to each physical chunk. This many-to-one relationship complicates many functionalities that are supported by traditional storage systems, such as caching, capacity planning, and support for quality of service [25, 33, 40]. At the same time, it presents an opportunity to rethink other functionalities to be deduplication-aware and more efficient.

Keyword search is one such functionality, which is supported by some storage systems and is a necessary operation for numerous tasks. For example, an organization may need to find a document containing particular terms, and if the search is mandated by legal discovery [37], it has to be applied to backup systems [45] and document repositories that may include petabytes of content. Virus scans and inappropriate content searches may also include a phase of scanning for specified byte strings corresponding to a virus signature or a pirated software image [29, 44]. Finally, data analysis and machine learning tools often rely on preprocessing stages to identify relevant documents with a string search. Our focus is on *offline* search of large, deduplicated storage systems for legal or analytics purposes.

Logging and data analytics systems support fast keyword searches by constructing an index of strings during data ingestion [2, 8]. While they provide very fast lookup times, such indexes carry non-negligible overheads: their size is proportional to the logical size of the data, and thus they may consume a large fraction of the physical storage capacity [6, 31]. In addition, their data structures must be continually updated as new data is received. Thus, an index is typically not maintained in systems where search is a rare operation, such as backups. Another limitation of index structures is that they often assume a delimiter set such as whitespace, which is not useful for binary strings or more complex keyword patterns.

When an index is unavailable or cannot support the search query, an exhaustive scan of the data is required. A *naïve search* algorithm would process a file system by progressing through the files, opening each file, and scanning its content for the specified keywords. Even without the effects of deduplication, traversing the file system in its logical 'tree' order is inefficient due to fragmentation and resulting random accesses. With deduplication, a given chunk of data may be read repeatedly from storage, once for every file that references it.

We propose an alternative algorithm, *DedupSearch*, that progresses in two main phases. We begin with a *physical phase* that performs a physical scan of the storage system and scans each chunk of data for the keywords. This has the twin benefits of reading the data sequentially with large I/Os as well as reading each chunk of data only once. For each chunk of data, we record the exact matches of the keyword, if it is found, as well as prefixes or suffixes of the keyword (partial matches) found at chunk boundaries. We use a widely-used [11] string-matching algorithm to efficiently identify multiple keywords in a single scan, as well as their prefixes and suffixes.

We then continue with a *logical phase* that performs a logical scan of the filesystem by traversing the chunk pointers that make up the files. Instead of reading the actual data chunks, we check our records of exact and partial matches in those chunks, and whether partial matches in logically adjacent chunks complete the requested keyword. This mechanism lends itself to also supporting standard search parameters such as file types, modification times, paths, owners, etc.

The database of chunk-level matches generated during the physical scan can become excessively large when a keyword begins or ends with common byte patterns or characters, such as 'e'. Our experiments show that very short prefix and suffix matches can become a sizable fraction of the database even though they are rarely part of a completed query. We separate records of "tiny" partial matches into a dedicated database which is written to SSD/HDD and is accessed only when the tiny prefix/suffix is needed to complete the keyword match.

We implemented DedupSearch in the Destor open-source deduplication system [21], and evaluated it with three real-world datasets containing Linux kernel versions, Wikipedia archives, and virtual machine backups. DedupSearch is faster than the naïve search by orders of magnitude: its search time is proportionate to the physical size of the data, while the naïve search time increases with its logical size. Despite its potential overheads, the logical phase becomes dominant only when the number of files is very large compared to the size of the physical data, as is the case in the archives of the Linux kernel versions. Even in these use cases, DedupSearch outperforms the naïve search thanks to its efficient organization of the partial results, combined with reading each data chunk only once. These advantages are maintained when searching for multiple keywords at once and when varying the average chunk size and number of duplicate chunks in the system.

2 Background and Challenges

Data in deduplicated systems is split into chunks, which are typically 4KB-8KB in average size. Duplicate chunks are identified by their *fingerprint*—the result of hashing the chunk's content using a hash function with very low collision probability. These fingerprints are also used as the chunks' keys in the fingerprint-index, which contains the location of the chunk on the disk. When a new chunk is identified, it is written into a *container* that is several MBs in size to optimize disk writes. A container is written to the disk when it is full, possibly after its content is compressed. A file is represented by a *recipe* that lists the fingerprints of the file's chunks. Reading a file entails looking up the chunk locations in the fingerprint index, reading their containers (or container sub-regions) from the disk, and possibly decompressing them in memory.

Consider, for example, the four files in Figure 1(a). Each file contains two chunks of 5 bytes each, where some of the chunks have the same content. The total logical size of these files is eight chunks, and this is also their size in a traditional storage system, without deduplication. Figure 1(b) illustrates how these files will be stored in a storage system with deduplication. We assume, for simplicity, that the files were written in order of their IDs, and that the chunks are all of size 5 bytes.[1] When deduplication is applied, only four unique chunks are stored in the system, in two 10-Byte containers.

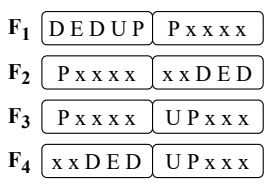
A keyword search in a traditional storage system would scan each files' chunks in order, with a total of eight sequential chunk reads. The same naïve search algorithm can also be applied to the deduplicated storage: following the file recipes it would scan the chunks in the following order: $C_0, C_1, C_1, C_2, C_1, C_3, C_2, C_3$—a total of eight chunk reads. If this access pattern spans a large number of containers (larger than the cache size), entire containers might be fetched from the disk several times. Moreover, the data in each chunk will be processed by the underlying keyword-search algorithm multiple times—once for each occurrence in a file.

Our key idea is to read and process each chunk in the system only once. Our algorithm begins with a *physical phase*, which reads all the containers in order of their physical addresses, and processes each of their chunks. In our example, we will perform two sequential container reads, and process a total of four chunks. The challenges in searching for keywords in the physical level result from the fact that most deduplication systems do not maintain "back pointers" from chunks to the files that contain them. Thus, we cannot directly associate keyword matches in a chunk with the corresponding file or files. Furthermore, keywords might be split between adjacent chunks in a file, preventing the identification of the keyword when searching the individual chunks.

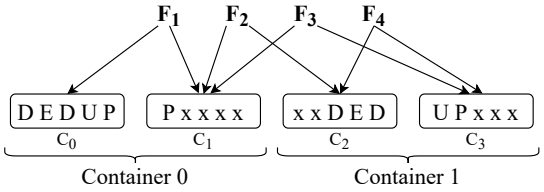
Consider, for example, searching for the keyword DEDUP in the files in Figure 1. The naïve search will easily identify the matches in files F_1 and F_4, even though the word is split between chunks C_2 and C_3. The physical search will only identify the exact match of the word in chunk C_0 but will not be able to correlate it with file F_1 or identify F_4 as a match.

To address these challenges, we add a *logical phase* following the completion of the physical phase, that collects the matches within the chunks and identifies the files that contain them. To identify keywords split between chunks, we must also record *partial matches*—prefixes of the keyword that appear at the end of a chunk and suffixes that appear at the beginning of

[1] At the host level, files are split into blocks. We assume, for this example, that each host-level block corresponds to a deduplication-level chunk.



(a) Traditional system



(b) Deduplicated system

FP	|Prefix|	|Suffix|	# Exact	Offset
FP_0	0	0	1	0
FP_1	0	1	0	0
FP_2	3	0	0	0
FP_3	0	2	0	0

(c) Chunk results

Figure 1: Four files containing four unique chunks in a traditional storage system (a) and in a deduplicated system (b), with their corresponding chunk-result records (c).

a chunk. For example, in addition to recording the full match in chunk C_0, the physical phase will also record the prefix of length 3 in the end of chunk C_2, and the suffix of size 2 in the beginning of chunk C_3. We must also record the suffix of length 1 in chunk C_1, to potentially match it with the prefix DEDU, even though this prefix does not appear in any chunk.

This introduces an additional challenge: some prefixes and suffixes might be very frequent in the searched text. Consider, for example, a keyword that begins with the letter 'e', whose frequency in English text is 12% [23]. Recording all prefix matches means we might have to record partial matches for 12% of the chunks in the system. In other words, the number of partial matches we must store during the physical phase is not proportionate to the number of keyword matches in the physical (or logical) data. This problem is aggravated if we search for multiple keywords during the same physical scan. In the worst case, we might have to store intermediate results for all or almost all the chunks in the system. In the following, we describe how our design addresses these challenges.

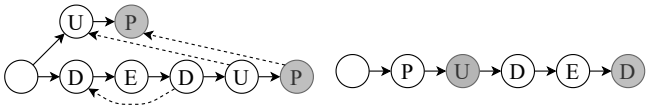
3 The Design of DedupSearch

We begin by describing the underlying keyword-search algorithm and how it is used to efficiently identify partial matches during the physical search phase. We then describe the data structures used to store the exact and partial matches between the two phases. Finally, we describe how the in-memory and on-disk databases are accessed efficiently for the generation of the full matches during the logical phase.

3.1 String-matching algorithm

To identify keyword matches within chunks, we use the *Aho-Corasick* string-matching algorithm [11]. This is a trie-based algorithm for matching multiple strings in a single scan of the input. We explain here the details relevant for our context, and refer the reader to the theoretical literature for a complete description of the algorithm and its complexity.

The *dictionary*—set of keywords to match—is inserted into a trie, which represents a finite-state deterministic automaton. The root of the trie is an empty node (*state*), and the edges between consecutive nodes within a keyword are called *child links*. Each child link represents a state transition that occurs when the next character in the input matches the next character in the keyword. Thus, each node in the trie represents the occurrence in the input of the substring represented by the path



(a) Search trie (b) Reverse trie

Figure 2: The Aho-Corasick trie (a) and reverse trie (b) for the dictionary {DEDUP,UP}

to that node. Specifically, each leaf represents an exact match of its keyword in the dictionary and is thus an accepting state in the automaton.

In addition to the child links, a special link is created between node u and node v whenever v is the longest strict suffix of u in the trie. These links are mainly used when the matching of an entire keyword fails, and are thus referred to in the literature as *failure links*. For example, Figure 2(a) illustrates the trie created for the dictionary {DEDUP,UP}, where the dashed arrows represent the failure links.

The characters in the input are used to traverse the automaton. If an accepting state is reached, the algorithm emits the corresponding keyword and its location in the input. If the search fails in an internal node (i.e., when the next character in the input does not correspond to any child link) with a failure link, this means that the substring at the end of the link occurs in the input, and the search continues from there. For example, if the input is DEDE, then after reading the first three characters we will reach the node corresponding to DED. After the next character, E, we will backtrack to the node corresponding to D, continuing the search from the same input location, immediately transitioning to the next node by traversing the child link E. There is an implicit failure link to the root from every node that does not have an explicit failure link to another node.

The failure links guarantee the linear complexity of the algorithm: they prevent it from having to backtrack to earlier positions in the input whenever one keyword is found, or when the search fails. For example, when the string DEDUP is identified in the input, the failure link to the node representing UP allows the algorithm to emit *all* the keywords that occur in the input so far, continuing the search from the current location. The overall complexity of the Aho-Corasick search is linear in the total length of the dictionary plus the length of the input plus the number of keyword matches.

We use the Aho-Corasick algorithm with minimal modification to identify keyword prefixes. When the end of a chunk is reached and the current state is an internal node, then this

node's corresponding substring is the longest substring of at least one keyword. We can traverse the path of failures links starting from this node and emit all the longest prefixes found. For example, if the chunk ends with the string DEDU, then the current state corresponds to this prefix of DEDUP. The failure link points to U, which is the longest prefix of UP.

To identify suffixes at the beginning of a chunk, we construct a trie for the *reverse dictionary*—the set of strings which are each a reverse of a string in the original dictionary. We use it to search, in reverse order, the first n bytes of the chunk, where n is the length of the longest string in the dictionary. For example, Figure 2(b) shows the trie for the reverse dictionary of {DEDUP,UP}. To find the suffixes in chunk C_3 from Figure 1(b), we use this trie on the (reverse) input string "XXXPU".

Partial matches. As demonstrated in Figure 1, keywords might be split between adjacent chunks. Let n denote the length of the keyword, and p_i and s_i denote a prefix and a suffix of length i, respectively. p_i and s_i are considered *prefix or suffix matches* if they constitute the last or first i characters in the chunk, respectively. A *full match* occurs if the jth chunk in the file contains a prefix match of length i and the $(j+1)$th chunk (likely not stored consecutively with the jth chunk) contains a suffix match of length $n-i$.

In some cases, a chunk may contain several prefix or suffix matches. For example, chunk C_2 in Figure 1(b) contains p_3=DED as well as p_1=D. Thus, this prefix can be part of two possible full matches if the following chunk contains either s_2=UP or s_4=EDUP. To minimize the size of the partial results generated by the physical phase, we record only the longest prefix and longest suffix in each chunk, if a partial match is found. Note that if a chunk contains a prefix match of length i (e.g., DED) and some suffix of this prefix is itself a prefix of size $j < i$ of the keyword (e.g., D), then the partial match of p_j is implied by the record of the match p_i.

To facilitate the identification of all possible full matches, we construct, for each keyword, the set of all prefix and suffix matches. For example, for the word DEDUP, a full match can be generated by combining the following pairs of longest partial matches: D+EDUP, DE+DUP, DED+UP, DEDU+P, and DED+EDUP. The pairs can be represented by a set of integer pairs corresponding to the substring lengths: $\{(1,4),(2,3),(3,2),(4,1),(3,4)\}$. This set is constructed offline, before the start of the logical phase. We store it in the *partial-match table*, which is kept in memory for the duration of the logical phase. It is implemented as a two dimensional array such that cell (i,j) holds a list of all match offsets found in $p_i + p_j$. For example, Table 1 is the partial-match table for keyword DEDUP, where the offsets are calculated with respect to the beginning of the prefix. For example, the entry $(3,4)$ indicates that a match begins two characters after the beginning of the partial match DED. During the logical phase, when adjacent chunks contain a prefix p_i and a suffix s_j, we check the table for the pair (i,j) to determine if and where a full match is found.

	$j=1$	2	3	4
$i=1$				0 [D+EDUP]
2			0 [DE+DUP]	
3		0 [DED+UP]		2 [DED+EDUP]
4	0 [DEDU+P]			

Table 1: Partial-match table for DEDUP

3.2 Match result database

Exact matches. Exact matches are identified within individual chunks during the physical phase. We record the existence of an exact match by the offset of its first character.A chunk may contain several exact matches, which would require recording an arbitrarily large number of offsets. In practice, however, the vast majority of the chunks contain at most one exact match. This led us to define our basic data structures as follows.

Chunk-result record: this is the basic record of search results in a single chunk. It contains five fields: `fingerprint` (20 bytes), `longest prefix length` (1 byte), `longest suffix length` (1 byte), `number of exact matches` (1 byte), and `offset` of the first exact match (2 bytes). The total (fixed) size of this object is 26 bytes, although it might vary with the system's fingerprint and maximum chunk sizes. Figure 1(c) shows the content of the chunk-result records for the chunks in Figure 1(b), when searching for the keyword DEDUP.

Location-list record: this is a variable-sized list of the locations which is allocated (and read) only if the chunk contains more than one exact match. The first field is the `fingerprint` (20 bytes), and the remaining fields contain one offset (within the chunk), each. The number of offset fields is recorded in the `number of exact matches` field in the corresponding chunk-result record. The value 255 is reserved to indicate that there are more than 254 exact matches in the chunk. In that case, we use the following alternative record.

Long location-list record: this object is identical to the location-list record, except for one additional field. Following the chunk fingerprint, we store the precise number of exact matches, whose value determines the number of offset fields in the record.

Tiny substrings. Keywords that begin or end with frequent letters in the alphabet might result in the allocation of numerous chunk-result records whose partial matches never generate a full match. To prevent these objects from unnecessarily inflating the output of the physical phase, we record them in a different record type and store them in a separate database (described below). Each *tiny-result record* contains three fields: `fingerprint` (20 bytes) and two Booleans, `prefix` and `suffix`, indicating whether the chunk contains a prefix match or a suffix match, respectively.

The tiny-result records are allocated only if this is the only match in the chunk, i.e., the chunk does not contain any exact match nor a partial match longer than one character. For example, the chunk-result record for chunk C_1 in Figure 1(c) will be replaced by a tiny-result record. Tiny-result records are accessed during the logical phase only if the adjacent chunk

contains a prefix or suffix of length $n-1$.[2]

We use tiny-result records for substrings of a single character: our results show that this captures the vast majority of tiny substrings, as we expected from text in a natural language [23]. However, when searching for non-ASCII keywords, we might encounter different patterns of tiny frequent substrings. The tiny-result records could then be used for variable-length substrings which are considered short. In this case, the record would contain an additional field indicating the length of the substring. To improve space utilization, several Boolean fields can be implemented within a single byte.

Multiple keywords. When the dictionary includes multiple keywords, we list them and assign each keyword its serial number as its ID. We then replace the individual per-chunk records with lists of <keyword-ID,result-fields> pairs. The structure of the records (chunk-result, locations-list, and tiny-result) is modified as follows. It includes one copy of the chunk fingerprint, followed by a list of <keyword-ID,result-fields> pairs. The result fields correspond to the fields in each of the three original records, and a pair is allocated for every keyword with non-empty fields. For example, if we were searching for two keywords, DEDUP and UP, then the chunk-result object for chunk C_3 in Figure 1(b) would include the following fields:

FP	ID	|Prel	|Sufl	#Exact	Off	ID	|Prel	|Sufl	#Exact	Off
FP_3	0	0	2	0	0	1	0	0	1	0

Database organization. We store the output of the physical search phase in three separate databases, where the chunk fingerprint is used as the lookup key. The *chunk-result index*, *location-list index*, and *tiny-result index* store the chunk-result records, location-list records, and tiny records, respectively. The first two databases are managed as in-memory hash tables. The tiny-result index is stored in a disk-based hash table. In a large-scale deduplicated system, chunks can be processed (and their results recorded) in parallel to take advantage of the parallelism in the underlying physical storage layout.

3.3 Generation of full search results

After all the chunks in the system have been processed, the logical phase begins. For each file in the system, the file recipe is read, and the fingerprints of its chunks are used to lookup result records in the database. The fingerprints are traversed in order of their chunk's appearance in the file. The process of collecting exact matches and combining partial matches for each fingerprint is described in detail in Algorithm 1, which is performed separately for every keyword.

This process starts by emitting the exact match in the chunk-result record, if a match is found (lines 4-5). If the chunk contains more than one match, it fetches the relevant location-list record and emits the additional matches (lines 6-9). If the chunk contains a suffix, it attempts to combine it with a prefix in the previous chunk (lines 10-14). If the chunk contains a prefix or a suffix of length $n-1$, then the tiny-result index is queried for the corresponding one-character suffix or prefix (lines 15-22). Thus, regular prefixes and suffixes (or tiny suffixes recorded in a regular chunk-result record) are matched when the suffix is found, while tiny substrings are matched when the respective $(n-1)$-length substring is found.

[2]This optimization is not effective for keywords of length 2. We do not include specific optimizations for this use case in our current design.

Algorithm 1 DedupSearch Logical Phase: handling FP_i in File F

Input: FP_i, FP_{i-1}, FP_{i+1}, res_{i-1}

```
1:  res_i ← chunk_result[FP_i]
2:  if res_i = NULL then
3:      return
4:  if res_i.exact_matches > 0 then
5:      add file name, match offset to output
6:      if res_i.exact_matches > 1 then
7:          locations ← list_locations[FP_i]
8:          for all offsets in locations do
9:              add file name, offset to output
10: if res_i.longest_suffix > 0 then
11:     if res_{i-1} ≠ NULL then
12:         if res_{i-1}.longest_prefix > 0 then
13:             for all matches in partial-match_table
                [res_{i-1}.longest_prefix, res_i.longest_suffix]
                do
14:                 add file name, match offset to output
15:     else if res_i.longest_suffix = n − 1 then
16:         tiny ← tiny_result[FP_{i-1}]
17:         if tiny ≠ NULL & tiny = prefix then
18:             add file name, match offset to output
19: if res_i.longest_prefix = n − 1 then
20:     tiny ← tiny_result[FP_{i+1}]
21:     if tiny ≠ NULL & tiny = suffix then
22:         add file name, match offset to output
```

The logical phase can also be parallelized to some extent: while each file's fingerprints must be processed sequentially, separate backups or files within them can be processed in parallel by multiple threads. Even for a large file, it is possible to process sub-portions of the file recipe in parallel. Both physical and logical phases can be further distributed between servers, requiring appropriate distributed result databases. This extension is outside the scope of this paper.

4 Implementation

We used the open-source deduplication system, Destor [21], for implementing DedupSearch (*DSearch*). The physical phase of DedupSearch is composed of two threads operating in parallel: one thread sequentially reads entire containers and inserts their chunks into the chunk queue. The second thread pops the chunks from the queue and processes them, as described in Sections 3.1 and 3.2: it identifies exact and partial matches of all the keywords, creates the respective result records, and stores them in their respective databases.

We used Destor's restore mechanism for implementing the logical phase. Destor's existing restore operates in three parallel threads: one thread reads the file recipes and inserts them into the recipe queue. Another thread pops the recipes from their queue, fetches the corresponding chunks by reading their containers, and inserts the chunks in order of their appearance in the file to the chunk queue. The last thread pops the chunks from their queue and writes them into the restored file.

The logical phase uses the second thread of the restore mechanism. It uses the fingerprints to fetch chunk-result records, rather than the chunks themselves, and inserts them into the result queue with the required metadata. An additional thread pops the result records from the queue, processes them according to Algorithm 1, and emits the respective full matches.

The implementation of the chunk-result index and location-list index is similar to Destor's fingerprint index. This is an in-memory hash table, whose content is staged to disk if the memory becomes full. The tiny-result index is implemented as an on-disk hash table using BerkeleyDB [5, 39]. We used BerkeleyDB's default setup with transactions disabled, because, in our current implementation, accesses to the tiny-result index are performed from a single thread in each phase.

We modified a publicly available implementation of the Aho-Corasick algorithm in C++ [22] to improve its data structures, memory locality, and suffix matching, and to support non-ASCII strings. For best integration of this implementation into Destor, we refactored the Destor code to use C++ instead of C. Our entire implementation of DedupSearch consists of approximately 1600 lines of code added to Destor, which is available online [1].

5 Evaluation Setup

For comparison with DedupSearch, we implemented the traditional (*Naïve*) search within the same framework, Destor. Naïve uses Destor's restore mechanism by modifying its last thread: instead of writing the chunk's data, it is processed with the Aho-Corasick trie of the input keywords. To identify keywords that are split between chunks, the last $n-1$ characters (where n is the length of the longest keyword) of the previous chunk are concatenated to the beginning of the current chunk.

We ran our experiments on a server running Ubuntu 16.04.7, equipped with 128GB DDR4 RAM and an Intel® Xeon® Silver 4210 CPU running at 2.40GHz. The backing store for Destor was a DellR 8DN1Y 1TB 2.5" SATA HDD, and the tiny-result index was stored on another identical HDD. We remounted Destor's partition before each experiment, to ensure it begins with a clean page cache.

5.1 Datasets

Our goal was to generate datasets that differ in their deduplication ratio and content type. To that end, we used data from three different sources—Wikipedia backups [9, 10], Linux kernel versions [4], and web server VM backups—and used Destor to create several distinct datasets from each source. Destor ingests all the data in a specified target directory, creating one backup file. This file includes the data chunks and the metadata required for reconstructing the individual files and directory tree of the original target directory. We created two or four versions of each of our datasets, each with a different average chunk size: 2KB, 4KB, 8KB, and 16KB.

Dataset	Logical size (GB)	Physical size + metadata size (GB) 2KB	4KB	8KB	16KB
Wiki-26 (skip)	1692		667+16 40.4%	861+9 51.4%	
Wiki-41 (consecutive)	2593		616+22 24.6%	838+12 32.8%	
Linux-197 (Minor versions)	58	10+1 19%	10+1 19%	11+1 20.7%	13+1 24.1%
Linux-408 (every 10th patch)	204	10+4 6.9%	10+4 6.9%	15+2 7.4%	16+2 8.8%
Linux-662 (every 5th patch)	377	10+7 4.5%	11+5 4.2%	13+4 4.5%	17+3 5.3%
Linux-1431 (every 2nd patch)	902	10+18 3.1%	11+13 2.7%	10+13 2.5%	17+8 2.8%
Linux-2703 (every patch)	1796	10+34 2.5%	10+26 2.0%	13+20 1.9%	17+17 1.9%
VM-37 (1-2 days skips)	2469	145+33 7.2%	129+18 6.0%	156+10 6.7%	192+5 8.0%
VM-20 (3-4 days skips)	1349	143+19 12.0%	125+10 10.0%	150+6 11.6%	181+3 13.6%

Table 2: The datasets used in our experiments. 2KB-16KB represent the average chunk size in each version. The value below the physical size is its percentage of the logical size.

The Linux version archive includes tarred backups of all the Linux kernel history, ordered by version, major revision, minor revision, and patch. The size of the kernel increased over time, from 32 MB in version 2.0 to 1128 MB in version 5.9.14 (the latest in our datasets). Naturally, versions with only a few patches between them are similar in content. Thus, by varying the number of versions included, we created five datasets that vary greatly in their logical size, but whose physical size is very similar, so the effective space savings increases with number of versions. All our Linux datasets span the same timeframe, but vary in the "backup frequency", i.e., the number of patches between each version. They are listed in Table 2.

The English Wikipedia is archived twice a month since 2017 [9, 10]. We used the archived versions that exclude media files, and consist of a single archive file, each. We created two datasets from these versions. Our first dataset includes 41 versions, covering three consecutive periods of 4, 5, and 15 months between 2017 and 2020 (chosen based on bandwidth considerations). To create the second dataset, we skipped every one or two versions, resulting in roughly half the logical size and almost the same physical size as the first dataset. The list of backups in each dataset appears in [19].

For experimenting with binary (non-ASCII) keywords, we created a dataset of 37 VM backups (`.vbk` files) of two WordPress servers in the Technion CS department over two periods of roughly two weeks each. The backups were generated every

one or two days, so as not to coincide with the existing, regular backup schedule of these servers. The first dataset consists of all 37 backups. The second consists of 20 of these backups, with longer intervals (three to four days) between them. Table 2 summarizes the sizes and content of our datasets.

5.2 Keywords

We created dictionaries of keywords with well-defined characteristics to evaluate the various aspects of DedupSearch. Specifically, we strived to include keywords that appear sufficiently often in the data, and to avoid outliers within the dictionaries, i.e., words that are considerably more common than others. We also wanted to distinguish between keywords with different probabilities of prefix or suffix matches, and different suffix and prefix length. Our dictionaries consist of multiple keywords, to evaluate the efficiency of DedupSearch in scenarios such as virus scans or offline legal searches.

We sampled 1% of a single Wikipedia backup (~1GB), and counted the occurrences of all the words in this sample, using white spaces as delimiters. As we expected, the frequency distribution of the keywords was highly skewed. We chose approximately 1000 words whose number of occurrences was similar (between 500 and 1000), and whose length is at least 4. We counted the number of occurrences of each keyword's prefixes and suffixes in the sample. We also calculated the average prefix and suffix length, which were less than 1.2 for all keywords, confirming that the vast majority of substring matches are of a single character. We then constructed the following three dictionaries of 128 keywords each: *Wiki-high*, *Wiki-low*, and *Wiki-med* contain keywords with the highest, lowest, and median number of prefixes and suffixes, respectively.

We repeated the process separately for Linux using an entire (single) Linux version, resulting in the corresponding dictionaries *Linux-high*, *Linux-low*, and *Linux-med*. We created an additional dictionary, *Linux-line*, that constitutes entire lines as search strings, separating strings by EOL instead of white spaces. We chose 1000 lines with a similar number of occurrences, sorted them by their prefix and suffix occurrences, and chose the lines that make up the middle of the list.

For the binary keyword dictionary, we sampled 1GB from both of the VM backups, and counted the number of occurrences of all the binary strings of length 16, 64, 256 and 1024 bytes. We chose strings with similar number of occurrences and the median number of prefix and suffix matches. The resulting dictionaries for the four keyword lengths are *VM-16*, *VM-64*, *VM-256*, and *VM-1024*. The statistics of all our dictionaries are summarized in Table 3.

Dictionary	Avg. pre/suf length	Avg. # pre/suf	Avg. # occurrences	Avg. keyword length
Wiki-high	1.09	85.3 M	722	8.4
Wiki-med	1.10	42.2 M	699	7.8
Wiki-low	1.08	5.7 M	677	6.0
Linux-high	1.09	64.8 M	653	10.5
Linux-med	1.20	32.8 M	599	10.4
Linux-low	1.13	5.7 M	583	10.4
Linux-line	1.22	31.4 M	63	25.9
VM-16	1.00	8.7 M	31	16
VM-64	1.00	8.6 M	29	64
VM-256	1.00	8.6 M	27	256
VM-1024	1.00	8.6 M	27	1024

Table 3: Characteristics of our keyword dictionaries.

6 Experimental Results

The goal of our experimental evaluation was to understand how DedupSearch (*DSearch*) compares to the Naïve search (*Naïve*), and how the performance of both algorithms is affected by the system parameters (dedup ratio, chunk size, number of files) and search parameters (dictionary size, frequency of substrings). We also wanted to evaluate the overheads of substring matching in DedupSearch, and how it varies with these system and search parameters.

6.1 DedupSearch performance

Effect of deduplication ratio. In our first set of experiments, we performed a search of a single keyword from the 'med' dictionaries, i.e., with a median number of substring occurrences. We repeated this search on all the datasets and chunk sizes detailed in Table 2. Figure 3 shows, for each experiment, the total search time and the time of the physical and logical phases of DedupSearch as compared to Naïve. The result of each experiment is an average of four independent experiments, each with a different keyword. The standard deviation was at most 6% of the average in all our measurements except one.[3]

We first observe that DedupSearch consistently outperforms Naïve, and that the difference between them increases as the deduplication ratio (the ratio between the physical size and the logical size) decreases. For example, with 8KB chunks, DedupSearch is 2.5× faster than Naïve on Linux-197 and 7.5× faster on Linux-2703. The total time of Naïve increases linearly with the logical size of the dataset, as the number of times chunks are read and processed increases. The total time of DedupSearch also increases with the number of versions. However, the increase occurs only in the logical phase, due to the increase in the number of file recipes that are processed. The time of the physical phase remains roughly the same, as it depends only on the physical size of the dataset.

Effect of chunk size. Chunk sizes present an inherent tradeoff in deduplicated storage: smaller chunks result in better deduplication, but increase the size of the fingerprint index. This tradeoff is also evident in the performance of both search algorithms. The search time of Naïve on the Linux datasets and most of the VM datasets decreases as the average chunk size increases. While this increases the physical data size, it reduces the number of times each container is read on average, as well as the number of times each chunk is processed. On the Wikipedia datasets and on the VM-37 dataset with

[3]The standard deviation of time of the logical phase in the Linux datasets was as high as 15%, due to the variation in the number of prefix and suffix matches for the different keywords.

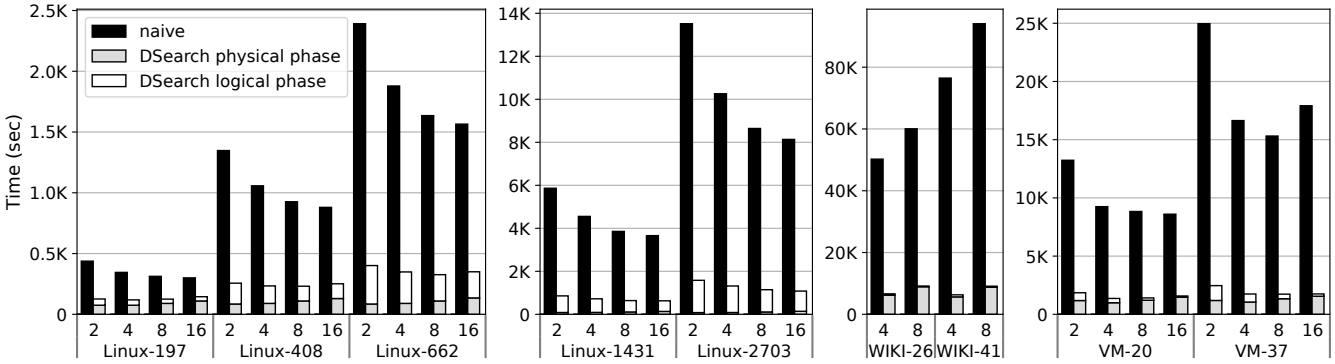


Figure 3: Search times of DedupSearch and Naïve with one word from the 'med' dictionary. The numbers 2-16 in the x-axis indicate the average chunk size in KB.

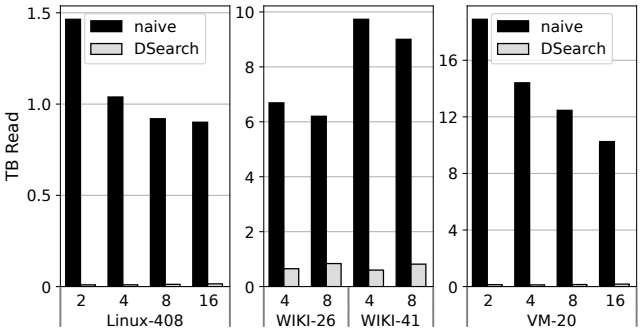


Figure 4: Amount of data read in DedupSearch and Naïve with one word from the 'med' dictionary. The numbers 2-16 in the x-axis indicate the average chunk size in KB.

16KB chunks, however, the increase in chunk size increases the search time of Naïve. The reason is their physical size, which is much larger than the page cache: although fewer containers are fetched by Destor, more of their pages miss in the cache and incur additional disk accesses.

The time of the physical phase in DedupSearch increases with the chunk size due to the corresponding increase in the data's physical size. This increase is most visible in our Wikipedia datasets, which are our largest datasets. In contrast, the logical phase is faster with larger chunks. The main reason is the reduction in the size of the file recipes and the number of chunk fingerprints they contain. Larger chunks also mean fewer chunk boundaries, which reduce the overall number of partial results that are stored and processed. These results were similar in all our datasets.

Figure 4 shows the amount of data read by both search algorithms on representative datasets. It confirms our observations that the main benefit of DedupSearch comes from reducing the amount of data read and processed by orders of magnitude, compared to Naïve. For Naïve, the amount of data read increases with the logical size and decreases with the chunk size. For DedupSearch, the amount of data read is proportionate to the physical size of the dataset, regardless of its logical size.

Effect of dictionary size. To evaluate the effect of the dictionary size on the efficiency of DedupSearch, we used subsets of different sizes from the 'med' dictionary. Figure 5 shows the results for the Linux-408 and Wikipedia-41 workloads with 8KB chunks (the results for the other datasets are similar). We repeated this experiment with two underlying keyword-search algorithms: Aho-Corasick, as explained in Section 3.1, and the native C++ `find`, described below. Both implementations use the result records, data structures, and matching algorithm described in Sections 3.2- 3.3, but differ in the way they handle multiple keywords and partial matches.

When the Aho-Corasick algorithm is used, the chunks' processing time (denoted as 'search chunks' in the figure) increases sub-linearly with the number of keywords in the search query. Nevertheless, the processing time is lower than the time required for reading the chunks from physical storage, which means that the time spent in the physical phase does not depend on the dictionary size. The logical phase, however, requires more time as the number of keywords increases: more keywords result in more exact and partial matches generated in the physical phase. As a result, more time is required to process the result records and to combine potential partial matches. We observe this increase only when the dictionary size increases beyond eight keywords. For smaller dictionaries (e.g., when comparing two keywords to one) increasing the number of keywords means that each thread of the logical phase processes more records per chunk. This reduces the frequency of accesses to the shared queues, thus reducing context switching and synchronization overheads. For example, the logical phase of Linux-408 with two keywords is five seconds faster than that with one keyword.

C++ `find` [3] scans the data until the first character in the keyword is encountered. When this happens, the scan halts and the following characters are compared to the keyword. If the string comparison succeeds, the match is emitted to the output. Regardless of whether a match was found or not, the scan then resumes from where it left off, which means the search backtracks whenever a keyword prefix is found in the data. This process is more efficient than Aho-Corasick when the number of keywords is small (see the difference in the 'search chunks' component): it's overhead is lower and its implementation is likely more efficient than our Aho-Corasick implementation. However, its search time increases linearly

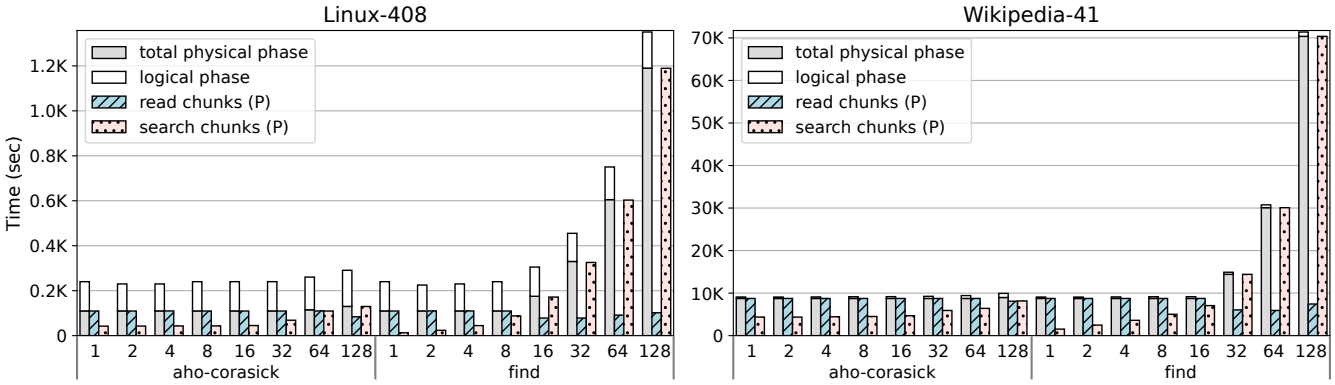


Figure 5: DedupSearch times of different number of keywords from the 'med' dictionary with Aho-Corasick vs. C++ `find` and 8KB chunks.

with the number of keywords: it exceeds the time used by Aho-Corasick when the dictionary size is 8 or higher, and its processing time exceeds the time required to read the physical chunks when the dictionary size exceeds 16 and 64, in Linux-408 and Wikipedia-41, respectively. The difference between the datasets stems from their different content: the prefixes in the Linux dictionaries are longer, which causes `find` to spend more time on string comparison.

Effect of keywords in the dictionary. To evaluate the effect of the type of keywords, we compared the search times of DedupSearch and Naïve (Figure 6) when using the full (128-word) dictionaries from Table 3 on four representative datasets: Linux-408, Linux-2703, Wikipedia-41, and VM-20, all with 8KB chunks. The results for all four binary (VM-*) dictionaries were identical, and so we present only results with 64-byte keywords. Our results show that in the physical phase, the time spent searching for keywords within the chunks increases with the number of substring occurrences: it is shortest for the 'low' dictionary and longest for the 'high' dictionary, where all the keywords start and end with popular characters (e, t, a, i, o, and '_'). The duration of the logical phase increases slightly with the number of substrings in the database, because more partial results are fetched and processed.

Surprisingly, as the chunk processing time increases, the time spent waiting for disk reads decreases. This reduction is a result of the operating system's readahead mechanism: the next container is being read in the background while the chunks in the current one are being processed. The page cache also explains the results of Naïve: it processes each chunk several times, but the processing time, which is higher with more prefixes and suffixes, is not masked by the reading time: many chunks already reside in the cache. Thus, Naïve is more sensitive to the dictionary type when searching the Linux datasets because they are small enough to fit almost entirely in memory.

6.2 DedupSearch data structures

Index sizes. Figure 7(top) shows the number of chunk-result, list-locations and tiny-result records that are generated by the physical phase when searching for a single keyword. Comparing the datasets to one another shows that the number of search results (rightmost, white bar) increases with the logical size, while the number of result records (i.e., objects stored in the database) depends only on the physical size. The results of each dataset are an average of four experiments, with four different words from the 'med' and '64' dictionaries. Unlike the performance results, the standard deviation here is larger because the results are highly sensitive to the number of substring matches of each keyword. However, the trend for each keyword is similar to the trend of the average in all the datasets.

This figure also shows that, in all the datasets, a large percentage of the records are tiny-result records (note the log scale of the y-axis). Figure 7(bottom) shows the size of each of the databases: the memory-resident chunk results and list locations, and the on-disk tiny results. The tiny results constitute 62%, 84% and 98% of the space occupied by the result databases in Linux-408, Wiki-41 and VM-20, respectively. Storing them on the disk successfully reduces the memory footprint of both logical and physical phases. The location lists occupy a small portion of the overall database size: 3%, 4% and 0% of the database size of Linux-408, Wiki-41 and VM-20, respectively. There are, on average, 3.3 offsets in each location list. Separating these offsets into dedicated records allows us to minimize the size of the more dominant chunk-result records.

Figure 8 shows the number of result records for a representative dataset, Linux-408 (the trend for the other datasets is similar), when varying the chunk size and the keyword type. When the number of chunks increases (chunk size decreases), more keyword matches are split between chunks. As a result, there are fewer exact matches and fewer list locations, but more chunk-result records with prefixes and suffixes, and more tiny-result records. The overall database size increases with the number of records, from 0.82 MB for 16KB chunks to 2.28 MB with 2KB chunks.

The results of the different dictionaries show the sensitivity of DedupSearch to the keyword type. Although the number of search results for the entire high, med, and low dictionaries is similar, the number of result records generated during the search varies drastically. For example, there are 4% more keyword matches when searching for the Linux-high dictionary than for Linux-med, but 55% [120%] more records [tiny records] in the database. Thanks to the compact representation

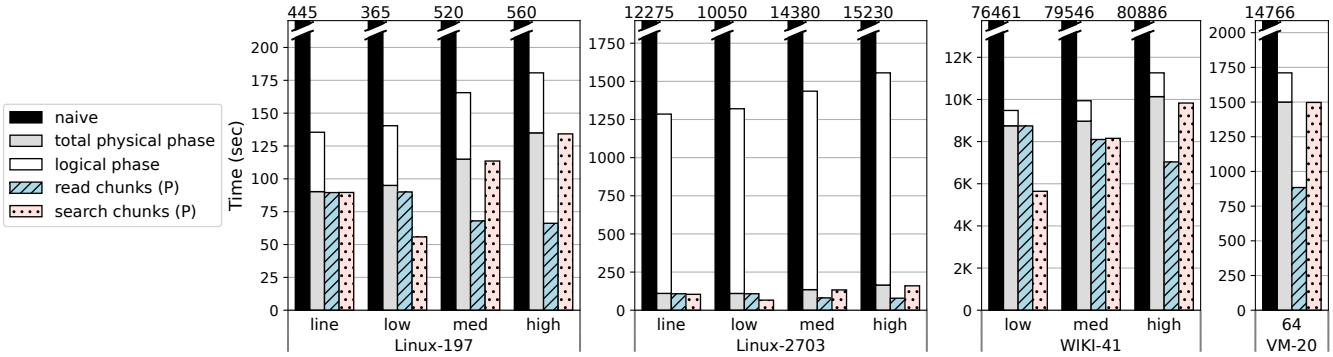


Figure 6: Breakdown of DedupSearch times of 128 words from all dictionaries and 8KB chunks.

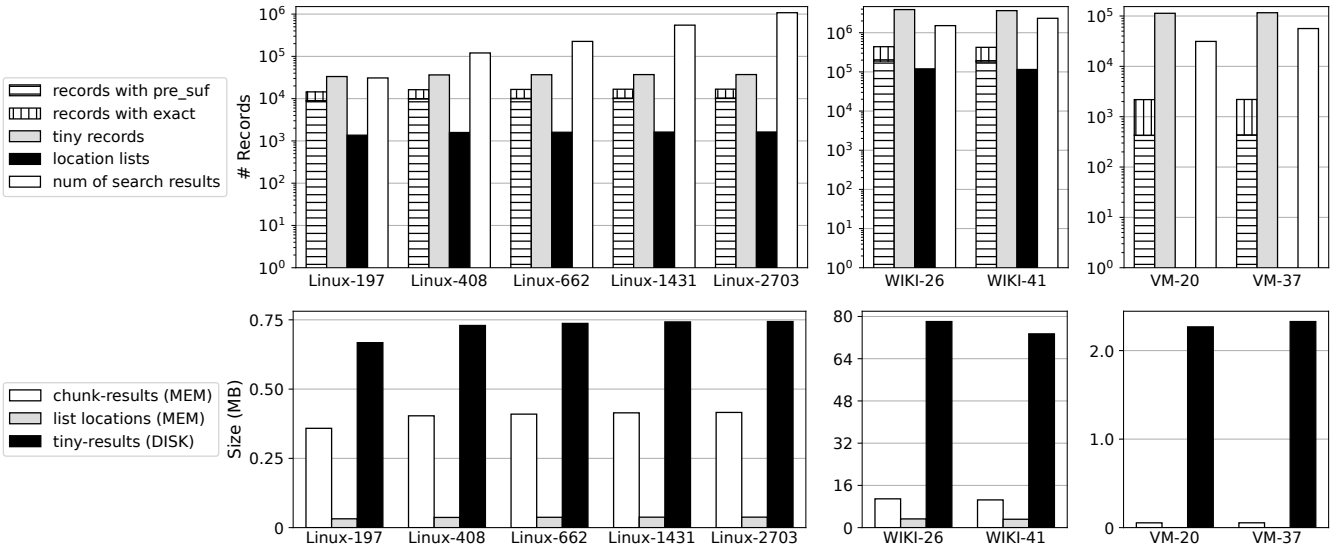


Figure 7: The number of search results and result objects during the search of a single keyword from the 'med' dictionary (top, note the log scale of the y-axis), and the corresponding database sizes (bottom).

of the tiny records (and their location on the disk), the database for Linux-high is only 16% larger than that of Linux-med, and its memory usage is also only 15% higher.

All the tiny-result databases in our experiments were small enough to fit in our server's memory. The largest tiny-result database, 3.6GBs, was created when searching for the Wiki-high dictionary on the Wikipedia-41 dataset with 4KB chunks. Nevertheless, we designed and implemented DedupSearch to avoid memory contention in much larger datasets.

Database accesses. Table 4 presents additional statistics of database usage and access during the search of keywords from the 'med' and '64' dictionaries (on the 8KB-chunk datasets). The top line for each dataset presents an average of four experiments, each with a different word from the dictionary. The bottom line presents results for searching the entire dictionary. Less than 0.2% of the keyword matches were split between chunks in the textual (Linux and Wikipedia) datasets. The percentage of split results was higher in the binary datasets because the keywords in the dictionary were considerably longer.

The number of accesses to the tiny-result index increases with the dataset's logical size and with the number of keywords. However, it is still several orders of magnitude lower than the number of records in the database: the tiny-result index is accessed only when the rest of the keyword is found in the chunk. The probability that the missing character is found in the adjacent chunk ('tiny hit') depends on the choice of keywords. For comparison, the percentage of successful substring matches out of all attempts is approximately 5% in the Linux datasets and 30% in the Wikipedia datasets. These differences are due to the different text types in the two datasets, and to some short (4-letter) keywords in the Wikipedia dataset.

Although the number of accesses to the tiny-result index can be as high as hundreds of thousands when searching large dictionaries, these numbers are orders of magnitude smaller than the random accesses that Naïve performs when fetching the data chunks in their logical order. Furthermore, repeated accesses to the index result in page-cache hits, as the operating system caches frequently accessed portions of the index. Thus, even though our on-disk index represents the worst-case performance of DedupSearch, we expect that moving it to memory would not add significant performance gains.

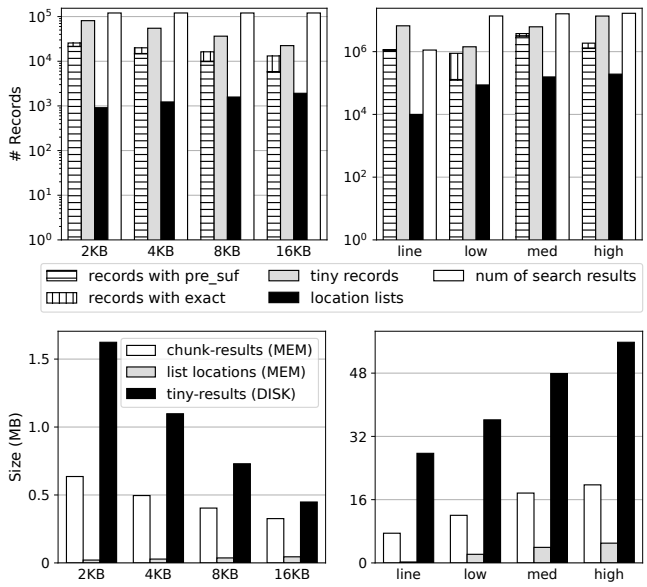


Figure 8: The number of search results, result records and the corresponding database sizes in Linux-408 during a search of one keyword (left) and entire dictionaries (right).

Dataset	# results (M)	% matches split	# tiny records (M)	# tiny accesses	tiny hit rate
Wiki-26	1.52	0.05	3.90	1167	0.10
	208.50	0.10	490.31	44,719	0.94
Wiki-41	2.34	0.05	3.67	1780	0.10
	321.07	0.09	459.96	69,094	0.94
Linux-197	0.03	0.19	0.03	59	0.08
	5.08	0.12	4.19	1,665	0.73
Linux-408	0.12	0.19	0.04	197	0.15
	16.08	0.11	4.57	5,986	0.71
Linux-662	0.23	0.19	0.04	360	0.16
	29.16	0.11	4.63	11,101	0.70
Linux-1431	0.55	0.18	0.04	855	0.16
	68.96	0.11	4.67	26,682	0.70
Linux-2703	1.08	0.18	0.04	1673	0.17
	134.65	0.11	4.68	52,391	0.69
VM-20	0.03	0.00	0.11	0	N/A
	4.02	1.61	14.62	0	N/A
VM-37	0.06	0.00	0.12	0	N/A
	7.24	1.61	14.96	0	N/A

Table 4: Percentage of keywords split between chunks and usage of the tiny-result index. The numbers are from searching one (top) and 128 (bottom) keywords from the 'med' and '64' dictionaries.

Dataset	Logical size	Physical size	Dedup ratio	Naïve time	DSearch time (logical)
Wiki-1	76	76	99.8%	616	620 (11)
LNX-1	1	0.80	80%	7.4	6.7 (0.6)
LNX-1-merge	0.82	0.78	95%	6.2	6.1 (0.1)
LNX-408	204	17	7.4%	926	231 (121)
LNX-408-merge	169	19	11.2%	768	203 (28)

Table 5: The size (in GB) and dedup ratio of the datasets created from a single archived version with 8KB chunks, and the time (in seconds) to search a single keyword from the 'med' dictionary.

6.3 DedupSearch overheads

Physical phase. To measure the time of the extra processing per chunk, we created five mini-backups with no duplicates of container-sized (4 MB) samples from Wikipedia. We read the data in advance and kept it in memory for the duration of the experiment to eliminate I/O delays. We measured the time of the processing thread of the physical phase (containing the keyword search algorithm and storing the records) and compared it to the processing time of Naïve. We performed a search with one and 128 keywords from the 'high' dictionary. With one word, DedupSearch and Naïve spent the same time processing the data. However, with 128 keywords, the processing thread of DedupSearch ran 32% longer than that of Naïve. The reason is that the number of records stored by DedupSearch is not proportional to the number of full matches, especially in the 'high' dictionary. With 128 keywords, there are hundreds of records, compared to dozens with one keyword. This experiment represents the worst-case for DedupSearch, with the highest number of partial matches and zero I/O cost. In practice, we expect such situations to be rare.

Logical phase. In addition to the datasets described in Table 2, we created three small datasets, each consisting of a single archived Linux/Wikipedia version. Table 5 shows the characteristics of these datasets. They exhibit the least amount of deduplication, allowing us to evaluate the overheads of DedupSearch in use-cases where its benefits are minimal. The table also shows the time spent by Naïve and by DedupSearch when searching a single keyword from the 'med' dictionary.

In the Wikipedia dataset, which exhibits minimal deduplication, DedupSearch is slower than Naïve by 0.8%. The reason is that Naïve emits its search results as soon as the chunks are processed, while DedupSearch requires the additional logical phase. DedupSearch reads and processes 20% and 0.02% less data, respectively (recall that data is read and processed in the granularity of containers and chunks, respectively).

In the Linux dataset, the physical size is 20% smaller than the logical size, and thus the physical phase of DedupSearch is shorter than Naïve's total time. The logical phase on this dataset, however, is 600 msecs, which are 9% of the total time of DedupSearch. The reason is the large number of files (64K) in the single Linux version. The logical phase parallelizes reading the file recipes from disk, fetching chunk results from their database, and collecting the full matches for the files. As the number of files increases, the overhead of context switching and synchronization between the threads increases.

To illustrate this effect, we include a merged dataset of the same Linux version, where the content of the entire archived version is concatenated into a single file. The physical size of Linux-1 and Linux-1-merge is similar and so is the time of the physical phase when searching them. The logical phase, however, is six times shorter, because it has to process only a single file recipe. We repeated this experiment with a larger number of versions: we created the Linux-408-merge dataset

by concatenating each of the versions in the Linux-408 dataset into a single file. This dataset contains 408 files, compared to a total of 15M files in Linux-408. The logical phase when searching the merged dataset is 4.3× faster. This effect also explains the long times of the logical phase when searching the Linux datasets (see Figure 3). The number of files in each Linux dataset increases from 4.3M in Linux-197 to 133M in Linux-2703. We conclude that the overheads of DedupSearch are low, even when the deduplication is very low. When deduplication ratios are high, these overheads become negligible as DedupSearch is faster than Naive by orders of magnitude.

7 Discussion

Extended search options. DedupSearch lends itself to several extensions that can enhance the functionality of the search. The first is the use of "wildcards"—special characters that represent entire character groups, such as numbers, punctuation marks, etc. Grep-style '*' wildcards can be supported by creating a dictionary that includes all the precise (non-*) substrings in the query. The logical phase would have to ensure that they all appear in the file in the correct order.

It would be more challenging to support keywords that span more than two chunks, since our prefix/suffix approach is insufficient. We would have to also identify chunks whose entire content constitutes a substring of the keyword, which means attempting to match the chunk content starting at all possible offsets within the keyword. Supporting regular expressions is similarly challenging, because the matched expression might span more than two chunks.

Approximate search. Some applications of keyword search only require the list of files containing the keyword, without the offset of all occurrences within the file, so the logical phase can stop processing a file's recipe as soon as a keyword is found. This eliminates the need for the location-list records. Alternatively, a best-effort search could focus on exact matches within a chunk for a faster, though imperfect search.

Additional applications. Dividing the search into physical and logical phases can potentially accelerate keyword search in highly fragmented or log-structured file systems as well as copy-on-write snapshots where logically adjacent data blocks are not necessarily physically adjacent.

The benefit of DedupSearch might be smaller when a large portion of the chunks can be marked irrelevant a priori by Naïve. Examples include binary data in textual search, file-system metadata, or chunks belonging to files that are excluded from the search for various reasons. Additional aspects that may affect the performance of DedupSearch and its advantage over Naïve include the underlying storage media (i.e., faster SSD), and parallel processing of chunks and file recipes. We leave these for future work.

8 Related Work

Deduplication is a maturing field, and we direct readers to survey papers for general background material [35, 46]. Our search technique follows on previous work that processed post-deduplication data sequentially along with an analysis phase on the file recipes, which has been applied to garbage collection and data migration [16, 17, 33]. We leverage this basic concept by processing the post-deduplication data with large, sequential I/Os instead of a logical scan of the file system with random I/O. Thus far, we have not found previous research that optimized string search for deduplicated storage.

String matching. String matching is a classical problem with a rich family of solutions that are used in a variety of areas. The longstanding character-based exact string matching algorithms are still at the heart of modern search tools. These include the Boyer-Moore algorithms [14], hashing-based algorithms such as Rabin-Karp [27], and suffix-automata based methods such as Knuth-Morris-Pratt [28] and Aho-Corasick [11]. ACCH [15] accelerates Aho-Corasick on Compressed HTTP Traffic by recording partial matches in referenced substrings. GPU-based string matching is used in network intrusion detection systems [42, 47].

Indexing. Offline algorithms use indexing to achieve sub-linear search time. Indexing methods include suffix-trees [24], metric trees [12] and *n*-gram methods [34], and the rank and select structure for compressed indexing [20]. Indeed, many systems scan the data in advance to map terms to their locations. Examples include Elasticsearch [2], Splunk [8], and CLP [38] for log searches and Apache Solr [7] for full-text search. An index is useful when queries are frequent and latency must be low. Its downside is that it precludes searching keywords that are not indexed, such as full sentences or arbitrary binary strings. Moreover, its size might become a substantial fraction of the dataset size: 5-10% in practice [6, 31]. Our approach is thus more appropriate when queries are infrequent and moderate latency is acceptable such as in legal discovery [37, 45].

DedupSearch can be viewed as a form of **near-storage processing**, where the storage system supports certain computations to reduce I/O traffic and memory usage. For example, YourSQL [26] and REGISTOR [36] offload functions to the SSD. BAD-FS [13] and Quiver [30] coordinate batch workloads or jobs to minimize I/O in large-scale systems. DedupSearch shares their underlying principle of fetching and/or processing data once for use in several contexts.

9 Conclusions

String search is a widely-used storage function and can be redesigned to leverage the properties of deduplicated storage. We present a two-phase search algorithm. The physical phase scans the storage space and stores matches per chunk. Most of the results are stored on the disk while only the popular are in memory. The logical phase goes over all file recipes and uses the chunk results to collect matches, including matches that span logically consecutive chunks. Our evaluation demonstrates significant savings of time and reads in DedupSearch in comparison to the Naïve search, thanks to the physical scan that reads duplicated chunks only once.

Acknowledgments

We thank the reviewers and our shepherd, Andrea Arpaci-Dusseau, for their feedback and suggestions, Amnon Hanuhov for help with the Aho-Corasick implementation, and Dita Jacobovitz for help with the VM datasets. This research was supported by the Israel Science Foundation (grant No. 807/20).

References

[1] DedupSearch implementation. https://github.com/NadavElias/DedupSearch.

[2] Elasticsearch: The heart of the free and open Elastic Stack. //https://www.elastic.co/elasticsearch/.

[3] libstdc++. https://gcc.gnu.org/onlinedocs/gcc-7.5.0/libstdc++/api/a00293_source.html#l01188.

[4] Linux Kernel Archives. https://mirrors.edge.kernel.org/pub/linux/kernel/.

[5] Oracle Berkeley DB. https://www.oracle.com/database/technologies/related/berkeleydb.html.

[6] Search indexing in Windows 10: FAQ. https://support.microsoft.com/en-us/windows/search-indexing-in-windows-10-faq-da061c83-af6b-095c-0f7a-4dfecda4d15a.

[7] Solr. https://solr.apache.org/.

[8] splunk. https://www.splunk.com/.

[9] Wikimedia data dump torrents. https://meta.wikimedia.org/wiki/Data_dump_torrents.

[10] Wikimedia downloads. https://dumps.wikimedia.org/enwiki/.

[11] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[12] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No.98EX207)*, pages 14–22, 1998.

[13] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control a batch-aware distributed file system. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[14] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[15] Anat Bremler-Barr and Yaron Koral. Accelerating multipattern matching on compressed HTTP traffic. *IEEE/ACM Transactions on Networking*, 20(3):970–983, 2012.

[16] 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.

[17] 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.

[18] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication—large scale study and system design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[19] Nadav Elias. Keyword search in deduplicated storage systems. Technical report, Computer science department, Technion, 2021.

[20] Antonio Fariòa, Susana Ladra, Oscar Pedreira, and Ángeles S. Places. Rank and select for succinct data structures. *Electron. Notes Theor. Comput. Sci.*, 236:131–145, April 2009.

[21] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[22] Christopher Gilbert. Aho-Corasick implementation (C++). https://github.com/cjgdev/aho_corasick.

[23] Gintautas Grigas and Anita Juskeviciene. Letter frequency analysis of languages using latin alphabet. *International Linguistics Research*, 1:p18, 03 2018.

[24] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, USA, 1997.

[25] 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.

[26] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. YourSQL: A high-performance database system leveraging in-storage computing. *Proc. VLDB Endow.*, 9(12):924–935, August 2016.

[27] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[28] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, March 1977.

[29] Geoff Kuenning. How does a computer virus scan work? *Scientific American*, January 2002.

[30] 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)*, 2020.

[31] Sergey Melink, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a distributed full-text index for the web. *ACM Transactions on Information Systems (TOIS)*, 19(3):217–241, 2001.

[32] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[33] 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.

[34] Gonzalo Navarro, Ricardo Baeza-Yates, Erkki Sutinen, and Jorma Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng Bull*, 24:19–27, 11 2001.

[35] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):1–30, 2014.

[36] Shuyi Pei, Jing Yang, and Qing Yang. REGISTOR: A platform for unstructured data processing inside SSD storage. *ACM Trans. Storage*, 15(1), March 2019.

[37] Martin H Redish. Electronic discovery and the litigation matrix. *Duke Law Journal*, 51:561, 2001.

[38] Kirk Rodrigues, Yu Luo, and Ding Yuan. CLP: Efficient and scalable search on compressed text logs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[39] Margo I. Seltzer and Ozan Yigit. A new hashing package for UNIX. In *USENIX Winter*, 1991.

[40] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnala. 99 deduplication problems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[41] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[42] Giorgos Vasiliadis and Sotiris Ioannidis. *GrAVity: A Massively Parallel Antivirus Engine*, pages 79–96. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

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

[44] Jau-Hwang Wang, Peter S Deng, Yi-Shen Fan, Li-Jing Jaw, and Yu-Ching Liu. Virus detection using data mining techinques. In *IEEE 37th Annual 2003 International Carnahan Conference on Security Technology, 2003. Proceedings.*, pages 71–76. IEEE, 2003.

[45] Kenneth J Withers. Computer-based discovery in federal civil litigation. *Fed. Cts. Law Rev.*, 1:65, 2006.

[46] 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.

[47] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *2006 Symposium on Architecture For Networking And Communications Systems*, pages 93–102, 2006.

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

DeepSketch: A New Machine Learning-Based Reference Search Technique for Post-Deduplication Delta Compression

Jisung Park[1]* Jeonggyun Kim[2]* Yeseong Kim[2] Sungjin Lee[2] Onur Mutlu[1]
[1]*ETH Zürich* [2]*DGIST*

Abstract

Data reduction in storage systems is becoming increasingly important as an effective solution to minimize the management cost of a data center. To maximize data-reduction efficiency, existing post-deduplication delta-compression techniques perform delta compression along with traditional data deduplication and lossless compression. Unfortunately, we observe that existing techniques achieve significantly lower data-reduction ratios than the optimal due to their limited accuracy in identifying similar data blocks.

In this paper, we propose *DeepSketch*, a new reference search technique for post-deduplication delta compression that leverages the learning-to-hash method to achieve higher accuracy in reference search for delta compression, thereby improving data-reduction efficiency. DeepSketch uses a deep neural network to extract a data block's *sketch*, *i.e.,* to create an approximate data signature of the block that can preserve similarity with other blocks. Our evaluation using eleven real-world workloads shows that DeepSketch improves the data-reduction ratio by up to 33% (21% on average) over a state-of-the-art post-deduplication delta-compression technique.

1 Introduction

As modern data centers generate a tremendous volume of new data every day, it becomes critical for sustainability to store such large amounts of data in an economical way. Employing a data-reduction technique is one of the effective solutions to cut down the management cost of a data center. A data-reduction technique reduces the amount of data physically stored in storage media by reducing data redundancy, which allows a data center to handle the same amount of data with fewer or smaller resources (*e.g.,* storage devices and servers).

Many prior works have proposed various data-reduction techniques based on data compression [9,30,46,51,52,58] and data deduplication [12,21,22,26,36,49,59,62,67,76,88]. Data compression encodes a data block using lossless-compression algorithms so that a smaller number of bits can represent the data block. Data deduplication prevents a data block from being written if there already exists an *identical* data block (*i.e.,* a block that contains exactly the same data) in the storage system. To achieve a high data-reduction ratio (*i.e., Original Data Size / Reduced Data Size*), some studies [45, 55] integrate the two techniques in a manner that first applies data deduplication for incoming (*i.e.,* to-be-stored) blocks and performs lossless compression on non-deduplicated blocks.

Delta compression [3, 8, 64, 75, 81, 82, 86] has recently received increasing attention as a complementary method to overcome the limitations of data compression and data deduplication. It compares the data block to compress with a *reference* data block and extracts only different bit patterns between the two blocks, which are then encoded using lossless compression. The more similar the data block and the reference (*i.e.,* the smaller the delta between the data and the reference), the higher the data-reduction ratio. By leveraging the similarity between two blocks, delta compression can achieve a high data-reduction ratio even for non-duplicate data (which cannot benefit from data deduplication) and high-entropy data (which lossless compression cannot efficiently handle). Several prior works [64, 75, 82, 86] demonstrate that *post-deduplication delta compression*, which performs deduplication, delta compression, and lossless compression in order, can significantly improve the data-reduction ratio over simple integration of deduplication and lossless compression.

A key challenge for post-deduplication delta-compression techniques is how to find a good reference block that provides a high data-reduction ratio. The most intuitive approach is to scan all the data blocks stored in the storage system and use the one that provides the highest data-reduction ratio as the reference for the incoming block. Unfortunately, doing so is practically infeasible due to its prohibitive performance overhead. To address this, prior works [64, 75, 82, 86] use *locality-sensitive hash (LSH)* functions [7, 34] to generate similar data signatures for data blocks with similar bit patterns, which is called *data sketching*. Data sketching enables quick reference search across a large-scale storage system by comparing only the signatures (*i.e.,* sketches) of data blocks.

In this work, we observe that existing post-deduplication delta-compression techniques [75, 86] achieve significantly lower data-reduction ratios than the optimal due to the high *false-negative rate (FNR)* of LSH-based reference search. Our analysis using six real-world workloads shows that, although a state-of-the-art reference search technique [86] is effective at identifying a *very similar* reference block (which thus provides a very high data-reduction ratio) for an incoming block, it also fails to find *any* reference block for a large number of incoming blocks (up to 75.5%) that actually have at least one good reference block within the storage system. Tuning the used LSH function may be able to reduce the high FNR in reference search, but it would require significant human effort to identify the best settings for each workload.

Our goal is to improve the space efficiency of a storage system by increasing the accuracy of reference search in post-deduplcation delta compression, thereby reducing the gap between existing data-reduction techniques and the optimal.[1] To this end, we propose *DeepSketch*, a

*J. Park and J. Kim are co-primary authors.

[1] In this work, we focus on data reduction rather than other optimizations (*e.g.,* mitigation of performance/memory overheads), targeting systems where space efficiency is the highest priority (*e.g.,* archival or backup systems).

new machine learning (ML)-based data sketching mechanism specialized for reference search in delta compression. **Our key idea** is to use the *learning-to-hash* method [43, 80] to automatically generate similar data signatures for any two data blocks that would provide a high data-reduction ratio when delta-compressed relative to each other.

For each incoming data block, DeepSketch generates the block's sketch using a deep neural network (DNN) model. It performs DNN inference with the target data block as an input of the DNN and uses the resulting activation values in the DNN's last hidden layer as the data block's sketch. We envision that DeepSketch's DNN is *pre-trained* before building or updating a DeepSketch-enabled system, using data sets collected from other existing systems that store similar (or the same) types of data.

While many prior works [10, 11, 48, 50, 70, 89] demonstrate the high effectiveness of the learning-to-hash method in various nearest-neighbor search applications (*e.g.,* image recognition and classification), applying the learning-to-hash method to the reference search problem in post-deduplication delta compression is not straightforward. A key problem is that, unlike existing ML-based applications that deal with specific known data types (*e.g.,* images and voices), DeepSketch needs to process *general binary data*, which introduces two key challenges. First, there is no well-defined labeled data or semantic information (*e.g.,* cats, dogs, and monkeys in image classification) within our target data sets. Second, possible bit patterns of a data block have an extremely high dimensional space (*e.g.,* $2^{4,096\times 8}$ unique bit patterns for a 4-KiB data block). Due to the high dimensionality of the target data set, it is difficult to collect large enough data to train the DNN with high inference accuracy using known training methods.

To address the above challenges, we develop a new method to train the DNN of DeepSketch, which generates hash values suitable for reference search in post-deduplication delta compression. We extend the traditional unsupervised learning approach [29] in three ways. First, based on the k-means clustering algorithm [53], we design a new clustering method, called *dynamic k-means clustering (DK-Clustering)*, which effectively classifies high-dimensional data without any knowledge of the number of clusters. Second, after clustering, we ensure each cluster to have a sufficient number of data blocks by adding data blocks slightly and randomly modified from each cluster's representative block. Doing so prevents DNN training from being biased towards some specific data patterns that occur frequently. Third, we perform two-stage DNN training to enable DeepSketch to generate a data block's hash value. We first train a DNN to classify data blocks into the clusters formed by DK-Clustering and then transfer the knowledge of the trained DNN to build the learning-to-hash network that generates the hash values (*i.e.,* sketches) of data blocks.

We integrate our DeepSketch engine into a state-of-the-art post-deduplication delta-compression technique [86]. Unlike existing techniques that aim to find a reference block whose sketch *exactly* matches that of the incoming block, we exploit a state-of-the-art *approximate* nearest-neighbor search algorithm [16]. Doing so allows DeepSketch to tolerate small differences within data sketches (i.e., it can identify similar blocks even when the blocks' sketches are different), thereby increasing the chance of delta compression for an incoming data block. Our evaluation using eleven real-world workloads shows that DeepSketch improves the data-reduction ratio by up to 33% (21% on average) over the state-of-the-art baseline.

This paper makes the following key contributions:

- We propose DeepSketch, the first machine learning-based reference search technique for post-deduplication delta compression. We demonstrate that the learning-to-hash method can be an effective solution to generate delta-compression-aware data signatures for general binary data.
- We introduce a new training method that allows DeepSketch to learn delta-compression-aware data representation for an extremely high-dimensional data set.
- We integrate DeepSketch into the state-of-the-art post-deduplication delta-compression technique [86]. Evaluation results using eleven real-world workloads show that DeepSketch improves the data-reduction ratio by up to 33% (21% on average) compared to the state-of-the-art baseline.

2 Background

We provide brief background on data-reduction techniques in storage systems necessary to understand the rest of the paper.

2.1 Data Reduction in Storage Systems

There are three major data-reduction approaches: 1) data deduplication, 2) lossless compression, and 3) delta compression.

Data Deduplication. Data deduplication [12, 21, 22, 26, 36, 49, 59, 62, 67, 76, 88] reduces the amount of data physically written to storage devices by eliminating duplicate data in the storage system. In data deduplication, an incoming data block is not physically written if it has the same data content as a data block previously stored in the storage system. Instead, the storage system maintains a table that stores mapping information between such a deduplicated block and the previously-stored block with the same content (called *reference*), so that future reads to any deduplicated blocks can be serviced from their reference. This mechanism allows data deduplication to store only a single copy of any block-granularity unique data content in the storage system.

To quickly identify an incoming block's reference, deduplication uses a strong hash function (*e.g.,* SHA1 [78] or MD5 [69]) to generate a data block's *unique* signature, commonly called a *fingerprint*. Given two blocks, deduplication determines whether or not they have the same content, by comparing only the two blocks' fingerprints. To avoid any data loss due to hash collision, it is common practice for deduplication to use a strong hash function to generate fingerprints whose collision rate is lower than the uncorrectable bit-error rate (UBER) requirement of a disk (*e.g.,* $< 10^{-15}$ to 10^{-16} [17, 25, 26, 67]).

Lossless Compression. Data compression [31, 74, 90] is a fundamental method to reduce the size of data in computing systems. Given a data block, it encodes the block's content to

be represented by a smaller number of bits in a manner that replaces repetitive bit patterns with smaller metadata or symbols. Doing so results in an increase in the *entropy* [74] of the compressed data. For a data block with low entropy (*i.e.,* the block contains many repeated bit patterns), lossless compression can achieve a high *data-reduction ratio* (*i.e., Original Data Size / Compressed Data Size*).

Delta Compression. Delta compression [3,8,64,75,81,82,86] eliminates redundant bit patterns that coexist in two different blocks. It stores only either of the two blocks and the difference (*i.e., delta*) between the two blocks. Leveraging the *similarity* of two different data blocks enables delta compression to achieve higher data reduction over 1) deduplication, which removes only *identical* data blocks, and 2) lossless compression, which eliminates redundancy only *within a block* and does *not* work well with high-entropy data. For this reason, delta compression has gained increasing attention in recent studies [64, 75, 82, 86] as a complementary method bridging deduplication and lossless compression.

A key challenge for delta compression in large-capacity storage systems is how to find a good reference block that provides a high data-reduction ratio for each incoming block. Designing a reference search technique for delta compression is similar to solving a nearest-neighbor search problem, as its goal is to find the most similar data block (which does not have to be an *exact* match) within a large data set for a given incoming block. The most widely-used approach is to use locality sensitive hashing (LSH) [7, 34] to generate a *sketch* of a block [64, 75, 82, 86], which is a *more approximate* signature than the block's fingerprint (used in deduplication for exact-match searching). An LSH function $L(d_i)$ is designed to hash data d_i, such that the more similar the given data d_1 and d_2, the lower the bit-pattern difference between $L(d_1)$ and $L(d_2)$. LSH-based data sketching enables quick reference search by comparing only the sketches of data blocks.

2.2 Combined Data-Reduction Technique

For systems where storage efficiency is the paramount requirement, prior works propose to combine the three major data-reduction techniques, called *post-deduplication delta compression*. Figure 1 depicts the overall architecture of a storage system that adopts post-deduplication delta compression [75, 86] to perform deduplication (❶–❸), delta compression (❹–❼), and lossless compression (❽) in order. A data-reduction module (DRM), which can be implemented as an intermediate layer between a file system and storage devices, performs post-deduplication delta compression for a write request to reduce its size. For a read request, it looks for the location of the corresponding compressed data in storage devices and returns the decompressed data. To this end, the DRM maintains a fingerprint (FP) store and a sketch (SK) store for quick reference search for deduplication and delta compression, respectively, along with a reference (Ref.) table to store the mapping information between each deduplicated or delta-compressed block and its reference block.

For each incoming data block, the DRM ❶ first checks

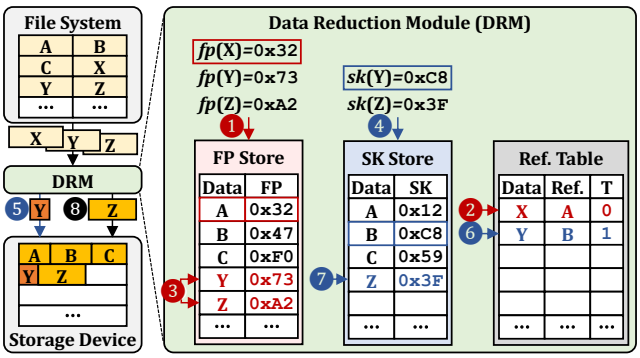


Figure 1: Overview of post-deduplication delta compression.

if the storage system already contains a data block with the same content by referring to the FP store. If the incoming block's fingerprint matches one in the FP store (*e.g.,* block X matching block A in Figure 1), the DRM skips writing the block to the storage device and ❷ just updates its mapping information in the reference table in order to redirect future reads for the incoming block to the matching reference block. To use non-deduplicated blocks (*e.g.,* blocks Y and Z) as potential reference data for deduplication in the future, the DRM ❸ writes their fingerprints into the FP store.

If there is no matching fingerprint in the FP store, the DRM ❹ searches for matching sketches in the SK store to find a reference block for delta compression. When it finds a reference block (*e.g.,* block B for block Y), the DRM ❺ performs delta compression with the reference and stores the compressed data. There is a possibility of having multiple matching references in the SK store (see Section 3.1). In such a case, the DRM usually selects the first-found candidate (called *first-fit* selection) [75, 86]. The DRM then ❻ updates the reference table to map the incoming block to the reference block so that it can decompress the delta-compressed data using the reference block for future read requests. If no matching sketch is found in the SK store (*e.g.,* block Z), the DRM ❼ adds the incoming block's sketch into the SK store to use the incoming block as a potential reference block for delta compression in the future. Finally, the DRM ❽ compresses the block with a lossless compression algorithm and stores the result.

3 Motivation

In this section, we discuss 1) the limitations of existing LSH-based post-deduplication delta-compression techniques [75, 86] and 2) the potential of the learning-to-hash method [43,80] for more accurate reference search in delta compression.

3.1 Limitations of LSH-Based Sketching

As explained in Section 2.1, LSH-based data sketching enables quick search for a reference block (*i.e.,* reference search) in post-deduplication delta compression. Figure 2 describes the high-level idea of state-of-the-art LSH-based sketching schemes [75, 86], which we call *super-feature data sketching (SFSketch)*. SFSketch generates a data block's sketch using m *features* extracted from the block (*e.g.,* $m = 12$ in Figure 2). To extract a feature F_i(A) of block A ($0 \leq i \leq m-1$), as shown in

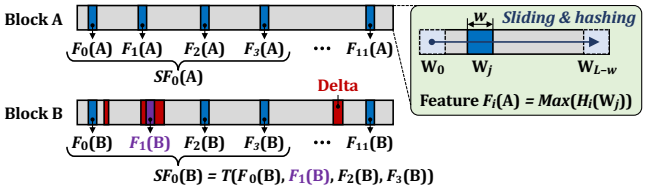


Figure 2: An example of LSH-based sketching.

Figure 2 (right), SFSketch calculates the hash value $H_i(W_j)$ of each sliding window W_j, where j is the starting byte position of the window in the block. Given a block size of L and a window size of w, $(L-w+1)$ hash values are calculated in total, and the maximum hash value $Max(H_i(W_j))$ is selected as feature F_i(A). SFSketch repeats this process to extract m features using a different hash function for each feature (*i.e.,* H_i for F_i) and then builds N super-features (SFs) by transposing the m features (*e.g.,* given $m = 12$ and $N = 3$, SF_k(A) = $T(F_{4k}$(A), F_{4k+1}(A), F_{4k+2}(A), F_{4k+3}(A)), where $0 \leq k \leq 2$).

Using multiple SFs as the sketch of a data block enables SFSketch to tune the accuracy of reference search by changing the *matching criteria* (i.e., criteria for judging the similarity of given two blocks). Consider the example of Figure 2 where block A's data content is almost the same as block B, except for the red regions marked as *Delta*. Suppose that, due to the small differences between blocks A and B, every hash function H_i other than H_1 has the same maximum value $Max(H_i(W_j))$ for blocks A and B, *i.e.,* every feature F_i except F_1 is identical between blocks A and B. In such a case, SF_0(A) does not match SF_0(B) ($\because F_1(A) \neq F_1(B)$), while all the other SFs are identical between blocks A and B. The two blocks can be considered either similar or not depending on the matching criteria; SFSketch may either decide that the two blocks are dissimilar because there exists a different SF or judge that block A resembles block B since their other two SFs (*i.e.,* SF_1 and SF_2) match each other. There are many possible matching criteria, but to maximize the data-reduction ratio, existing SFSketch-based techniques [75, 86] consider that two blocks are similar if there exists at least one matching SF.

While existing SFSketch-based delta-compression techniques provide significant improvement in data reduction compared to a simple combination of deduplication and lossless compression [75, 86], we observe that SFSketch-based reference search often fails to identify a good reference block that can provide a high data-reduction ratio for an incoming block. To show this, we compare a state-of-the-art SFSketch-based reference search technique [86] to *brute-force* search that performs delta compression of an incoming block with *every* stored block and selects the stored block that provides the highest data-reduction ratio as the incoming block's reference.[2] For our evaluation, we use 4,090,975 4-KiB data blocks collected from six different workloads in real systems (see Section 5.1 for our evaluation methodology and workloads).

[2]While brute-force search guarantees the highest data-reduction ratio for a workload, it is infeasible to use due to its prohibitively high performance overhead. For example, in our evaluation environments (see Section 5.1 for more detail), brute-force search takes more than 300 hours for the Install trace that writes a total of 8.83-GB data to the storage system.

We use two major metrics to evaluate the accuracy of SFSketch compared to brute-force search: 1) *false-negative rate (FNR)*, the probability of identifying no reference block for an incoming data block even though brute-force search can find one, and 2) *false-positive rate (FPR)*, the probability of identifying a reference block *different* from what brute-force search finds. For FN cases, SFSketch compresses the data block using the LZ4 algorithm [15] because there is no reference block. For FP cases, SFSketch uses the Xdelta algorithm [56, 57] to perform delta compression of the block with the reference block that it identifies. For both cases, we measure the average data-reduction ratio (DRR) and compare it with that of brute-force search. Table 1 shows FNR, FPR, and DRR for FN/FP cases of the SFSketch-based reference search. DRR is normalized to that of brute-force search.

Table 1: Accuracy of LSH-based reference search [86].

Workload		PC	Install	Update	Synth	Sensor	Web	Avg.
FNR		35.3%	51.8%	56.3%	75.5%	48.1%	5.5%	35.7%
FPR		21.1%	15.8%	11.3%	14.1%	47.3%	60.6%	23.1%
DRR	FN cases	0.474	0.488	0.578	0.639	0.567	0.539	0.562
	FP cases	0.621	0.608	0.644	0.683	0.798	0.674	0.669

We make three observations from Table 1. First, the existing SFSketch-based technique suffers from high FNR (up to 75.5% and 35.7% on average), failing to find *any* reference block for many incoming blocks that actually have one or more reference blocks. Except for Web, SFSketch's FNR is higher than 35% for every workload. For FN cases, each data block is compressed by the LZ4 algorithm, and thus its DRR is considerably lower compared to when the block is delta-compressed with the reference block identified by brute-force search. As shown in Table 1, the normalized DRR in FN cases is 0.562 on average, showing that SFSketch provides 43.8% lower data reduction compared to the optimal for the FN cases (*i.e.,* for 35.7% of the entire data blocks on average).

Second, the SFSketch-based reference search frequently chooses a sub-optimal reference in some workloads, *e.g.,* Sensor and Web, which have a FPR of 47.3% and 60.6%, respectively. The sub-optimal selection of reference blocks results in lower data-reduction ratios over brute-force search. As shown in the last row in Table 1, the normalized DRR in FP cases is 0.669 on average, which means that SFSketch provides 33.1% lower data reduction compared to the optimal for the FP cases (*i.e.,* for 23.1% of the entire data blocks).

Third, FN cases are more common and have more negative impact on the DRR than FP cases. On average, FN cases occur for 35.7% of the incoming blocks, whereas FP cases occur for 23.1%. When a FN case happens, the data-reduction ratio using LZ4 is lower than when an FP case happens, which still uses delta compression albeit with a sub-optimal reference block; on average, the normalized DRR in FP cases (0.669) is 19% higher than that in FN cases (0.562).

The limited accuracy of SFSketch mainly stems from its inherent property; SFSketch is highly optimized to identify *only very similar* data. Considering the SF-based sketching process explained in Figure 2, it is highly unlikely that two blocks have at least one matching SF, unless they are very

similar. This property enables SFSketch to provide a high data-reduction ratio even when it selects a sub-optimal reference block for an incoming block (*i.e.,* for FP cases). However, it also causes SFSketch to frequently fail to find a *sufficiently good* reference block that is not very similar to the incoming block but is still beneficial for improving the data-reduction ratio.

It is challenging to optimize existing SF-based sketch algorithms to increase both FPR and FNR at the same time. The accuracy of SFSketch highly depends on its settings such as the number of features (m), the number of super features (N), the sliding window size (w), and the matching criteria. For example, under a matching criterion where two blocks are considered similar if they have at least one common SF, increasing the number of SFs (*i.e., N*) for each data block would reduce overall FNR. However, at the same time, it might increase FPR and reduce data-reduction ratios in FP cases because more dissimilar blocks could be chosen as reference blocks. Moreover, as shown in Table 1, the FNR/FPR trend of SFSketch-based search greatly varies across workloads, which makes it even more difficult to find the best configuration on average as well as on a per-workload basis. Instead, we investigate applicability of deep-learning algorithms for data sketching in delta compression, which can reduce the human effort for developing a new sketching scheme or fine-tuning existing techniques for different workloads.

3.2 Learning-to-Hash Method

The learning-to-hash method [43, 80] is a promising machine learning (ML)-based approach for the nearest-neighbor search problem. It trains a neural network (NN) to generate a hash value for a given input data block such that any two similar data blocks have similar hash values. Many prior works [10, 11, 23, 47, 48, 50, 89] demonstrate the high effectiveness of the learning-to-hash method at nearest-neighbor search in various applications, such as image recognition and classification.

Figure 3 depicts how a representative learning-to-hash scheme [50] generates a binary hash value of an image for content-based image retrieval. It extracts the hash value of an input image from the last hidden layer (*e.g.,* HL_N in Figure 3) of a NN that is trained to classify the input image to one of C possible classes. During inference, the activations in the last hidden layer of two similar images are likely to be largely the same if the two images belong to the same class. Therefore, their hash values should also be similar because they are directly extracted from the last layer by translating the output of each activation into a binary ('1' or '0').

The learning-to-hash method has potential to be used for

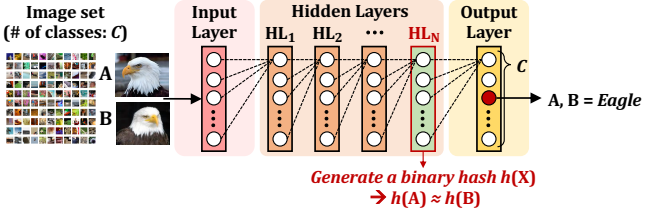


Figure 3: Learning-to-hash for image retrieval.

reference block search in post-deduplication delta compression, another nearest-neighbor search problem. In particular, rapid advances in machine learning have enabled learning-based algorithms to outperform a human or human-made heuristics in various problems, such as facial recognition [65], speech recognition [83, 84], image classification [50], and system optimizations (*e.g.,* branch prediction [37, 38], memory scheduling [35], and prefetching [4]). These successful examples motivate us to develop a learning-based sketching scheme that could be more effective than existing LSH-based sketching schemes relying on human-designed heuristics and metrics.

4 DeepSketch

The key idea of DeepSketch is to use the learning-to-hash method to generate similar data signatures (*i.e.,* sketches) for any two data blocks that would provide a high data-reduction ratio when delta-compressed relative to each other. The main difference of DeepSketch over the existing post-deduplication delta-compression approach [75, 86] (described in Figure 1) is that DeepSketch generates a data block's sketch by using a deep neural network (DNN) model, instead of using an LSH function (*e.g., sk*(X) in Figure 1). For each incoming data block, DeepSketch performs DNN inference with the block as input and uses the resulting activation values in the DNN's last hidden layer as the block's sketch.

We envision that DeepSketch's DNN is *pre-trained* offline in other machines with more powerful computation resources before building or updating a storage machine. For example, to adopt DeepSketch in a new storage server of a data center, one can train DeepSketch's DNN using randomly-selected data blocks from existing storage servers that contain specific types of data (*e.g.,* databases, images, web caching, etc.) expected to be stored in the new storage server. Similarly, to further enhance the accuracy of DeepSketch, one can retrain DeepSketch's DNN and use the enhanced DNN to build new storage servers or reorganize existing ones.

While the high-level idea may sound simple, applying the learning-to-hash method for reference search in post-deduplication delta compression is not straightforward. This is because DeepSketch needs to deal with *general binary data*, which introduces the following two technical challenges:

Challenge 1. Lack of Semantic Information. The target data set of DeepSketch can contain *any* data from various applications, such as text, images, binary executable files, and so on. Compared to existing learning-to-hash approaches focusing on pre-categorized data (*e.g.,* Imagenet [20], CIFAR [42], and MNIST [44]), DeepSketch needs to process a much wider range of data without any well-defined semantic information about the delta-compressibility of data blocks.

Challenge 2. High Dimensional Space. The lack of semantic information in DeepSketch's target data sets leads us to perform *unsupervised learning* that is used for drawing inferences from a data set without labeled information. The most common unsupervised learning approach is to use a clustering algorithm that groups the target data set according to a certain

measure, *e.g.,* similarity of bit patterns in our case. However, possible bit patterns of a data block for DeepSketch have extremely high dimensional space (*e.g.,* $2^{4,096\times8}$ assuming a 4-KiB data block), which makes it difficult to 1) set a proper number of final clusters and 2) collect a data set large enough to cover all possible data patterns for a clustering algorithm.

To address the above challenges, we develop a new clustering algorithm, called *dynamic k-means clustering (DK-Clustering)*, which groups data blocks that would provide high delta-compression ratios when delta-compressed relative to each other (Section 4.1). To cope with potential groups of data blocks that are missing in the collected data sets, after clustering, we generate new data blocks by randomly and slightly modifying existing blocks. We then generalize the understanding of the similarity relationship between data blocks using the learning-to-hash method, so that DeepSketch can generate a learning-based data sketch for any given block (Section 4.2). With the sketch values computed by the learning-to-hash model, DeepSketch identifies the most similar reference block to each incoming block based on an approximate nearest-neighbor search technique (Section 4.3). We also perform hyper-parameter exploration for our DNN model to find the appropriate sketch size (Section 4.4).

4.1 Dynamic K-Means Clustering

DK-Clustering is based on the existing k-means clustering algorithm [53] that partitions a data set into a given number (*i.e., k*) of clusters such that each data element belongs to the cluster with the nearest mean value. Unfortunately, in our case, the value of *k* is initially unknown. Figuring out the most suitable value for *k* by exploring a given data set is time-consuming, considering the extremely high dimensionality of the data set that DeepSketch deals with.

The hierarchical clustering algorithm [39] is known to be suitable for such data sets, but it introduces prohibitive computation and memory overheads for a large-size data set. To be specific, the computation and space complexities of hierarchical clustering are $O(N^3)$ and $O(N^2)$, respectively, where N is the number of data blocks to cluster. This means that, for example, hierarchical clustering of a 4-GB data set requires TB-scale memory space assuming a data block size of 4 KiB.

There exist a number of *adaptive* clustering algorithms (*e.g.,* [5, 28, 63, 72, 87]) that aim to cluster a data set with limited knowledge of the number of final clusters. Unfortunately, using them for DNN training in DeepSketch is not straightforward either, because their efficiency also highly depends on the initial parameters that are set either randomly or manually, such as the initial number of clusters [28, 63, 72, 87] or the distance threshold to determine the similarity of given two objects [5]. Since the target data set of DeepSketch has an extremely high dimensional space while there is no available hint for good initial parameters, using existing techniques could either require significant effort to find appropriate initial parameters or lead to limited accuracy and/or prohibitive performance overhead due to the use of inappropriate initial parameters.

To overcome the above challenges, we develop DK-Clustering by extending the existing k-means clustering algorithm with specialized initialization steps to dynamically refine the value of *k* while clustering data without any hints for initial parameters. Figure 4 describes the overall process of DK-Clustering composed of two steps that are performed iteratively: **Step 1.** coarse-grained clustering and **Step 2.** fine-grained clustering. Coarse-grained clustering first creates rough clusters within an unlabeled data set, and then fine-grained clustering adjusts the assignment of data blocks by running a modified k-means clustering algorithm. Fine-grained clustering returns a data block to be unlabeled if the block is an *outlier* in the cluster, so that coarse-grained clustering can re-categorize the block at the next iteration. After Steps 1 and 2 converge, DK-Clustering repeats the above steps for *each* cluster in a recursive manner, which enables us to form fine-grained clusters that only contain data blocks sufficiently similar to each other.

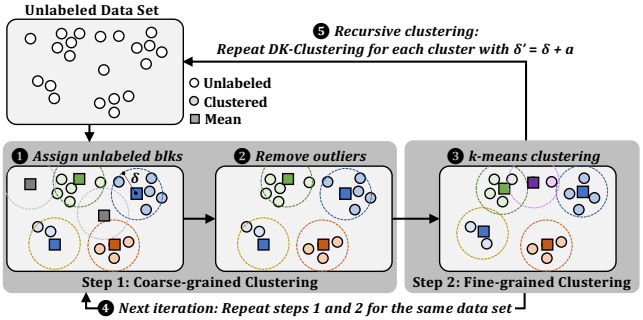


Figure 4: Overall procedure of dynamic k-means clustering.

Step 1: Coarse-Grained Clustering. Coarse-grained clustering takes a set of unlabeled blocks and clusters as the input, and aims to categorize all the unlabeled blocks. Initially, there exist only unlabeled blocks but no cluster, so DK-Clustering creates a new cluster and assigns the first block as the representative block (*i.e., mean*) of the cluster. After that, for each unlabeled block, DK-Clustering measures the data-reduction ratio when the block is delta-compressed with the mean of each cluster through the target delta-compression algorithm (*e.g.,* Xdelta [56,57]). DK-Clustering selects the cluster whose mean provides the highest data-reduction ratio for the unlabeled data block. If the data-reduction ratio is higher than a threshold δ, DK-Clustering adds the unlabeled block to the selected cluster. Otherwise, it creates a new cluster, and the unlabeled block becomes the new cluster's mean (❶ in Figure 4). After categorizing all unlabeled blocks, coarse-grained clustering removes clusters that contain only a single data block from the data set as there are likely no other blocks sufficiently similar to that block (❷).

Step 2: Fine-Grained Clustering. Since coarse-grained clustering roughly assigns unlabeled blocks to clusters, it cannot guarantee that all the data blocks belonging to the same cluster are sufficiently similar to each other. To address this, DK-Clustering performs fine-grained clustering for the resulting clusters from coarse-grained clustering. Fine-grained clustering performs a variant of k-means clustering, adjusting the

mean of each cluster and re-assigning each data block to the cluster containing the nearest mean (❸). Fine-grained clustering operates differently from the typical k-means clustering in three aspects. First, instead of Euclidean distance [19], it uses the delta-compression ratio of two data blocks as the distance function. Second, it derives a cluster's mean by *selecting* the block that provides the highest average data-reduction ratio when delta-compressed relative to each of the other blocks in the cluster. Third, if there is a data block whose delta-compression ratio when delta-compressed relative to the cluster's mean is lower than the threshold δ, DK-Clustering excludes the block from the cluster and considers it as an unlabeled block. After finishing fine-grained clustering on all the clusters, DK-Clustering repeats Steps 1 and 2 over all the clusters until no unlabeled data blocks exist (❹).

Step 3: Recursive Clustering. Fine-grained clustering guarantees that *every* resulting data block belongs to an appropriate cluster where the data block provides a data-reduction ratio higher than the given threshold δ when delta-compressed relative to the cluster's mean. Even though a sufficiently high value for δ would allow DK-Clustering to group only similar data blocks into the same cluster, other values for δ can provide better clustering results. In order to automatically find the best δ for a data set, once DK-Clustering reaches the convergence with a given threshold δ, it performs Steps 1 and 2 for *each* cluster using a new threshold $\delta' = \delta + \alpha$ in a recursive manner (❺). Data blocks assigned to each cluster are considered unlabelled again for the next recursion with the new threshold δ′. The recursion terminates when splitting a cluster shows no more benefit in improving the data-reduction ratio. More specifically, DK-Clustering stops the recursion for a cluster if the average data-reduction ratio of data blocks in the cluster is similar or lower than the average ratio of sub-clusters spawned from the cluster.

DK-Clustering Complexity. The space complexity of DK-Clustering is $O(N)$ since it only requires storing per-block information about which cluster the block belongs to. The computation complexity of DK-Clustering is $O(N \times K_F) + O(N^2/K_C) < O(N^2)$, where K_C and K_F are the number of total clusters after coarse-grained and fine-grained clustering steps, respectively. Although the number of iterations for DK-Clustering can vary depending on workload, DK-Clustering finishes within up to eight iterations for our training data sets. Note that, even for an extreme case where DK-Clustering requires a large number of iterations, we can easily limit the maximum number of iterations at minimal degradation in clustering quality. For example, one can set a threshold distance to finish DK-Clustering once it groups all data blocks such that any data block's distance from the corresponding cluster's mean is lower than the threshold distance.

4.2 Neural-Network Training

Figure 5 shows our method to train a DNN model for DeepSketch to generate a data block's sketch, which consists of two steps. In the first step, we train a *classification* model (❶) using the C_{TRN} clusters formed by DK-Clustering as different target classes. The first part consists of three standard 1D convolutional layers applying the max pooling and batch normalization techniques, which capture spatial locality of neighbor bytes within the data block. The network is then connected to dense layers to learn the relationship between the extracted spatial features and the target class.[3]

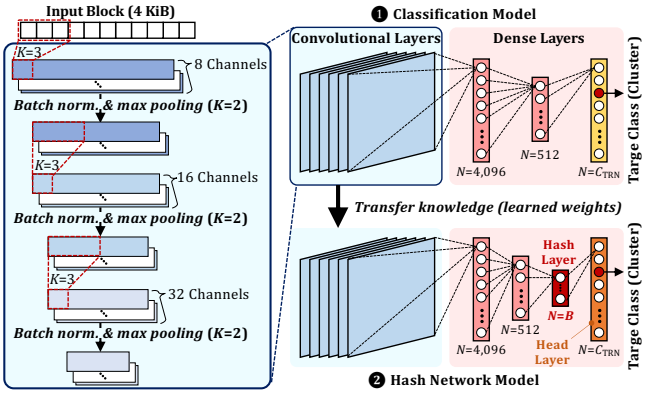


Figure 5: NN models of DeepSketch.

After training the classification model, in the second step, we transfer the learned knowledge of the classification model to a *hash* network model (❷). We employ a state-of-the-art learning-to-hash technique called GreedyHash [79]. We first initialize the hash network with the weights of the classification model. Instead of using the last layer of the classification model, we train the hash network with two different layers, a hash layer and a head layer, each of which learns the binary hash and class likelihood, respectively.

A key challenge in NN training for DeepSketch is that data blocks are not uniformly distributed over C_{TRN} clusters. In our data set, the largest 10% clusters contain 47.93% of the total data blocks. It would render training of the NN to be significantly biased towards specific bit patterns. To address this, we resize each of C_{TRN} clusters to have the same number of N_{BLK} blocks by 1) randomly selecting N_{BLK} blocks within a cluster containing more blocks than other clusters and 2) adding data blocks randomly and slightly modified from ones in a cluster containing fewer blocks.

Once training the hash network, the hash layer yields the B-bit representation for an input block, *i.e.,* the input block's sketch, allowing any two similar data blocks to have similar sketches with low Hamming distance. Note that, even if two data blocks do not belong to any of C_{TRN} clusters, we can infer their binary hash values based on the likelihood that each block belongs to the clusters, which dramatically improves the adaptability of our NN model over various data sets.

[3] We explore multiple NN structures and choose the one that provides the best classification accuracy and data-reduction ratio (shown in Figure 5). For example, when using a much simpler multi-layer perceptron (MLP) networks [24], DeepSketch hardly provides data-reduction benefits (less than 1%) over existing SF-based techniques. Adding the number of dense layers in the classification model in Figure 5 does not improve classification quality, either. We discuss detailed results for hyper-parameter search in Section 4.4.

4.3 Reference Selection

DeepSketch identifies whether or not any two given data blocks are similar by comparing the two blocks' sketches generated from the hash network model. A key challenge here is that the traditional exact-matching-based search method (which uses a hash table for the SK store) is not effective for the learning-to-hash model. For example, the hash network model may generate similar but few-bit different sketches for some blocks beneficial to be delta-compressed, which causes an exact-matching-based search method to misjudge those blocks to be dissimilar.

To address this issue, we use the approximate nearest-neighbor search (ANN) technique. Unlike the standard exact nearest-neighbor search, ANN techniques provide a scalable and performance-efficient way to find the most similar values by relaxing search conditions. In particular, we use the NGT library [16] that supports searching with high-dimensional binary data using neighborhood graphs and tree indexing.

Figure 6 illustrates the reference selection procedure of DeepSketch. For each incoming block, DeepSketch first computes its sketch, $\mathbf{H}$, using the hash network model. It then searches for the similar block from two SK stores. The first SK store utilizes the ANN technique, and DeepSketch queries it with $\mathbf{H}$ to get the data block with the most similar sketch, $\widehat{\mathbf{H}}$, in the ANN model. The other SK store buffers the sketches of R most-recently-written blocks. Let $\Delta(\mathbf{H}, \widehat{\mathbf{H}})$ be the Hamming distance between the two hash values. For each recent block in the buffer store, DeepSketch checks if there is a block with a Hamming distance smaller than $\Delta(\mathbf{H}, \widehat{\mathbf{H}})$. If there exists, DeepSketch chooses the block from the buffer store as the reference for the incoming block. Otherwise, it uses the block from the ANN-based SK store (*i.e.,* the block whose sketch is $\widehat{\mathbf{H}}$) as the reference.

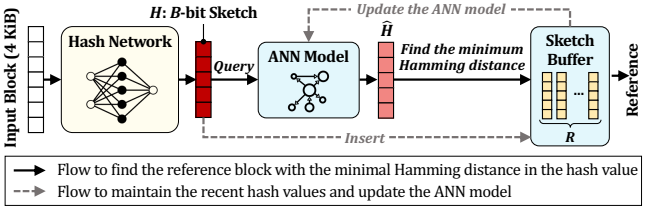


Figure 6: Overview of the reference selection procedure.

The underlying reason for using the two SK stores is that, under the current implementation using the NGT library, updating the ANN model takes a non-negligible amount of time. To avoid frequent updates of the data structure that would hurt the performance of DeepSketch, we design DeepSketch to update the ANN model in a batch by buffering the sketches of recently-written data blocks. When the number of sketches in the buffer exceeds a threshold T_{BLK} (*e.g.,* 128 in our default settings), DeepSketch flushes the buffered sketches to the ANN-based SK store. Note that it is important to check the sketch buffer in order to maximize the data-reduction ratio of DeepSketch. In our evaluation, 13.8% of the reference blocks are found in the sketch buffer on average (up to 33.8%).

4.4 Hyper-Parameter Exploration

This section presents our hyper-parameter exploration for DeepSketch to achieve high accuracy in reference search with a convolutional hash network.

Classification Model. As discussed in Section 4.2, the DNN training procedure of DeepSketch has two steps to train the classification model and the hash network model, respectively. To generate accurate sketches of data blocks, the classification model should predict the correct target classes (*i.e.,* the clusters formed by DK-Clustering). We identify the best hyper-parameters for the proposed classification model using the standard machine learning practice of the grid search along with nested cross-validation. We choose the number of the convolutional and dense layers from the grid $\langle 1,2,3\rangle$, the number of the convolution channels size from $\langle 8,16,32,64\rangle$, the number of neurons for each dense layer from $\langle 512,1,024,2,048,4,096\rangle$, the dropout rate for the dense layers from $\langle 0.0,0.1,0.2,0.5\rangle$, and the learning rate from $\langle 0.01,0.02,0.005,0.1,0.5\rangle$. We utilize ReLU for the activation function for each layer and train the model with the Adam optimizer [41]. We use 10% of samples in our data sets for training and the remaining 90% for testing. Finally, we select the proposed classification model structure that shows the best testing accuracy in the cross-validation.

Figure 7 shows the loss and testing accuracy changes over training epochs for the classification model. The proposed classification model accurately predicts the target cluster identified in DK-Clustering even though the data sets used in our evaluation has a relatively large number of the clusters, $C_{\text{TRN}} = 34,025$. After training with 350 epochs, the model training procedure sufficiently converges, achieving 93.42% for Top-1 and 96.02% for Top-5 accuracy. It implies that the deep learning method itself can accurately identify similar blocks for an incoming block without any other information.

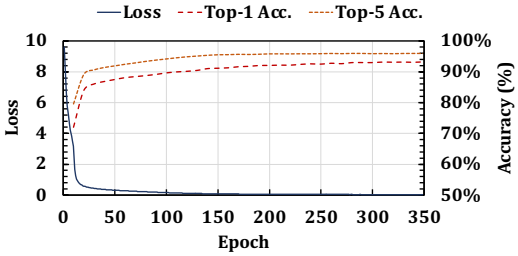
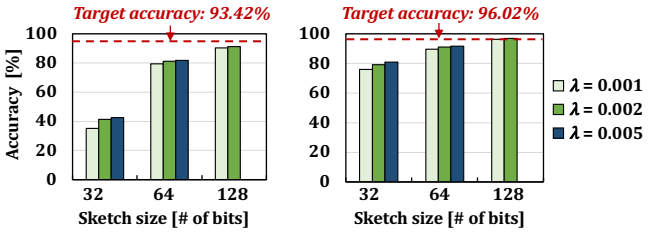


Figure 7: Loss and accuracy of classification model.

Hash Network Model. Next, we train the hash network model while changing the sketch size B. With the smaller B, similar data blocks would have a higher chance to have the same hash value, but it also increases the false-positive rate, *i.e.,* dissimilar blocks belonging to different clusters would have the same hash value. One may set the number of bits with a sufficiently large number, but doing so increases the memory overhead for the SK store and the computation time for ANN search and update processes.

To determine the best sketch size, we verify when the hash network model could achieve the classification model's original accuracy. Recall that the hash network model learns both

the hash coding and classification at the same time. Thus, we can verify whether it correctly classifies the target class by checking the last head layer's activations. Figure 8 summarizes our evaluation results. We evaluate three candidate values for B, 32, 64, and 128, over different learning rates λ. Note that the model does not converge when $B = 128$ and $\lambda = 0.005$, so we omit the results. The results show that the hash network model does not recover the accuracy of the classification model with the small hash bits, 32 and 64, since the representation capability of the hash coding is insufficient. When $B = 128$, we observe that the hash network model also predicts the target clusters with a high accuracy, *e.g.,* it achieves the Top-5 accuracy of 96.92% with $\lambda = 0.002$, exceeding the original target accuracy of the classification model. Thus, we decide to use $B = 128$ for our implementation of DeepSketch.



(a) Top–1 accuracy. (b) Top–5 accuracy.

Figure 8: Accuracy of hash network model.

5 Evaluation

In this section, we evaluate the data-reduction benefits and performance/memory overheads compared to the state-of-the-art super feature (SF)-based sketching technique [86].

5.1 Methodology

Evaluation Platform. We develop a post-deduplication delta-compression platform that is used as a general workbench to implement and evaluate various reference search techniques.[4] Our platform runs on a server machine that employs Intel's Xeon 4110 CPU with 8 cores running at 2.1 GHz, 128-GB DDR4 DRAM, and 8 Samsung 860PRO 1-TB SSDs, while using GeForce RTX 2080 for DNN inference in DeepSketch.

Our platform operates as described in Figure 1; for every host write, it performs deduplication, delta compression, and lossless compression in order. It maintains three main data structures: 1) a fingerprint store for deduplication, 2) a sketch store for delta compression, and 3) a reference table for serving future read requests. The data block size is 4 KiB, which is identical to the default block size of widely-used file systems [18, 60]. We use the MD5 cryptographic hash algorithm [69] to generate a 128-bit fingerprint of an incoming data block and the Xdelta delta-compression algorithm [56, 57] to compress a non-deduplicated data block with its reference block. If there are multiple reference blocks similar to an incoming data block, our platform uses the first-found one as a reference by default. When the platform cannot find a reference block, it compresses the incoming block using the LZ4 algorithm [15]. We set the threshold for the number of buffered sketches to invoke ANN updates to 128, which we empirically determine to minimize the performance overhead of exhaustive search and prevent too frequent ANN updates.

Baseline Technique. We compare DeepSketch against *Finesse* [86], the state-of-the-art SF-based technique that provides much higher throughput while retaining almost the same data-reduction ratio compared to the representative post-deduplication delta-compression technique [75]. We configure Finesse using the default settings presented in [86], which are already optimized and have been shown to provide the best data-reduction efficiency with low overhead for a wide range of workloads. Finesse generates three 192-bit SFs, each of which can be obtained by transposing four features from different hash functions (*i.e.,* twelve (= 3 × 4) Rabin fingerprint functions [68] with a window size of 48 bytes are used in total). It considers that two data blocks are similar if they have one or more matching SFs, and selects the data block that has the largest number of matching SFs with the incoming block as the reference block for delta compression.

Workloads. We use eleven block I/O traces that we collect by running different applications on real systems and capturing write requests (including the requested data) to the storage devices. There is no backup process during trace collection. Table 2 summarizes the characteristics of the traces in terms of the size, deduplication ratio (*i.e., Orignial Data-Set Size / Data-Set Size after Deduplication*), and average compression ratio (*i.e., Original Data-Set Size / Compressed Data-Set Size*). We collect the I/O traces including contents written in the storage system from real desktop machines and servers while running different applications.

Table 2: Summary of the evaluated workloads.

Workload	Description	Size	Dedup. ratio	Comp. ratio
PC	General Ubuntu PC usage	1.57 GB	1.381	2.209
Install	Installing & executing programs	8.83 GB	1.309	2.45
Update	Updating & downloading SW packages	3.73 GB	1.249	2.116
Synth	Synthesizing hardware modules	653 MB	1.898	2.083
Sensor	Sensor data in semiconductor fabrication	91.2 MB	1.269	12.38
Web	Web page caching	959 MB	1.9	6.84
SOF0	Storing Stack Overflow database [33] as of 2010 (SOF0) and 2013 (SOF1–4)	8.98 GB	1.007	2.088
SOF1		13.6 GB	1.01	1.997
SOF2		13.6 GB	1.01	1.996
SOF3		13.6 GB	1.01	1.997
SOF4		13.6 GB	1.01	1.996

For DeepSketch, we use different sets of data for training and testing. In order to evaluate the *adaptability* of DeepSketch (*i.e.,* how well DeepSketch operates under a workload totally different from ones used in its DNN training), we do *not* use the five traces collected from Stack Overflow database [33] (SOF0–4) for training the DNN of DeepSketch. By default, we train DeepSketch's DNN model using a single data set that contains 10% of all the remaining six traces and evaluate DeepSketch with the remaining 90% of the six traces and entire SOF traces.

[4] We open source our platform along with the data sets used in our evaluation [1].

5.2 Overall Data Reduction

Figure 9 shows the data-reduction ratio after post-deduplication delta compression with the two reference search techniques, Finesse and DeepSketch, under the eleven workloads. We only present SOF1 as a representative result of SOF1-4 as they show little variations lower than 0.01%. All values are normalized to the data-reduction ratios of a baseline system that performs only deduplication and lossless compression in order, which we call *no delta compression (noDC)*.

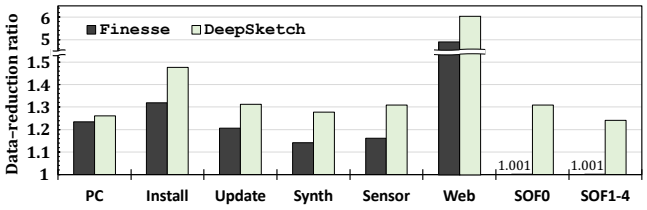


Figure 9: Comparison of overall data-reduction ratio.

We make two main observations from Figure 9. First, DeepSketch significantly outperforms Finesse in most workloads. Except for PC in which DeepSketch provides the similar data-reduction ratio with Finesse, DeepSketch exhibits up to 33% (on average 21%) higher data-reduction ratios than Finesse. In particular, DeepSketch greatly improves the data-reduction ratio by *at least* 24% over Finesse under SOF workloads. This suggests that 1) DeepSketch can improve the data-reduction efficiency for workloads that the state-of-the-art SF-based search technique cannot effectively cope with, and 2) DeepSketch has high adaptability (*i.e.,* it can work efficiently for data sets that are not used for the DNN training). Second, DeepSketch provides higher data-reduction ratios even for highly compressible workloads. Under Web, Finesse significantly reduces the write traffic by about 80% over noDC, but DeepSketch increases the data-reduction ratio even further by 33% compared to Finesse. From our observations, we conclude that DeepSketch is an effective solution to maximize the data-reduction ratio for various workloads.

5.3 Reference Search Pattern Analysis

To better understand how DeepSketch can outperform the state-of-the-art technique, we analyze the reference-search efficiency of DeepSketch and Finesse. Given a data block B, we measure $S_{FS}(B)$ and $S_{DS}(B)$, the number of *saved bytes* by Finesse and DeepSketch, respectively. $S_{FS}(B)$ (or $S_{DS}(B)$) is obtained by subtracting the size of B when delta-compressed with the reference block found by Finesse (or DeepSketch) from the original size of B (*i.e.,* 4 KiB). The larger the $S_{FS}(B)$ (or $S_{DS}(B)$) value, the higher the reference-search efficiency. If a reference search technique fails to find a reference block for B, we compress it using the LZ4 algorithm and then use the compressed size to calculate data saving.

Figure 10 plots coordinates of $x = S_{FS}(B_i)$ and $y = S_{DS}(B_i)$ for a block B_i in each workload. If $x = y$, Finesse and DeepSketch exhibit the same delta-compression ratio (highlighted with a red line in Figure 10), which implies that they select the same reference block. A coordinate $(S_{FS}(B_i), S_{DS}(B_i))$ above

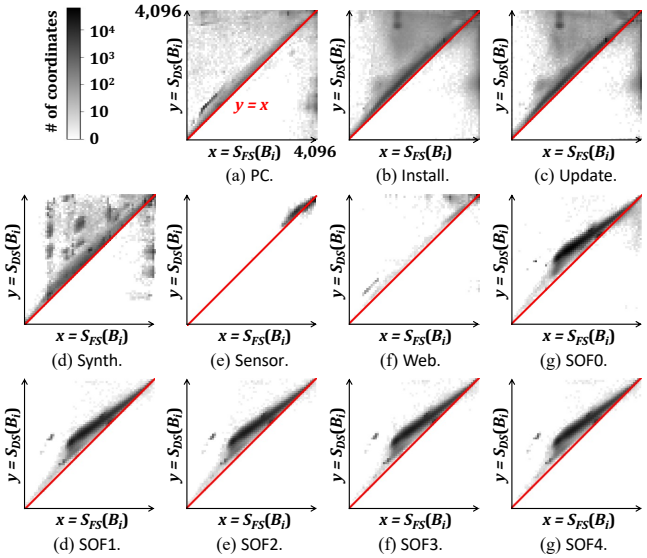


Figure 10: Comparison of the reference-search pattern.

(or below) the line means that DeepSketch provides higher (or lower) data-reduction ratio for block B_i than Finesse.

From Figure 10, we make three observations. First, as expected, DeepSketch provides higher data savings compared to Finesse for a large number of blocks under every workload. Second, despite the higher data savings of DeepSketch over Finesse in general, there are also a non-trivial number of blocks for which Finesse selects better references, achieving higher data savings than DeepSketch. Excluding the SOF workloads, Finesse selects higher-quality references compared to DeepSketch for up to 11.8% of the total blocks. Third, DeepSketch and Finesse show quite different patterns in reference search. As shown in Figure 10, the coordinates in $y > x$ region (*i.e.,* where DeepSketch outperforms Finesse) are close to the line $y = x$, and at the same time, many of them are scattered across a wide range of the region compared to the coordinates in $y < x$ region. On the other hand, a majority of the coordinates in $y < x$ region (*i.e.,* where Finesse outperforms DeepSketch) tend to have a very large y value (*e.g.,* > 3,072). These imply that, while Finesse is effective to find a reference highly similar to an input block, it also misses a number of blocks that DeepSketch can find and use to improve the data-reduction efficiency.

5.4 Combination with Existing Techniques

Our second and third observations in Section 5.3 motivate us to combine DeepSketch with existing techniques to maximize the data-reduction ratio. We design a storage system that employs both Finesse and DeepSketch. When the two techniques find different reference blocks for an incoming block, the system chooses the one that provides a higher data-reduction ratio. Such an approach increases the memory and computation overheads for data sketching but would be desirable for a system where data reduction is paramount (*e.g.,* backup systems). We leave the study of efficiently combining DeepSketch with existing techniques as future work.

Figure 11 shows the combined approach's data-reduction benefits compared to when using either Finesse or DeepSketch alone.[5] We also measure the *optimal* data-reduction ratio (*i.e.,* when every data block is delta-compressed with the best reference block found by brute-force search) for each workload to understand room for improvement after applying the combined approach. To emphasize the benefits of the combined approach over the standalone techniques, we normalize all the results in Figure 11 to Finesse.

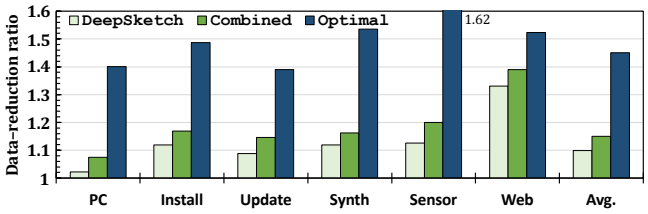


Figure 11: Data-reduction improvement of a combination of DeepSketch and Finesse.

We observe that, as expected, the combined approach further improves the data-reduction ratio compared to the two standalone techniques under most workloads. The combined approach achieves up to 38% and 6.6% (15% and 4.8% on average) data-reduction improvements over Finesse and DeepSketch, respectively. We also observe that the combined approach can reduce the gap in data-reduction ratios between the existing reference search techniques and the optimal. Although there is still large room for improvement (*i.e.,* up to 35% and 26% on average) even after applying the combined approach, the combined approach reduces the gap by up to 81% (*i.e.,* 62% $\rightarrow$ 9.6% under Web) and by 42% on average. From our observations, we conclude that DeepSketch can also be used as a useful method to complement the weakness of existing post-deduplication delta-compression techniques.

5.5 Impact of Training Data-Set Quality

We evaluate the impact of the training data-set quality on the data-reduction ratio of DeepSketch. Figure 12 shows the average data-reduction ratio of DeepSketch for the entire workloads listed in Table 2 when we use two different types of training data sets. First, we evaluate how DeepSketch's benefit changes when we train its DNN model using 1%/2%/3%/5%/10% of the *entire* data sets (the blue line in Figure 12). Second, we measure DeepSketch's benefit when we use 10% of requests only from Sensor for DNN training (the dashed red line in Figure 12). When we use x% (< 10%) of each trace for training, we use the remaining $(100-x)$% to evaluate the data-reduction ratio of DeepSketch. All values in Figure 12 are normalized to the data-reduction ratio obtained when using 10% of the entire data sets for DNN training.

We make two observations from Figure 12. First, while a larger training data set increases DeepSketch's benefit, DeepSketch can provide a fairly good data-reduction ratio even with

[5] We omit the results of the SOF workloads in Figure 11 because there is no motivation to combine DeepSketch with Finesse under such workloads for which Finesse provides negligible data reduction.

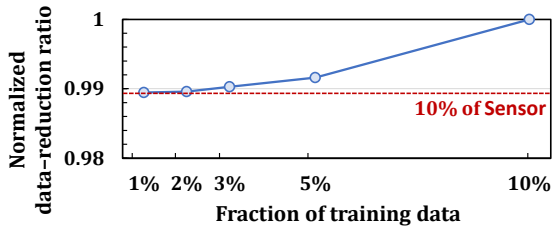


Figure 12: Effect of training data set on data-reduction ratio.

a very small training data set. Using only 1% of the traces for DNN training provides 98.9% of the data-reduction ratio obtained when using 10% of the traces. Second, DeepSketch can provide a high data-reduction ratio even when we use a training data set collected from a single trace. Compared to when we use 10% of all traces for DNN training, using 10% of only Sensor decreases the data-reduction ratio by less than 1%. Based on our observations, we conclude that it is possible to train an effective DNN model for DeepSketch with a limited data set, while providing high adaptability for diverse input data sets.

To study the detailed impact of the training data-set quality, we analyze how the sketches generated by DeepSketch change with different training data sets. To this end, we measure the average *data-saving ratio* (*i.e.,* 1$-$ *Delta-Compressed Data Size / Original Data Size*) of delta-compressed blocks depending on the Hamming distance between the sketches of the input and reference blocks (*i.e.,* $\Delta(\mathbf{H},\widehat{\mathbf{H}})$ in Section 4.3.) Figure 13 shows the relationship between the data-saving ratio and sketch Hamming distance for three different DNN models trained with 10% of Sensor (10%-Sensor) and 1%/10% of all traces (1%-All and 10%-All). In general, the higher the data-saving ratio at a low sketch Hamming distance, the more accurate the sketches generated by DeepSketch.

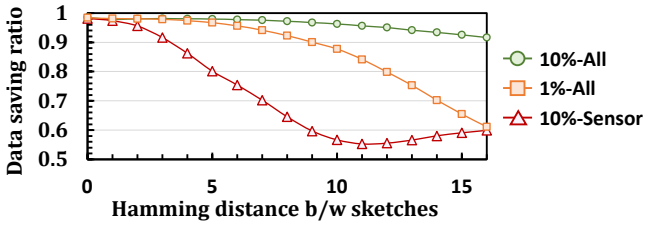


Figure 13: Effect of training data set on sketch accuracy.

We identify the following two findings from Figure 13. First, for all DNN models, DeepSketch provides extremely high data saving (close to 1) when $\Delta(\mathbf{H},\widehat{\mathbf{H}}) \leq 2$. The result shows that all the three DNN models enable DeepSketch to identify highly similar data blocks by generating almost identical sketches. It is due to the nature of the DNN-based learning-to-hash method: a DNN can be easily trained to yield the same hash values for the data with negligible differences. Second, in 1%-All and 10%-Sensor, the data-reduction ratio degrades more significantly as the Hamming distance increases, compared to 10%-All. It suggests that we can further improve the benefit of DeepSketch by increasing the accuracy of the sketch generation with a better DNN model, *e.g.,* using high-quality data sets and/or advanced model ar-

chitectures. In the current version of DeepSketch, the ANN model compensates for such potential accuracy loss by finding sufficiently-good reference blocks with best efforts.

5.6 Overhead Analysis

Performance Overhead. Figure 14 shows the average throughput of DeepSketch and the combined approach of DeepSketch and Finesse under different workloads, normalized to Finesse.[6] DeepSketch and combined approach provide up to 73.7% and 44.9% (44.6% and 28.4% on average across all workloads) of the average throughput of Finesse. This non-trivial performance overhead is due to the inherent trade-off between the data-reduction ratio and throughput in post-deduplication delta compression; performing delta compression for more data blocks would increase the data-reduction ratio, but it comes at the cost of performance degradation since delta compression takes more time compared to lossless compression (*e.g.,* in our current implementation, LZ4 takes 6.9 μs per block on average, which is less than 10% of the average execution time of Xdelta).

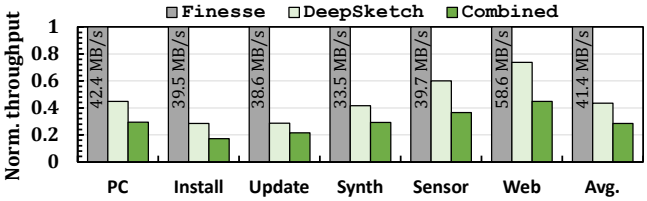


Figure 14: Performance overhead of DeepSketch.

To better understand the performance overheads of DeepSketch, we measure the average latency of each step per input data block during the post-deduplication delta-compression process. DeepSketch requires two modifications on existing techniques, 1) replacing the SF-based sketching engine with the DNN-based one and 2) using the ANN engine described in Section 4.3 as the SK store. For fair comparison, we implement the SK store of Finesse using the unordered-map data structure that provides $O(1)$ time complexity for lookup. Figure 15 visualizes the fraction of the average time spent for each step in the entire data-reduction process.

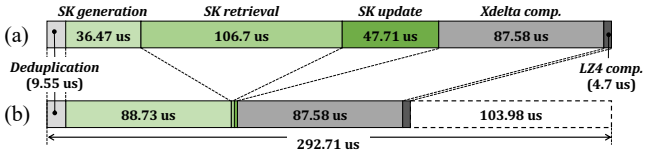


Figure 15: Average latency for each data-reduction step in (a) DeepSketch and (b) Finesse.

As shown in Figure 15, DeepSketch and Finesse operate differently in only three steps of the entire data-reduction process: 1) sketch generation for an incoming block, 2) sketch retrieval from the SK store, and 3) sketch update to the SK store. The other steps, including deduplication, Xdelta compression, and LZ4 compression, are performed in the same ways. Due to the simplicity of the hash network model network and GPU acceleration, DeepSketch reduces the latency of sketch generation from 88.73 μs to 36.47 μs (by 58.9%) over Finesse. However, using the ANN engine significantly increases the latencies for sketch retrieval and update, leading to 55.1% increase in the total average latency over Finesse.

The performance overhead of DeepSketch over Finesse is non-trivial, but it would not be a serious obstacle for its wide adoption due to two reasons. First, we target a system where data reduction is critically important so that DeepSketch's benefits outweigh its performance overheads. Second, the performance overhead of DeepSketch can be mitigated in several ways. For example, if the data-reduction process is performed in *background*, its negative performance impact could be relatively small. We can also leverage the parallelism of multi-core CPUs to optimize software modules. For example, the sketch update procedure can be performed in parallel with other modules. This hides the cost of updating sketches during the compression steps, thereby reducing the performance overhead by 45.8% (*i.e.,* 103.98 μs $\rightarrow$ 56.27 μs).

Memory Overhead. Like existing post-deduplication delta-compression techniques [75, 86], DeepSketch inevitably requires additional memory space for the sketch store. Despite the smaller sketch size of DeepSketch compared to existing techniques [75, 86] (128 bits vs. 192 bits), DeepSketch's memory overhead might be unacceptable if it keeps track of the sketches of *all* non-deduplicated data blocks. For example, suppose that the data block size is 4 KiB, and 80% of the stored data is unique (*i.e.,* non-deduplicated). Then, the required memory space for the sketch store is about 0.3% ($0.8\times16/4{,}096$) of the size of the stored data (e.g., around 100-GB memory space to work with 32-TB data).

However, the memory overhead would not be a significant obstacle to use DeepSketch in practice for two reasons. First, the memory overhead for the sketch store is a common problem in all the sketch-based techniques. Second, prior works demonstrate that a small fraction of data blocks are frequently used as the reference block for many input blocks [26, 64]. Thus, keeping only most-frequently-used sketches in a limited-size sketch store (i.e., a with least-frequently-used (LFU) eviction policy) would provide sufficiently high compression efficiency. We leave such further optimizations to mitigate DeepSketch's memory overhead for future work.

6 Discussion

Scalability to Larger Data Sets. As shown in our evaluation, DeepSketch provides high data-reduction ratios even for workloads that are not used in DNN training (*e.g.,* SOF workloads), which implies the high generalizability of DeepSketch. Nevertheless, due to the limited amount of data sets publicly accessible, it is difficult to say whether DeepSketch would be effective under *any* given workload. For example, DeepSketch may require a larger DNN model to provide suf-

[6] Note that DNN training does *not* affect DeepSketch's throughput because it can be performed *offline* as explained in Section 4. In our system described in Section 5.1, DNN training (including DK-Clustering) takes less than 4 hours with 300 epochs for our 1.6-GB training data set.

ficient benefits for large data sets (that we do not observe in this work), which would significantly increase the training overheads of DeepSketch. However, we believe that DeepSketch would be able to work for larger data sets due to three reasons. First, DeepSketch 's DNN model has much smaller computation complexity than state-of-the-art DNN models, so there is significant room for DeepSketch to use the larger DNN models. Second, the memory space required for training depends more on the size of the DNN model rather than the size of the training data set. Our current model is only a few hundreds of megabytes in size, which can be run on a single commonly-used GPU. Third, as explained in Section 4, DNN training can be performed offline in different machines with more computing/memory resources.

Cost-Effectiveness of DeepSketch. The current version of DeepSketch requires a powerful GPU for DNN inference/training and thus introduces non-trivial performance and power overheads. However, we believe that such overheads would not be a significant obstacle for the wide adoption of DeepSketch due to two reasons. First, as explained in Section 4, DNN training can be done in different machines (*e.g.,* in cloud servers) without requiring frequent retraining, and multiple storage servers (that store similar types of data) can use the same DNN model, amortizing the training cost. Second, there has been significant effort to develop high-performance and energy-efficient accelerators for both light-weight DNN inference (e.g., [6, 27, 66, 73]) and ANN search (e.g., [32, 40, 71]), which would greatly reduce the performance, power, and resource overheads of DeepSketch.

7 Related Work

To our knowledge, this work is the first to propose a learning-based data-sketching technique for accurate reference search in storage-level delta compression. As we have already discussed state-of-the-art techniques closely related to DeepSketch in Sections 2 and 3, in this section, we briefly discuss other recent works on 1) storage-level data reduction and 2) machine learning-based video/image compression.

Storage-level Data Reduction. The fundamental ideas of data-reduction techniques were proposed several decades ago. Hence, their theoretical properties and limitations, in terms of data reduction, have been studied intensively. Recent studies focus more on how to efficiently deploy them to various platforms to achieve space savings with faster compression speeds, lower computation costs, and less energy consumption [12, 26, 46, 64, 85, 86]. For example, SmartDedup [85] proposes a low-cost deduplication technique for resource-constrained devices where computing resources as well as energy budget are seriously limited. Finesse [86] is a representative example of enhancing delta-compression speed without loss in data-reduction ratio by relaxing the complexity of sketch generation. Some prior works [12, 26, 46] present that deduplication and compression could be integrated in an SSD controller to improve storage lifetime as well as performance.

The key difference of this study from the above recent works is that this work presents a new direction to improve *data-reduction ratio*, the fundamental goal of a data-reduction technique. Our work analyzes the limitations of probabilistic and statistical approaches and shows that emerging deep-learning methods can be promising alternatives to and/or complements the traditional methods.

Machine Learning for Video and Image Compression. Several works attempt to improve video/image compression efficiency using machine learning [2, 13, 14, 54, 61, 77, 77]. To enhance existing video compression algorithms, some leverage CNNs [13, 14, 77] and others employ Long Short-Term Memory (LSTM) networks to learn video representations [77] and predict future frames [54]. Their common idea is to accurately predict pixel values of next video frames and store only deltas for reconstruction. More recent studies use Generative Adversarial Networks (GAN) to generate part or all of the image content from a semantic label map [2, 61]. They achieve space savings by storing only a smaller amount of preserved data and the label map in storage devices.

DeepSketch is different from these studies in two aspects. First, unlike existing ML-based compression methods that target images and videos, DeepSketch aims to compress binary data, which requires handling extremely high-dimensional data sets without any semantic information. Second, ML-based compression methods are basically lossy compression algorithms, but our system is a lossless compression system that enables us to reconstruct original data without any data loss.

8 Conclusion

We introduce DeepSketch, the first learning-based reference search technique to improve the data-reduction efficiency of post-deduplication delta compression. DeepSketch uses the learning-to-hash method to overcome the limitations of existing techniques that miss a number of good reference candidates for delta compression of incoming data blocks. We present a new deep neural network training method that enables DeepSketch to efficiently learn delta-compression-aware data representation for unlabeled data sets with an extremely high dimensional space. Using various real-world data sets, we experimentally demonstrate that DeepSketch is an efficient solution not only as a replacement for but also as a complement to existing reference search techniques, significantly reducing the data-reduction gap from the optimal.

Acknowledgements

We would like to thank our shepherd Chao Tian and anonymous reviewers for their feedback and comments. We thank the SAFARI Research Group members for feedback and the stimulating intellectual environment they provide. We thank our industrial partners, especially Google, Huawei, Intel, Microsoft, and VMware, for their generous donations. This work was in part supported by the Semiconductor Research Corporation, SNU-SK Hynix Solution Research Center (S3RC), and the National Research Foundation (NRF) of Korea (NRF-2018R1A5A1060031, NRF-2020R1A6A3A03040573). (*Co-corresponding Authors: Sungjin Lee and Onur Mutlu*)

References

[1] DeepSketch GitHub repository, 2018. https://github.com/dgist-datalab/deepsketch-fast2022.

[2] Eirikur Agustsson, Michael Tschannen, Fabian Mentzer, Radu Timofte, and Luc Van Gool. Generative adversarial networks for extreme learned image compression. In *ICCV*, October 2019.

[3] Miklos Ajtai, Randal Burns, Ronald Fagin, Darrell D. E. Long, and Larry Stockmeyer. Compactly encoding unstructured inputs with differential compression. *JACM*, 2002.

[4] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO*, 2021.

[5] Sanjiv K. Bhatia. Adaptive k-means clustering. In *FLAIRS*, 2004.

[6] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F. Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. Google neural network models for edge devices: Analyzing and mitigating machine learning inference bottlenecks. In *PACT*, 2021.

[7] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *JCSS*, 2000.

[8] Randal Burns, Larry Stockmeyer, and Darrell D. E. Long. In-place reconstruction of version differences. *IEEE TKDE*, 2003.

[9] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *ASPLOS*, 1992.

[10] Yue Cao, Mingsheng Long, Bin Liu, and Jianmin Wang. Deep cauchy hashing for hamming space retrieval. In *CVPR*, 2018.

[11] Yue Cao, Mingsheng Long, and Jianmin Wang. Collective deep quantization for efficient cross-modal retrieval. In *AAAI*, 2017.

[12] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *USENIX FAST*, 2011.

[13] T. Chen, H. Liu, Q. Shen, T. Yue, X. Cao, and Z. Ma. DeepCoder: a deep neural network based video compression. In *VCIP*, 2017.

[14] Z. Chen, T. He, X. Jin, and F. Wu. Learning for video compression. *IEEE TCSVT*, 30(2), 2020.

[15] Yann Collet. LZ4 – extemely fast compression algorithm. http://lz4.github.io/lz4/.

[16] Yahoo! Japan Corp. Neighborhood graph and tree for indexing high-dimensional data. https://github.com/yahoojapan/NGT.

[17] Alvin Cox. JEDEC SSD endurance workloads. In *FMS*, 2011.

[18] Helen Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.

[19] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 1980.

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[21] Wei Dong, Fred Douglis, Kai Li, R Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *USENIX FAST*, 2011.

[22] 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 *USENIX FAST*, 2009.

[23] Venice Erin Liong, Jiwen Lu, Gang Wang, Pierre Moulin, and Jie Zhou. Deep hashing for compact binary codes learning. In *CVPR*, 2015.

[24] Matt W. Gardner and Stephen R. Dorling. Artificial neural networks (the multilayer perceptron) – a review of applications in the atmospheric sciences. *Atmospheric Environment*, 1998.

[25] Jim Gray and Catharine Van Ingen. Empirical measurements of disk failure rates and error rates. *arXiv*, 2007.

[26] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *USENIX FAST*, 2011.

[27] Ramyad Hadidi, Jiashen Cao, Yilun Xie, Bahar Asgari, Tushar Krishna, and Hyesoon Kim. Characterizing the deployment of deep neural networks on commercial edge devices. In *IISWC*, 2019.

[28] Greg Hamerly and Charles Elkan. Learning the k in k-means. In *NeurIPS*, 2003.

[29] Geoffrey E. Hinton, Terrence Joseph Sejnowski, et al. *Unsupervised Learning: Foundations of Neural Computation*. MIT press, 1999.

[30] Daniel Reiter Horn, Ken Elkabany, Chris Lesniewski-Lass, and Keith Winstein. The design, implementation, and deployment of a system to transparently compress hundreds of petabytes of image files for a file-storage service. In *USENIX NSDI*, 2017.

[31] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE*, 1952.

[32] Mohsen Imani, Yeseong Kim, and Tajana Rosing. Nngine: Ultra-efficient nearest neighbor accelerator based on in-memory computing. In *ICRC*, 2017.

[33] Stack Exchange Inc. Stack Exchange data dump. https://archive.org/details/stackexchange.

[34] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multi-dimensional spaces. In *STOC*, 1997.

[35] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

[36] Navendu Jain, Michael Dahlin, and Renu Tewari. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *USENIX FAST*, 2005.

[37] Daniel A. Jiménez. Fast path-based neural branch prediction. In *MICRO*, 2003.

[38] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *HPCA*, 2001.

[39] Stephen C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 1967.

[40] Himanshu Kaul, Mark A. Anders, Sanu K. Mathew, Gregory Chen, Sudhir K. Satpathy, Steven K. Hsu, Amit Agarwal, and Ram K. Krishnamurthy. 14.4A 21.5M-query-vectors/s 3.37 nJ/vector reconfigurable k-nearest-neighbor accelerator with adaptive precision in 14nm tri-gate CMOS. In *ISSCC*, 2016.

[41] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[42] Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009.

[43] Brian Kulis and Trevor Darrell. Learning to hash with binary reconstructive embeddings. In *NeurIPS*, 2009.

[44] Yann LeCun. The MNIST database of handwritten digits, 1998.

[45] Sungjin Lee, Taejin Kim, Jisung Park, and Jihong Kim. An integrated approach for managing the lifetime of flash-based SSDs. In *DATE*, 2013.

[46] Sungjin Lee, Jihoon Park, Kermin Fleming, and Jihong Arvind, Kim. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *IEEE TCE*, 2011.

[47] Cong Leng, Jiaxiang Wu, Jian Cheng, Xi Zhang, and Hanqing Lu. Hashing for distributed data. In *ICML*, 2015.

[48] 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 *USENIX ATC*, pages 395–410, 2019.

[49] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *USENIX FAST*, 2009.

[50] Kevin Lin, Huei-Fang Yang, Jen-Hao Hsiao, and Chu-Song Chen. Deep learning of binary hash codes for fast image retrieval. In *CVPR*, 2015.

[51] Xing Lin, Guanlin Lu, Fred Douglis, Philip Shilane, and Grant Wallace. Migratory compression: Coarse-grained data reordering to improve compressibility. In *USENIX FAST*, 2014.

[52] Weiqiang Liu, Faqiang Mei, Chenghua Wang, Maire O'Neill, and Earl E. Swartzlander. Data compression device based on modified LZ4 algorithm. *IEEE TCE*, 2018.

[53] Stuart Lloyd. Least squares quantization in PCM. *IEEE TIT*, 1982.

[54] William Lotter, Gabriel Kreiman, and David D. Cox. Deep predictive coding networks for video prediction and unsupervised learning. *CoRR*, abs/1605.08104, 2016.

[55] Jingwei Ma, Gang Wang, and Xiaoguang Liu. DedupeSwift: Object-oriented storage system based on data deduplication. In *TrustCom*, 2016.

[56] Josh MacDonald. Xdelta: Open-source binary diff, differential compression tools, VCDIFF (RFC 3284) delta compression. http://xdelta.org.

[57] Josh MacDonald. *File system support for delta compression*. PhD thesis, 2000.

[58] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. Using transparent compression to improve SSD-based I/O caches. In *EuroSys*, 2010.

[59] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *USENIX FAST*, 2016.

[60] 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*, 2007.

[61] Fabian Mentzer, George Toderici, Michael Tschannen, and Eirikur Agustsson. High-fidelity generative image compression. In *NeurIPS*, 2020.

[62] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *USENIX FAST*, 2011.

[63] S. Ehsan Yasrebi Nayini, Somayeh Geravand, and Ali Maroosi. A novel threshold-based clustering method to solve k-means weaknesses. In *ICECDS*, 2017.

[64] Jisung Park, Sungjin Lee, and Jihong Kim. DAC: Dedup-assisted compression scheme for improving lifetime of NAND storage systems. In *DATE*, 2017.

[65] P. Jonathon Phillips, Amy N. Yates, Ying Hu, Carina A. Hahn, Eilidh Noyes, Kelsey Jackson, Jacqueline G. Cavazos, Géraldine Jeckeln, Rajeev Ranjan, Swami Sankaranarayanan, Jun-Cheng Chen, Carlos D. Castillo, Rama Chellappa, David White, and Alice J. O'Toole. Face recognition accuracy of forensic examiners, superrecognizers, and face recognition algorithms. *Proc. NAS*, 2018.

[66] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. Binary neural networks: A survey. *Pattern Recognition*, 2020.

[67] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *USENIX FAST*, 2002.

[68] Michael O. Rabin. Fingerprinting by random polynomials. *Technical Report*, 1981.

[69] Ronald Rivest and S. Dusse. The MD5 message-digest algorithm, 1992.

[70] Chaitanya K. Ryali, John J. Hopfield, Leopold Grinberg, and Dmitry Krotov. Bio-inspired hashing for unsupervised similarity search. In *CVPR*, 2020.

[71] Jyotishman Saikia, Shihui Yin, Zhewei Jiang, Mingoo Seok, and Jae-sun Seo. K-nearest neighbor hardware accelerator using in-memory computing SRAM. In *ISLPED*, 2019.

[72] Ahamed Shafeeq and K. S. Hareesha. Dynamic clustering of data with modified k-means algorithm. In *ICICN*, 2012.

[73] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *ISCA*, 2016.

[74] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 1948.

[75] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. WAN optimized replication of backup datasets using stream-informed delta compression. In *USENIX FAST*, 2012.

[76] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *USENIX FAST*, 2012.

[77] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhutdinov. Unsupervised learning of video representations using LSTMs. In *ICML*, 2015.

[78] Secure Hash Standard. FIPS Pub 180-1. *National Institute of Standards and Technology*, 1995.

[79] Shupeng Su, Chao Zhang, Kai Han, and Yonghong Tian. Greedy hash: Towards fast optimization for accurate hash coding in cnn. In *NIPS*, pages 806–815, 2018.

[80] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. A survey on learning to hash. *IEEE TPAMI*, 2017.

[81] Guanying Wu and Xubin He. Delta-FTL: improving SSD lifetime via exploiting content locality. In *EuroSys*, 2012.

[82] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. DARE: A deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads. *IEEE TC*, 2015.

[83] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Michael L. Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. Achieving human parity in conversational speech recognition. *arXiv*, 2016.

[84] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Michael L. Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. Toward human parity in conversational speech recognition. *IEEE/ACM TASLP*, 2017.

[85] Qirui Yang, Runyu Jin, and Ming Zhao. SmartDedup: optimizing deduplication for resource-constrained devices. In *USENIX ATC*, 2019.

[86] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *USENIX FAST*, 2019.

[87] Xin Zheng, Qinyi Lei, Run Yao, Yifei Gong, and Qian Yin. Image segmentation based on adaptive k-means algorithm. *EURASIP JIVP*, 2018.

[88] Benjamin Zhu, Kai Li, and R. Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *USENIX FAST*, 2008.

[89] Han Zhu, Mingsheng Long, Jianmin Wang, and Yue Cao. Deep hashing network for efficient similarity retrieval. In *AAAI*, 2016.

[90] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE TIT*, 1978.

The *what*, The *from*, and The *to*: The Migration Games in Deduplicated Systems

Roei Kisous and Ariel Kolikant
Computer Science Department, Technion

Abhinav Duggal
DELL EMC

Sarai Sheinvald
ORT Braude College of Engineering

Gala Yadgar
Computer Science Department, Technion

Abstract

Deduplication reduces the size of the data stored in large-scale storage systems by replacing duplicate data blocks with references to their unique copies. This creates dependencies between files that contain similar content, and complicates the management of data in the system. In this paper, we address the problem of data migration, where files are remapped between different volumes as a result of system expansion or maintenance. The challenge of determining which files and blocks to migrate has been studied extensively for systems without deduplication. In the context of deduplicated storage, however, only simplified migration scenarios were considered.

In this paper, we formulate the general migration problem for deduplicated systems as an optimization problem whose objective is to minimize the system's size while ensuring that the storage load is evenly distributed between the system's volumes, and that the network traffic required for the migration does not exceed its allocation.

We then present three algorithms for generating effective migration plans, each based on a different approach and representing a different tradeoff between computation time and migration efficiency. Our *greedy algorithm* provides modest space savings, but is appealing thanks to its exceptionally short runtime. Its results can be improved by using larger system representations. Our *theoretically optimal algorithm* formulates the migration problem as an ILP (integer linear programming) instance. Its migration plans consistently result in smaller and more balanced systems than those of the greedy approach, although its runtime is long and, as a result, the theoretical optimum is not always found. Our *clustering algorithm* enjoys the best of both worlds: its migration plans are comparable to those generated by the ILP-based algorithm, but its runtime is shorter, sometimes by an order of magnitude. It can be further accelerated at a modest cost in the quality of its results.

1 Introduction

Many large-scale storage systems employ data deduplication to reduce the size of the data that they store. The deduplication process identifies duplicate data blocks in different files and replaces them with pointers to a unique copy of the block stored in the system. This reduction in the system's size comes at the cost of increased system complexity. While the complexity of reading, writing, and deleting data in deduplicated storage systems has been addressed by many academic studies and commercial systems, the high-level management aspects of large-scale systems, such as capacity planning, caching, and quality and cost of service, still need to be adapted to deduplicated storage [44].

This paper focuses on the aspect of *data migration*, where files are remapped between separate deduplication domains, or *volumes*. A volume may represent a single server within a large-scale system, or an independent set of servers dedicated to a customer or dataset. Files might be remapped as a result of volumes reaching their capacity limitation or of other bottlenecks forming in the system. Deduplication introduces new considerations when choosing which files to migrate, due to the data dependencies between files: when a file is migrated, some of its blocks may be deleted from its original volume, while others might still belong to files that remain on that volume. Similarly, some blocks need to be transferred to the target volume, while others may already be stored there. An efficient migration plan must optimize several, possibly conflicting objectives: the physical size of the stored data after migration, the load balancing between the system's volumes, i.e., the physical size of the data stored on each volume, and the network bandwidth generated by the migration itself.

Several recent studies address specific (simplified) cases of data migration in deduplicated systems. Harnik et al. [28] address capacity estimation and propose a greedy algorithm for reducing the system's size. Rangoli [41] is a greedy algorithm for *space reclamation*, where a set of files is deleted to reclaim some of the system's capacity. GoSeed [40] is an ILP (integer linear programming)-based algorithm for the *seeding* problem, in which files are remapped into an initially empty target volume. While even the seeding problem is shown to be NP-hard [40], none of these studies address the conflicting objectives involved in the full data migration problem. Namely, the tradeoff between minimizing the system size, minimizing the network traffic consumed during migration, and maximizing the load balance between the volumes in the system.

In this paper, we address, for the first time, the general case of data migration. We begin by formulating the data migration

problem in its most general form, as an optimization problem whose main goal is to minimize the overall size of the system. We add the traffic and load balancing considerations as constraints on the migration plan. The degree in which these constraints are enforced directly affects the solution space, allowing the system administrator to prioritize different costs. Thus, the problem of data migration in deduplication systems maps to finding what to migrate, where to migrate from, and where to migrate to within the traffic and load balancing constraints specified by the administrator.

We then introduce three novel algorithms for generating an efficient migration plan. The first is a greedy algorithm that is inspired by the greedy iterative process in [28]. Our extended algorithm distributes the data evenly between volumes while ensuring that the migration traffic does not exceed the maximum allocation. By breaking this process into several phases, we ensure that the allocated traffic is used for both load balancing and capacity reduction, balancing between the two (possibly conflicting) goals.

Our second algorithm is inspired by the ILP-based approach of GoSeed. GoSeed solves the seeding problem, whose single natural minimization objective is the system size. In contrast, our new algorithm addresses the inherently competing objectives (size, balance, traffic) of general migration. We reformulate the ILP problem with variables and constraints that express the traffic used during migration and the choice of volumes from which to remap files or to remap files onto. Our formulation for the general migration problem is naturally much more complex than the one required for seeding. Nevertheless, we successfully applied it to data migration in systems with hundreds of millions of blocks.

Our third algorithm is based on hierarchical clustering, which, to the best of our knowledge, has not been applied to data deduplication before. We group similar files into clusters, where the target number of clusters is the number of volumes in the system. We incorporate the physical location of the files into the clustering process, such that the similarity between files expresses the blocks that they share as well as their initial locations. Clusters are assigned to volumes according to the blocks already stored on them, and the migration plan remaps each file to the volume assigned to its cluster.

We implemented our three algorithms and evaluated them on six system snapshots created from three public datasets [6, 10, 38]. Our results demonstrate that all algorithms can successfully reduce the system's size while complying with the traffic and load balancing constraints. Each algorithm has different advantages: the greedy algorithm produces a migration plan in the shortest runtime (often several seconds), although its reduction in system size is typically lower than that of the other algorithms. The ILP-based approach can efficiently utilize the allowed traffic consumption, and improve as the load balancing constraints are relaxed. However, its execution must be timed out on the large problem instances, which often prevents it from yielding an optimal migration plan. The clustering algorithm empirically achieves comparable results to those of the ILP-based approach, and sometimes even exceeds them. It does so in much shorter runtimes.

We summarize our main contributions as follows. We formulate the general migration with deduplication as an optimization problem (§ 3), and design and implement three algorithms for generating general migration plans: the greedy (§ 4) and ILP-based (§ 5) approaches are inspired by previous studies, while the clustering-based (§ 6) approach is entirely novel. We methodologically compare our algorithms to analyze the advantages and limitations of each approach (§ 7).

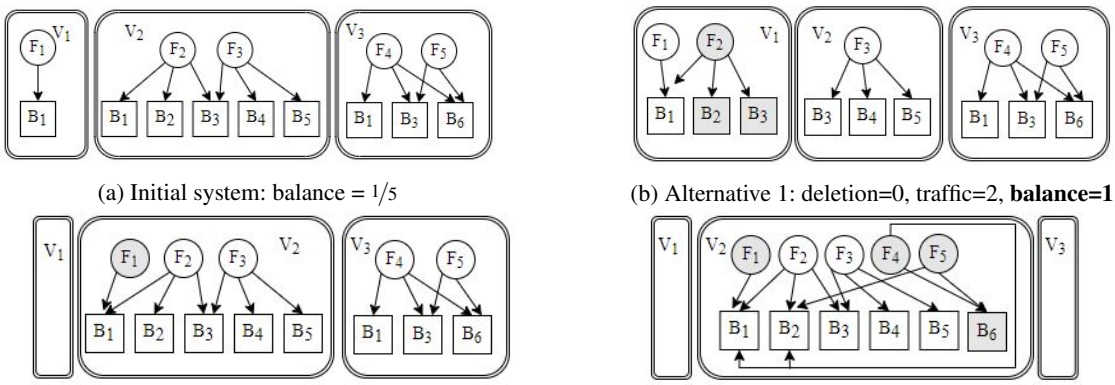
2 Background and related work

Data deduplication. In a nutshell, the deduplication process splits incoming data into fixed or variable-sized chunks, which we refer to as *blocks*. The content of each block is hashed to create a *fingerprint*, which is used to identify duplicate blocks and to retrieve their unique copy from storage. Several aspects of this process must be optimized so as not to interfere with storage system performance. These include chunking and fingerprinting [11,36,39,50,51], indexing and lookups [12,45,54], efficient storage of blocks [17, 19, 31, 33, 34, 45, 52], and fast file reconstruction [24, 30, 32, 53]. Although the first commercial systems used deduplication for backup and archival data, deduplication is now commonly used in high-end primary storage.

Data migration in distributed deduplication systems. Numerous distributed deduplication designs were introduced in commercial and academic studies [18, 22, 27]. We focus on designs that employ a separate fingerprint index in each physical server [15, 16, 20, 21, 28]. This design choice maintains a small index size and a low lookup cost, facilitates garbage collection at the server level, and simplifies the client-side logic. In this design, each server (*volume*) is a separate *deduplication domain*, i.e., duplicate blocks are identified only within the same volume. Recipes of files mapped to a specific volume thus point to blocks that are physically stored in that volume.

Deduplicated systems are different from traditional distributed systems in that striping files across volumes might reduce deduplication, even if it is done using a content-based chunking algorithm. Splitting files across a cluster also complicates garbage collection. Moreover, many storage systems (e.g., in DataDomain [23] and IBM [28]) are organized as a collection of independent clusters or "islands" of storage in the data center or across data centers. Deduplication is performed within each independent subsystem, but files might be migrated between the different appliances or clusters as a means to re-balance the entire system's utilization.

For example, if a subsystem becomes full while another subsystem has available capacity, migration is quicker and cheaper than adding capacity to the full subsystem. Existing mechanisms migrate files efficiently by transferring only the files' metadata and the chunks that are not already present in the target subsystem [23]. Monthly migration aligns with average



(a) Initial system: balance = 1/5

(b) Alternative 1: deletion=0, traffic=2, **balance=1**

(c) Alternative 2: deletion=1/9, **traffic=0**, balance=0

(d) Alternative 3: **deletion=**3/9, traffic=1, balance=0

Figure 1: Initial system (a) and alternative migration plans: with optimal balance (b), optimal traffic (c), and optimal deletion (d). All the blocks in the system are of size 1.

retention period which is seen for typical backup customers.

The coupling of the logical file's location and the physical location of its blocks implies that when a file is remapped from its volume, we must ensure that all its blocks are stored in the new volume. At the same time, the file's blocks cannot necessarily be removed from its original volume, because they might also belong to other files. For example, consider the initial system depicted in Figure 1(a), and assume we remap file F_2 from volume V_2 to volume V_1, resulting in the alternative system in Figure 1(b). Block B_1 is deleted from V_2 because it is already stored in V_1. Block B_2 is deleted from V_2, but must be copied to V_1, because it wasn't there in the initial system. Block B_3 must also be copied to V_1, but is not deleted from V_2 because it also belongs to F_3. The total sizes of the initial system and of this alternative are the same: nine blocks.

Existing approaches. Harnik et al. [28] presented a greedy iterative algorithm for reducing the total capacity of data in a system with multiple volumes. In each iteration, one file is remapped to a new volume, and the process continues until the total capacity is reduced by a predetermined deletion goal.

GoSeed [40] addresses a simplified case of data migration called *seeding*, where the initial system consists of many files mapped to a single volume. The migration goal is to delete a portion of this volume's blocks by remapping files to an empty target volume [23]. GoSeed formulates the seeding problem as an ILP (integer linear programming) instance whose solution determines which files are remapped, which blocks are *moved* from the source volume to the target, and which are *replicated* to create copies on both volumes. This approach is made possible by the existence of open-source [4, 5, 9] and commercial [2, 3] ILP-solvers—heuristic-based software tools for solving this NP-hard problem efficiently. GoSeed is applied to instances with millions of blocks with several acceleration heuristics, some of which we adapt to the generalized problem.

Rangoli [41] is a greedy algorithm for *space reclamation*—another specific case of data migration where a set of files is chosen for deletion in order to delete a portion of the system's physical size. Unlike the greedy and ILP-based approaches that inspire our own algorithms, the problem solved by Rangoli is too simplified for it to be extended for general migration. Shilane et al. [44] discuss additional data migration scenarios and their resulting complexities in deduplicated systems.

3 Motivation and problem statement

Minimizing migration traffic. High-performance storage systems typically limit the portion of their network bandwidth that can be used for maintenance tasks such as reconstruction of data from failed storage nodes [29, 43]. Data migration naturally involves significant network bandwidth consumption, and traditional data migration plans and mechanisms strive to minimize their network requirements as one of their optimization goals [13, 14, 23, 35, 37, 48]. In this work, we focus on the amount of data that is moved between nodes. The physical layout of the nodes and the precise scheduling of the migration are outside the scope of our current work.

In deduplicated storage, we distinguish between two costs associated with data migration. The *migration traffic* is the amount of data that is transferred between volumes during migration. The *replication cost* is the total size of duplicate blocks that are created as a result of remapping files to new volumes. Previous studies of data migration in deduplicated systems did not consider bandwidth explicitly. Harnik et al. [28] did not address this aspect at all. In the seeding problem addressed by GoSeed [40], the migration traffic is implicitly minimized as a result of minimizing the replication cost. In the general case, however, migration traffic is potentially independent of the amount of data replication.

For example, Alternative 1 in Figure 1(b) results in transferring two blocks, B_2 and B_3, between volumes, even though B_2 is eventually deleted from its source volume. In contrast, the alternative migration plan in Figure 1(c) does not consume traffic at all: file F_1 is remapped to V_2 which already stores its only block, and thus B_1 can simply be deleted from V_1. This alternative also reduces the system's size to eight blocks, making it superior to the first alternative in terms of both objectives. We note, however, that this is not always the case, and

that minimizing the overall system size and minimizing the amount of data transferred might be conflicting objectives.

Load Balancing. Load balancing is a major objective in distributed storage systems, where it often conflicts with other objectives such as utilization and management overhead [14, 42, 49]. Distributed deduplication introduces an inherent tradeoff between minimizing the total physical data size and maximizing load balancing: the system's size is minimized when all the files are mapped to a single volume, which clearly gives the worst possible load balancing. Thus, distributed deduplication systems weigh the benefit of mapping a file to the volume that contains similar files, against the need to distribute the load evenly between the volumes. Load balancing can refer to various measures of load, such as IOPS, bandwidth requirements, or the number of files mapped to each volume.

We follow previous work and aim to evenly distribute the *capacity load* between volumes [16, 20]. Balancing capacity is especially important in deduplicated systems that route incoming files to volumes that already contain similar files. In such designs, volumes whose storage occupancy is slightly higher than others might quickly become overloaded due to their larger amount of data 'attracting' even more new files, and so on. Capacity load balancing can be expected to lead to better network utilization and prevent specific volumes from becoming a bottleneck or exhausting their capacity. While performance load balancing is not our main objective, we expect it to improve as a result of distributing capacity. All our approaches can be extended to address it explicitly.

In this work, we capture the load balancing in the system with the *balance* metric, which is similar to a commonly used *fairness* metric [25]—the ratio between the size of the smallest volume in the system and that of the largest volume. For example, the balance of the initial system in Figure 1(a) is $|V_1|/|V_2| = 1/5$. Alternative 1 (Figure 1(b)) is perfectly balanced, with $balance = 1$, while Alternative 2 (Figure 1(c)) has the worst balance: $|V_1|/|V_2| = 0$.

Problem statement. There are various approaches for handling conflicting objectives in complex optimization systems. These include searching for the Pareto frontier [55], or defining a composite objective function of weighted individual objectives. We chose to keep the system's size as our main objective, and to address the migration traffic and load balancing as constraints on the migration plan. We define the general migration problem by extending the seeding problem from [40], and thus we reuse some of their notations for compatibility.

For a storage system S with a set of volumes V, let $B = \{b_0, b_1, \ldots\}$ be the set of unique blocks stored in the system, and let $size(b)$ be the size of block b. Let $F = \{f_0, f_1, \ldots\}$ be the set of files in S, and let $I_S \subseteq B \times F \times V$ be an inclusion relation, where $(b, f, v) \in I_S$ means that file f mapped to volume v contains block b which stored in this volume. We use $b \in v$ to denote that $(b, f, v) \in I_S$ for some file f. The size of a volume is the total size of the blocks stored in it, i.e., $size(v) = \Sigma_{b \in v} size(b)$. The size of the system is the total size of its volumes, i.e., $size(S) = size(V) = \Sigma_{v \in V} size(v)$.

The *general migration problem* is to find a set of files $F_M \subseteq F$ to migrate, the volume each file is migrated to, the blocks that can be deleted from each volume, and the blocks that should be copied to each volume. Applying the migration plan results in a new system, S'. The *migration goal* is to minimize the size of S'. This is equivalent to maximizing the size of all the blocks that can be deleted during the migration, minus the size of all the blocks that must be replicated.

The *traffic constraint* specifies T_{max}—the maximum traffic allowed during migration. It requires that the total size of blocks that are added to volumes they were not stored in is no larger than T_{max}. The *load balancing constraint* determines how evenly the capacity is distributed between the volumes. It specifies a *margin* $0 \leq \mu < 1$ and requires that the size of each volume in the new system is within μ of the average volume size. For a system with $|V|$ volumes, this is equivalent to requiring that $balance \leq \lceil size(S')/|V| \times (1-\mu) \rceil / \lceil size(S')/|V| \times (1+\mu) \rceil$.

For example, for the initial system in Figure 1(a), Alternative 1 (Figure 1(b)) is the only migration plan that satisfies the load balancing constraint (for any μ). For T_{max} lower than $2/9$, no migration is feasible. On the other hand, if we remove the load balancing constraint, the optimal migration plan depends on the traffic constraint alone: Alternative 2 (Figure 1(c)) is optimal for, e.g., $T_{max} = 0$, and Alternative 3 (Figure 1(d)) is optimal for $T_{max} = 3$.

Refinements. This generalized problem can be refined in several straightforward ways. For example, we can restrict the set of files that may be included in F_M, the set of volumes from which files may be removed, or the set of volumes to which files can be remapped. Similarly, we can require that a specific volume be removed from the system (enforcing all its files to be remapped), or that an empty volume be added. We demonstrate some of these cases in our evaluation.

4 Greedy

The basic greedy algorithm by Harnik et al. [28] iterates over all the files in each volume, and calculates the *space-saving ratio* from remapping a single file to each of the other volumes: the ratio between the total size of the blocks that would be replicated and the blocks that would be deleted from the file's original volume. In each iteration, the file with the lowest ratio is remapped. For example, if this basic greedy algorithm was applied to the initial system in Figure 1(a), it would first remap file F_1 to volume V_2, with a space-saving ratio of 0, resulting in Alternative 2 (Figure 1(c)). The process halts when the total capacity is reduced by a predetermined deletion goal. This algorithm is not directly applicable to the general migration problem because it does not consider traffic and load balancing.

Addressing the traffic constraint is relatively straightforward. In our extended greedy algorithm we make it the halting condition: the iterations stop when there is no file that can be

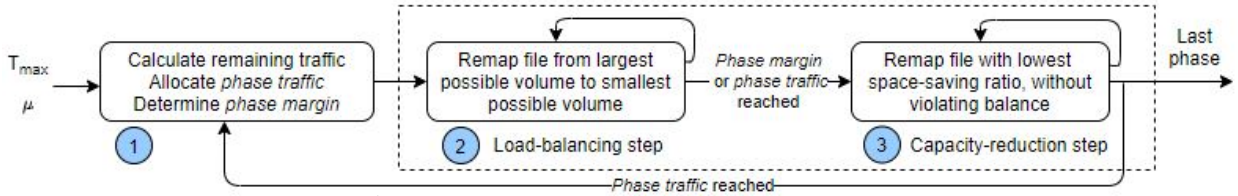


Figure 2: Overview of our extended greedy algorithm.

remapped without exceeding the maximum allocated traffic. A small challenge is that a file might be remapped in several iterations of the algorithm, while, in the resulting migration plan, it will only be remapped from its original volume to its final destination. As a result, the sum of traffic of all the individual iterations can be (and is, in practice) higher than the traffic required when executing migration plan. This will not violate the traffic constraint, but will cause the algorithm to halt before taking advantage of the maximum allowed traffic. Thus, we heuristically allow the algorithm to use 20% more traffic than the original traffic constraint, to prevent it from halting prematurely. The required traffic for the resulting migration plan is calculated before its execution. Thus, if it violates the original traffic constraint, a new plan can be generated by the algorithm without this heuristic. We include this simple extension, without a load-balancing constraint, in our evaluation.

Complying with the load-balancing constraint is more challenging. For example, if the basic greedy algorithm reached Alternative 2 (Figure 1(c)), it could no longer remap any single file to volume V_1 without increasing the system's capacity, and thus the system will remain unbalanced with at least one empty volume. A naive extension to this algorithm could enforce the load-balancing constraint by preventing files from being remapped if this increases the system's imbalance. However, such a strict requirement might preclude too many opportunities for optimization. For example, for the initial system in Figure 1(a), it would only allow to remap file F_2 to volume V_1, resulting in Alternative 1 (Figure 1(b)). The system would be perfectly balanced, but the basic algorithm would then terminate without reducing its size at all.

We address this challenge with two main techniques. The first is defining two iteration types: one whose goal is to balance the system's load, and another whose goal is to reduce its size. We perform these iterations interchangeably, to avoid the entire allocated traffic from being spent on only one goal. The second technique is to relax the load-balancing margin for the early iterations and continuously tighten it until the end of the execution. The idea is to let the early iterations remap files more freely, and to ensure that the iterations at the end of the algorithm result in a balanced system.

Figure 2 illustrates the process of our extended greedy algorithm. We divide the algorithm's process into phases. (1) Each phase is allocated an even portion of the traffic allocated for migration, and is limited by a local load-balancing constraint. Each phase is composed of two steps. (2) The *load-balancing step* iteratively remaps files from large volumes to small ones, until the volume sizes are within the margin defined for this phase, or its traffic is exhausted. (3) The *capacity-reduction step* uses the remaining traffic to reduce the system's size by remapping files between volumes, ensuring that volume sizes remain within the margin.

Each phase is limited by **local traffic and load-balancing constraints**, calculated at the beginning of the phase. The *phase traffic* determines the maximum traffic that can be used in each phase, and is roughly even for all the phases. The local *phase margin* determines the minimum and maximum allowed volume sizes in each phase. It is larger than the global margin, μ, in the first phase, and gradually decreases before each phase, until reaching μ in the last phase. By default, our greedy algorithm consists of $p = 5$ phases. The phase traffic for phase i, $0 \leq i < p$, is $1/(p-i)$ of the unused traffic, and the phase margin for the first phase is $\mu \times 1.5$.

The load balancing step is the first step in each phase. In each of its iterations, the volumes are sorted according to their sizes, and we attempt to remap files from the largest volumes to the small ones. A file can be remapped only if some blocks will be deleted from its source volume as a result. Namely, we look for a file to remap between a $\langle source, target \rangle$ pair of volumes, where *source* is the largest volume and the *target* is the smallest volume for which such a file exists. In each iteration, the amount of traffic required to remap the chosen file is calculated, and the iterations halt when the maximum allowed traffic or allowed volume sizes are reached.

The capacity-reduction step uses the remaining traffic allocation of the phase. It is similar to the original greedy algorithm, but it ensures that each file remap does not cause the volumes to become unbalanced. In other words, we can remap a file only if this does not cause its source volume to become too small, or its target volume to become too large. Note that the amount of traffic that remains for the capacity-reduction step depends on the degree of imbalance in the initial system. In the most extreme case of a highly unbalanced system, it is possible for the load balancing step to consume all the traffic allocated for the phase. In this case, the capacity-reduction step halts in the first iteration. For cases other than this extreme, a higher number of phases can divert more traffic for capacity-reduction, at the cost of longer computation time due to the increased number of iterations.

5 ILP

Our ILP-based approach is inspired by GoSeed [40], designed for the seeding problem, where files can only be remapped

from the source volume to the empty target volume. GoSeed thus defined three types of variables whose assignment specified (1) whether a file is *remapped*, (2) whether a block is *replicated* on both volumes, and (3) whether a block is deleted from the source and *moved* to the target. These limited options resulted in a fairly simple set of constraints, which cannot be directly applied to the general migration problem. The major difference is that the decision of whether or not we can delete a block from its source volume depends not only on the files initially mapped to this volume, but also on the files that will be remapped to it as a result of the migration. Thus, in our ILP-based approach, every block transfer is modeled as creating a copy of this block, and a separate decision is made whether or not to delete the block from its source volume.

The problem's constraints are defined over the set of volumes, files, and blocks from the problem statement in Section 2, the maximum traffic T_{max}, and the load-balancing margin μ. We define the target size of each volume v as w_v, given as percentage of the system size after migration. By default, $w_v = 1/|V|$. For each pair of volumes, v, u, we define their *intersection* as the set of blocks that are stored on both volumes: $Intersect_{vu} = \{b | b \in u \wedge b \in v\}$. The intersections are calculated before the constraints are assigned, and are used in the formulation below for better readability.

The constraints are expressed in terms of three types of variables that denote the actions performed in the migration: x_{fst} denotes whether file f is *remapped* from its source volume s to another (target) volume t. c_{bst} denotes whether block b is *copied* from its source volume s to another (target) volume t. Finally, d_{bv} denotes whether block b is *deleted* from volume v. The solution to the ILP instance is an assignment of 0 or 1 to these variables. The resulting migration plan remaps the set of files for which $x_{fst} = 1$ (for some volume t), transfers the blocks for which $c_{bst} = 1$ to their target volume, and deletes the blocks for which $d_{bv} = 1$ from their respective volumes.

Constraints and objective. The ILP formulation for migration with load balancing consists of 13 constraint types.

1. All Variables are Boolean.
2. A file can be remapped to at most one volume.
3. A block can only be deleted or copied from a volume it was originally stored in.
4. A block can be deleted from a volume only if all the files containing it are remapped to other volumes.
5. A block can be deleted from a volume only if no file containing it is remapped to this volume.
6. View all the blocks in the volume intersections as having been copied.
7. When a file is remapped, all its blocks are either copied to the target volume, or are initially there (as part of the intersection).
8. A block can be copied to a target volume only from one source volume.and volume t, $\Sigma_{s \text{ such that } b \notin Intersect_{st}} c_{bst} \leq 1$.
9. A block must be deleted if there are no files containing it on the volume.
10. *A* block cannot be copied to a target volume if no file will contain it there.
11. A file cannot be migrated to its initial volume.
12. *Traffic constraint*: the size of all the copied blocks is not larger than the maximum allowed traffic.
13. *Load balancing constraint*: for each volume v, $(w_v - \mu) \times Size(S') \leq Size(v') \leq (w_v + \mu) \times Size(S')$, where $Size(v')$ is the volume size after migration, i.e., the sum of its non-deleted blocks and blocks copied to i.

► *Objective*: maximize the sum of sizes of all blocks that are deleted minus all blocks that are copied. This is equivalent to minimizing the overall system size.

Constraints 12 and 13 formulate the traffic and load-balancing goals, and constraints 8, 9, and 10 ensure that the solver does not create redundant copies of blocks to artificially comply with the load balancing constraint. This is similar to the constraint that prevents *orphan* blocks in the seeding problem [40]. For evaluation purposes, we will also refer to a relaxed formulation of the problem without the load-balancing constraint. In that version, constraints 8, 9, 10, and 13 are removed, considerably reducing the problem complexity.

The ILP formulation given in this paper is designed for the most general case of data migration, where any file can be remapped to any volume. In reality, the migration goal might restrict some of the remapping options, potentially simplifying the ILP instance. For example, we can limit the set of volumes that files can be migrated to by eliminating the x_{fst} and c_{bst} variables where t is not in this set. We can similarly restrict the set of volumes files may be migrated from, or require that a set of specific files are (or are not) remapped.

Complexity and run time. The complexity of the ILP instance depends on $|B|$, $|F|$, and $|V|$—the number of blocks, files, and volumes, respectively. The number of variables is $|V|^2|F| + |V|^2|B| + |V| \times |B|$, corresponding to variable types x_{fst}, c_{bst}, and d_{bv}. Each of the constraints defined on these variables contributes a similar order of magnitude. An exception is constraint 13, which reformulates the system size, twice, to ensure that each individual volume's size is within the required margin. Indeed, the relaxed formulation without this constraint is significantly simpler than the full formulation.

We use two of the acceleration methods suggested by GoSeed to address the high complexity of the ILP problem. The first is *fingerprint sampling*, where the problem is solved for a subset of the original system's blocks. This subset (*sample*) is generated by preprocessing the block fingerprints and including only those that match a predefined pattern. Specifically, as suggested in [28], a sample generated with sampling degree k includes only blocks whose fingerprints consist of k leading zeroes, reducing the number of blocks in the problem formulation by $1/2^k$ on average.

The second acceleration method is *solver timeout*, which

halts the ILP solver's execution after a predetermined runtime. As a result, the server returns a feasible solution that is not necessarily optimal. A feasible solution to the ILP problem can be directly translated into a migration plan, i.e., a list of files to migrate and their destination volumes. Thus, even if the solution is not optimal (due to sampling or timeout), the process still produces a valid plan for the original system.

We do not repeat the detailed analysis of the effectiveness of these heuristics, which were shown to be effective in earlier studies. Namely, the analysis of GoSeed showed that most of the solver's progress happens in the beginning of its execution (hence, timing out does not degrade its quality too much), and that it is more effective to reduce the sample size than to run the solver longer on a larger sample, as long as the sampling degree is not higher than $k = 13$. Our experiments with the extended ILP formulation, omitted due to space considerations, confirmed these findings.

6 Clustering

Overview. Clustering is a well-known technique for grouping objects based on their similarity [1]. It is fast and effective, and is directly applicable to our domain: files are similar if they share a large portion of their blocks. Our approach is thus to create clusters of similar files and to assign each cluster to a volume, remapping those files that were assigned to a volume different from their original location. Despite its simplicity, three main challenges ($Ch1 - Ch3$) are involved in applying this idea to the general migration problem.

($Ch1$) **Unpredictable traffic** The traffic required for a migration plan can only be calculated after the clusters have been assigned to volumes. When the clustering decisions are being made, their implications on the overall traffic are unknown and thus cannot be taken into consideration.

($Ch2$) **Unpredictable system size** The load-balancing constraint is given in terms of the system's size after migration. However, this size is required to ensure, during the clustering process, that the created clusters are within the allowed sizes.

($Ch3$) **High sensitivity** The file similarity metric is based on the precise set of blocks in each file. When this metric is applied to a sample of the storage system's fingerprints, it is highly sensitive to the sampling degree and rule.

We address these challenges with several heuristics ($H1 - H4$):

($H1$) **Traffic weight** We define a new similarity metric for files. This metric is a weighted sum of the files' content similarity and a new distance metric that indicates how many source volumes contain files within a cluster. Our algorithm considers files as similar if they contain the same blocks and are mapped to the same source volume. Assigning a higher weight (W_T) to the content similarity will result in a smaller system but higher migration traffic.

($H2a$) **Estimated system size** We further use the weight to estimate the size of the system after migration. We calculate the size of a hypothetical system without duplicates, and predict that higher migration traffic will bring the system closer to this hypothetical optimum.

($H2b$) **Clustering retries** We use the estimated final system size to determine the maximum allowed cluster size. During the clustering process, we stop adding files to clusters that reach this size. If the process halts due to this limitation, we increase the maximum size by a small margin, and restart it.

($H3$) **Randomization** Instead of deterministic clustering decisions, we choose a random option from the set of best options. Different random seeds potentially result in different systems.

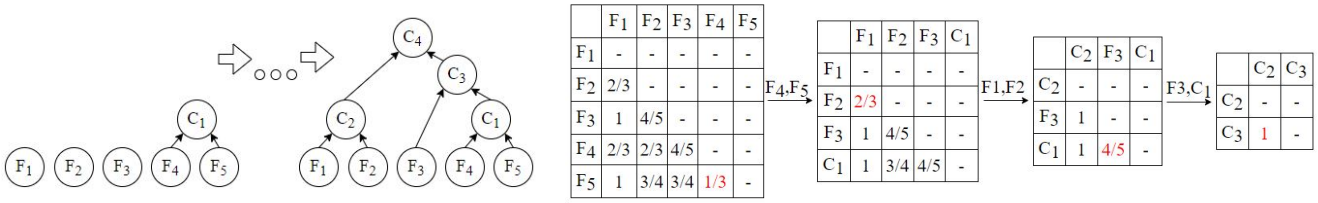
($H4$) **Multiple executions** Our heuristics introduce several parameters which we would be loath to overfit. We use the same initial state to perform repeated executions of the clustering process with multiple sets of parameter combinations (180 in our case), and choose the best migration plan from those executions as our final output.

In the following, we give the required background on the clustering process and describe each of our optimizations in detail.

Hierarchical clustering. *Hierarchical clustering* [26] is an iterative clustering process that, in each iteration, merges the most similar pair of clusters into a new cluster. The input is an initial set of objects, which are viewed as clusters of size 1. The process creates a tree whose leaves are the initial objects, and internal nodes are the clusters they are merged into. For example, Figure 3 shows the clustering hierarchy created from the set of initial objects $\{F_1,...,F_5\}$, where the clusters $\{C_1,...,C_4\}$ were created in order of their indices.

Hierarchical clustering naturally lends itself to grouping of files. Intuitively, files that share a large portion of their blocks are similar and should thus belong to the same cluster and eventually to the same volume. For example, the initial objects in Figure 3 represent the files in Figure 1(a): F_4 and F_5 share two blocks and are thus merged into the first cluster, C_1. Our clustering-based approach is simple: we group the files into a number of clusters equal to the number of volumes in the system and assign one cluster to each volume. This assignment implies which files should be remapped and which blocks should be transferred and/or deleted in the migration. For example, for a system with three volumes, we could halt the clustering process in Figure 3, resulting in a final set of three clusters: $\{C_1, C_2, F_3\}$. We develop this basic approach to the general migration problem, i.e., to maximize the deletion and to comply with the traffic and load-balancing constraints.

File similarity. The hierarchical clustering process relies on a similarity function that indicates which pair of clusters to merge in each iteration. We use the commonly used *Jaccard index* [26] for this purpose. For two sets A and B, their index is defined as $J(A,B) = {|A \cap B|}/{|A \cup B|}$. We view each file as a set



	F1	F2	F3	F4	F5
F1	-	-	-	-	-
F2	2/3	-	-	-	-
F3	1	4/5	-	-	-
F4	2/3	2/3	4/5	-	-
F5	1	3/4	3/4	1/3	-

	F1	F2	F3	C1
F1	-	-	-	-
F2	2/3	-	-	-
F3	1	4/5	-	-
C1	1	3/4	4/5	-

	C2	F3	C1
C2	-	-	-
F3	1	-	-
C1	1	4/5	-

	C2	C3
C2	-	-
C3	1	-

Figure 3: Hierarchical clustering with the files from Figure 1 (left) and the distance matrices created in the process (right).

of blocks, and thus, the Jaccard index for a pair of files is the portion of their shared blocks. From hereon, we refer to the complement of the index: the *Jaccard distance* which is defined as $dist_J = \overline{J(A,B)} = 1 - J(A,B)$. This is to comply with the standard terminology in which the two clusters with the smallest distance are merged in each iteration. For example, the leftmost table in Figure 3 depicts the *distance matrix* for the files in Figure 1. Indeed, the distance is smallest for the pair F_4 and F_5 which are the first to be merged.

The Jaccard distance could easily be applied to entire clusters, which can themselves be viewed as sets of blocks. However, calculating the distance between each new cluster and all existing clusters would require repeated traversals of the original file recipes in each iteration. This complexity is addressed in hierarchical clustering by defining a *linkage* function, which determines the distance between the merged cluster and existing clusters based on the distances before the merge. Specifically, we use *complete linkage*, defined as follows: $dist_J(A \cup B, C) = max\{dist_J(A,C), dist_J(B,C)\}$. For example, the row for C_1 in the second distance matrix in Figure 3 lists the distances between C_1 and each of the remaining files.

Traffic considerations $(H1)$**.** We limit the traffic required by our migration plan in two ways. The first is by assigning each of the final clusters to the volume that contains the largest number of its blocks. We calculate the size of the intersection (in terms of the size of the shared blocks) between each cluster and each volume in the initial system. We then iterativly pick the $\langle cluster, volume \rangle$ pair with the largest intersection from the clusters and volumes that have not yet been assigned.

This assignment alone might still result in excessive traffic, especially if highly similar files are initially scattered across many different volumes. To avoid such situations, we incorporate the traffic considerations into the clustering process itself. Namely, we define the *volume distance*, $dist_V(C)$, of a cluster as the portion of the system's volumes whose files are included in the cluster. For example, in Figure 3, $dist_V(C_1) = 1/3$ and $dist_V(C_2) = 2/3$.

We then define a new *weighted distance* metric that combines the Jaccard distance and the volume distance: $dist_W(A,B) = W_T \times dist_J(A,B) + (1 - W_T) \times dist_V(A \cup B)$, where $0 \leq W_T \leq 1$ is the *traffic weight*. Intuitively, increasing W_T increases the amount of traffic allocated for the migration, which increases the priority of deduplication efficiency over the network transfer cost. Nevertheless, it does not guarantee compliance with a specific traffic constraint. We address this limitation by multiple executions, described below.

Load-balancing considerations $(H2)$**.** We enforce the load balancing constraint by preventing merges that result in clusters that exceed the maximal volume size. We determine the maximal cluster size by estimating the system's size *after* migration. Intuitively, we expect that increasing the traffic allocated for migration will increase the reduction in system size, and we estimate this traffic with the W_T weight described above. Formally, we estimate the size of the final system as $Size(W_T) = W_T \times Size_{uniq} + (1 - W_T) \times size(S_{init})$, where $Size_{uniq}$ is the size of all the unique blocks in the system. The maximal cluster size is thus $C_{max} = {Size(W_T)}/{|V|}$

In each clustering iteration, we ensure that the merged cluster is not larger than C_{max}. This requirement might result in the algorithm halting before the target number of clusters is reached, due to merging decisions made earlier in the process. If this happens, we increase the value of C_{max} by a small ε and retry the clustering process. We continue retrying until the algorithm creates the required number of clusters. A small ε can potentially yield the most balanced system, but might require excessively many retries. We use $\varepsilon = 5\%$ as our default.

Sensitivity to sample $(H3)$**.** As in the ILP-based approach, we apply the hierarchical clustering process to a sample of the system, rather than to the complete set of blocks which can be excessively large. However, it turns out that the Jaccard distance is highly sensitive to the precise set of blocks that represent each file in the sample. We found, in our initial experiments, that different sampling degrees as well as different sampling rules (e.g., k leading ones instead of k leading zeroes in the fingerprint) result in small differences in the Jaccard distance of the file pairs.

Such small differences might change the entire clustering hierarchy, even if the practical difference between the pairs of files is very small. Thus, rather than merging the pair of clusters with the smallest distance, we merge a *random* pair from the set of pairs with the smallest distances. We include in this set only pairs whose distance is within a certain percentage of the minimum distance. Thus, for a maximum distance difference gap, we choose a random pair $\langle C_i, C_j \rangle$ from the 10 (or less) pairs for which $Dist_W(C_i, C_j) \leq$ minimum distance $\times (1 + gap)$.

Constructing the final migration plan $(H4)$**.** The main advantage of our clustering-based approach is its relatively fast runtime. Constructing the initial distance matrix for the individual files is time consuming, but the same initial matrix can be reused for all the consecutive clustering processes on the same initial system. We exploit this advantage to eliminate the sensitivity of our clustering process to the many parame-

ters introduced in this section. For the same given system and migration constraints, we execute the clustering process with six traffic weights ($W_T \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$), three gaps ($gap \in \{0.5\%, 1\%, 3\%\}$), and ten random seeds. This results in a total of 180 executions, some of which are performed in parallel (depending on the resources of the evaluation platform). We calculate the deletion, traffic, and balance of each migration plan (on the sample used as the input for clustering), and as our final result, use the plan with the best deletion that satisfies the load-balancing and traffic constraints.

We also include in our evaluation a relaxed scheme without the load-balancing constraint (i.e., $C_{max} = \infty$). In this scheme, the final migration plan must only satisfy the traffic constraint.

7 Evaluation

We wish to answer two main questions: (1) how do the algorithms compare in terms of the final system size, load balancing, and runtime? and (2) how is the performance of the different algorithms affected by the various system and problem parameters? In the following, we describe our evaluation setup and the experiments executed to answer those questions.

7.1 Experimental Setup

We ran our experiments on a server running Ubuntu 18.04.3, equipped with 128GB DDR4 RAM (with 2666 MHz bus speed), Intel® Xeon® Silver 4114 CPU (with hyper-threading functionality) running at 2.20GHz, one Dell®T1WH8 240GB TLC SATA SSD, and one Micron 5200 Series 960GB 3D TLC NAND Flash SSD.

File system snapshots. We used two static file system snapshots from datasets used to evaluate the seeding problem [40]: The UBC dataset [7, 38] includes file systems of 857 Microsoft employees, of which we used the first 500 file systems (UBC-500). The FSL dataset [10] consists of snapshots of students' home directories at the FSL Lab at Stony Brook University [46, 47]. We used nine weekly snapshots of nine users between August 28 and October 23, 2014 (Homes). These snapshots include, for each file, the fingerprints of its chunks and their sizes. Each snapshot file represents one entire file system, which is the migration unit in our model, and is represented as one file in our migration problem instances.

We created two additional sets of snapshots from the Linux version archive [6]. Our Linux-all dataset includes snapshots of all the versions from 2.0 to 5.9.14. We also created a smaller dataset, Linux-skip, which consists of every 5th snapshot. The latter dataset is logically (approximately) 5× smaller than the former, although their physical size is almost the same.

The UBC-500 and Homes snapshots were created with variable-sized chunks with Rabin fingerprints, whose specified average chunk size is 64KB. We created the Linux snapshots with an average chunk size of 8KB, because they are much smaller to begin with. We used these sets of snapshots to create six initial systems, with varying numbers of volumes. They are listed in Table 1. We emulate the ingestion of each snapshot into a simplified deduplication system which detects duplicates only within the same volume. In the UBC and Linux systems we assigned the same number of arbitrary snapshots to each volume. In the Homes-week system, we assigned snapshots from the same week to the same volume, such that each volume contains all the users' snapshots from a set of three weeks. In the Homes-user system, we assign each user to a dedicated volume such that each volume contains all the weekly snapshots of a set of three users.

System	Files	$\lvert V\rvert$	Chunks	Dedupe	Logical
UBC-500	500	5	382M	0.39	19.5 TB
Homes-week	81	3	19M	0.38	8.9 TB
Homes-user	81	3	19M	0.16	8.9 TB
Linux-skip	662	5 / 10	1.76M	0.12 / 0.19	377 GB
Linux-all	2703	5	1.78M	0.03	1.8 TB

Table 1: System snapshots in our evaluation. $|V|$ is the number of volumes, Chunks is the number of unique chunks, and Dedupe is the deduplication ratio—the ratio between the physical and logical size of each system. Logical is the logical size.

Implementation. All our algorithms are executed on a sample of the system's fingerprints, to reduce their memory consumption and runtime. We use a sampling degree of $k = 13$ unless stated otherwise. The final system size after migration, as well as the resulting balance and consumed traffic are calculated on the original system's snapshot. We use a calculator similar to the one used in [40]: we traverse the initial system's volumes and sum the sizes of blocks that remain in each volume after migration and those that are added to the volume as a result of it. We experimented with three T_{max} values, 20%, 40%, and 100% of each system's initial size, and three μ values, 2%, 5%, and 10% of the system size after migration.

For our greedy algorithm (*Greedy*), we maintain a matrix where we record, for each block, the number of files pointing to it in each volume. We update this array to reflect the file remap performed in each iteration. To determine the space-saving ratio of each file, we reread its original snapshot file and lookup the counters of its blocks in the array. This is more efficient than keeping the list of each file's blocks in memory. Our Greedy implementation consists of 680 lines of `C++` code.

For our ILP-based algorithm (*ILP*), we use the commercial Gurobi optimizer [3] as our ILP solver, and use its `C++` interface to define our problem instances. We use a two-dimentional array similar to the one used for Greedy to calculate the set of blocks shared by each pair of volumes. We then create the variables and constraints as we process each snapshot file, freeing the original array from the memory before invoking the optimization by Gurobi. Our program for converting the input files into an ILP instance and retrieving the solution from Gurobi consists of 1860 lines of `C++` code. We solve each ILP instance three times, each with a different random seed. The results in this section are the average of the three executions.

For our clustering algorithm (*Cluster*), we create a $|F| \times |B|$ bit matrix to indicate whether each file contains each block,

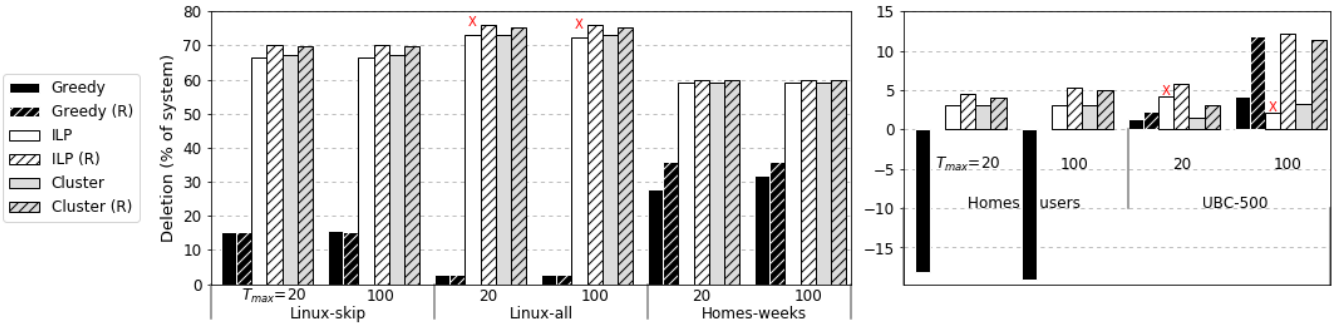


Figure 4: Reduction in system size of all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

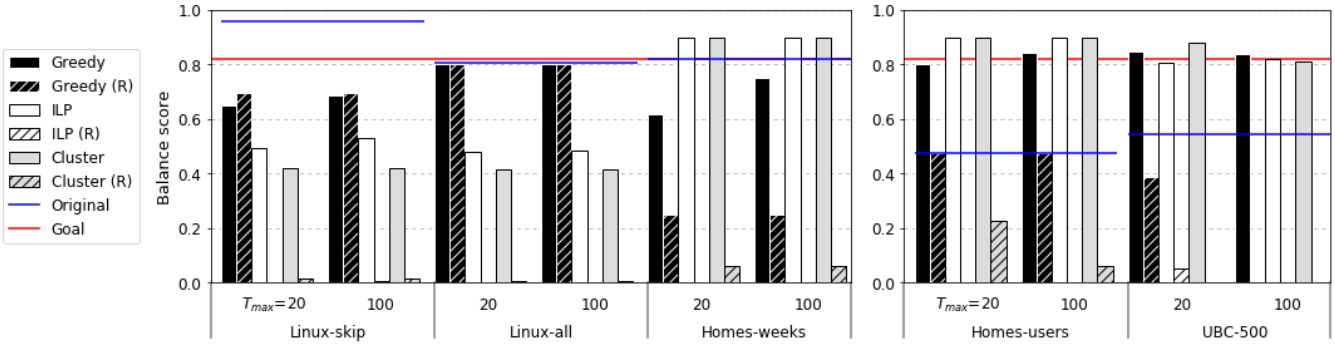


Figure 5: Resulting balance of all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

and use it to create the distance matrix (see Figure 3). The clustering process uses and updates only the lower triangular of this matrix. We use the upper triangular to record the initial distances, and to reset the lower triangular when the clustering process is repeated for the same system and different input parameters (W_T, gap, or random seed). When the clustering process completes, we use the file-block bit matrix to determine the assignment of clusters to volumes. Our program consists of approximately 1000 lines of `C++` code. Each clustering process is performed on a private copy of the distance matrix within a single thread, and our evaluation platform is sufficient for executing six processes in parallel.

Each algorithm has different resource requirements. Greedy is single threaded and requires a simple representation of the system's snapshot in memory. The ILP solver uses as much memory and as many threads as possible (38 in our case). The clustering algorithm ran in six processes, and used approximately 50% of our server's memory. We did not measure CPU utilization directly, although the runtime of the algorithms gives another indication of their compute overheads. Our implementation is open-source and available online [8]

7.2 Basic comparison between algorithms

Figure 4 shows the *deletion*—percentage of the initial system's physical size that was deleted by each algorithm. The deletion is higher for systems that were initially more balanced, i.e., the Linux and Homes-weeks systems. For all the systems except UBC-500, Greedy achieved the smallest deletion. For Homes-users, Greedy increased the system's size in attempt to comply with the load balancing constraint. In UBC-500, there is less similarity and therefore less dependency between files, which eliminates some of the advantage that Cluster and ILP have over Greedy, which outperforms them when $T_{max} = 100\%$.

ILP and Cluster achieve comparable deletions to one another, even though the ILP solver attempts to find the theoretically optimal migration plan. We distinguish between two cases when explaining this similarity. In the first case (Linux-skip and Homes), the ILP-solver produces an optimal solution on the system's sample, but it is not optimal when applied to the full (unsampled) system. The deletion of Cluster is up to 1% higher than that of ILP in those cases. In the second case, marked by a red 'x' in the figures, ILP times out (after six hours in our experiments) and returns a suboptimal solution. Specifically, the complexity of the UBC-500 system demonstrates an interesting limitation of ILP: its deletion with $T_{max} = 20\%$ is higher than with $T_{max} = 100\%$. The reason is that the solution space grows with T_{max}, and thus the best solution found when the solver times out is farther from the optimum.

The 'relaxed' (R) version of the algorithms, without the load balancing constraint, usually achieves a higher deletion than their full version. The largest difference is 558%, although the difference is typically smaller, and can be as low as 1.3%. In the case of Greedy in the Homes-users system, the relaxed version does not identify any file that can be remapped, and does not return any solution.

Figure 5 shows the balance achieved by each algorithm. With a margin of $\mu = 2\%$ and five volumes, the balance should be at least $18/22 = 0.82$. In practice, however, the balance might be lower, for two main reasons. Greedy might fail to bring the

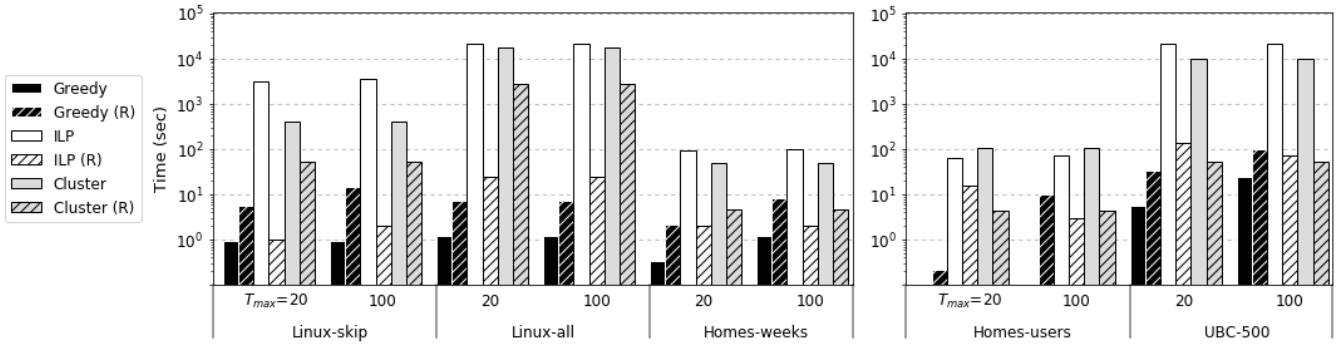


Figure 6: Algorithm runtime for all systems and all algorithms (with and without load balancing constraints. $k = 13$ and $\mu = 2\%$).

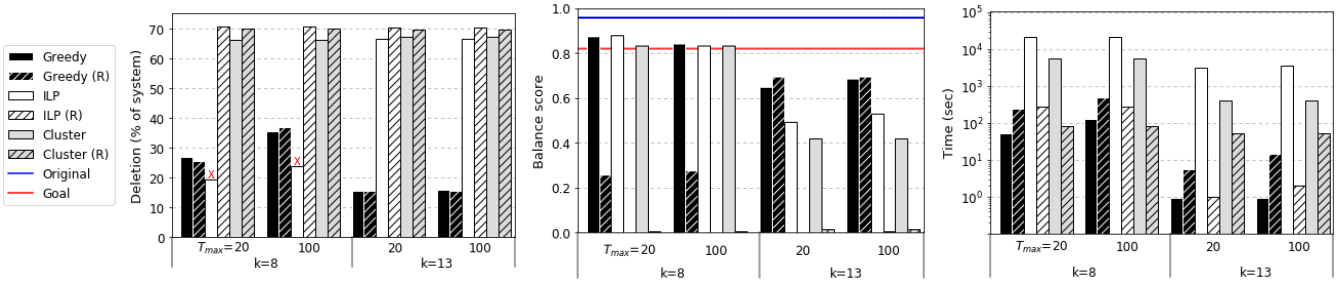


Figure 7: Linux-skip system with 5 volumes, $\mu = 2\%$, and two sampling degrees: $k = 8, 13$.

system to a balanced state if it exhausts (or thinks it exhausts) the maximum traffic allowed for migration. In contrast, Cluster and ILP generate a migration plan that complies with the load balancing constraint on the sample, but violates it when applied to the full (unsampled) system. The violation is highest in the Linux systems, where some files (i.e., entire Linux versions) consist of only one or two blocks. Nevertheless, specifying the load balancing constraint successfully improves the load balancing of the system. Without it, the relaxed Cluster and ILP versions create highly unbalanced systems, with some volumes storing no files at all, or very few small files. Greedy typically avoids such extremes, because it is unable to identify and group similar files in the same volume.

Figure 6 shows the runtime of each of the algorithms (note the log scale of the y-axis). Greedy generates a migration plan in the shortest runtime: 20 seconds or less in all our experiments. ILP requires the longest time, because it attempts to solve an NP-hard problem. Indeed, except for the Homes systems that have the fewest files, ILP requires more than an hour, and often halts at the six-hour timeout. The runtime of Cluster is longer than that of Greedy, and usually shorter than that of ILP. It is still relatively long, as a result of performing 180 executions of the clustering process. We note, however, that this runtime can be shortened by reducing the number of executions, e.g., by reducing the number of random seeds or gaps. We evaluate the effect of these parameters in the following subsection.

Removing the load balancing constraint reduces the runtime of ILP and Cluster by one or two orders of magnitude. In ILP, this happens because the problem complexity is significantly reduced. In Cluster, the clustering is completed in a single attempt, without having to restart it due to illegal cluster sizes. Surprisingly, removing this constraint from Greedy increases its run time. The reason is that each iteration in the capacity-reduction step is much longer than those in the load-balancing step, as it examines all possible file remaps between all volume pairs in the system. In the relaxed Greedy version, all the traffic is allocated to capacity savings and thus its runtime increases.

Implications. Our basic comparison leads to several notable observations. (1) Cluster and ILP have a clear advantage over Greedy. This was not the case in previous studies that examined simple cases of migration, i.e., seeding [40] and space reclamation [41]. However, the increased complexity of the general migration problem increases the gap between the greedy and the optimal solutions. (2) Cluster is comparable and might even outperform ILP, despite the premise of optimality of the ILP-based approach. This is a combination of the high complexity of the ILP problem with the ability to execute multiple clustering processes quickly and in parallel. We conclude that hierarchical clustering is highly efficient for grouping similar files, and that our heuristics for addressing the traffic and load balancing constraints are highly effective. (3) In most systems, adding the load balancing constraint limits the potential capacity reduction, but this limit is usually modest, i.e., several percents of the system's size. The extent of this limitation depends on the degree of similarity between files and the balance of the initial system.

7.3 Sensitivity to problem parameters

Effect of sampling degree. Figure 7 shows the deletion, load balancing, and runtime of all the algorithms on two samples of the Linux-skip system. The small and large samples were generated with sampling degrees of $k = 13$ and $k = 8$, respectively.

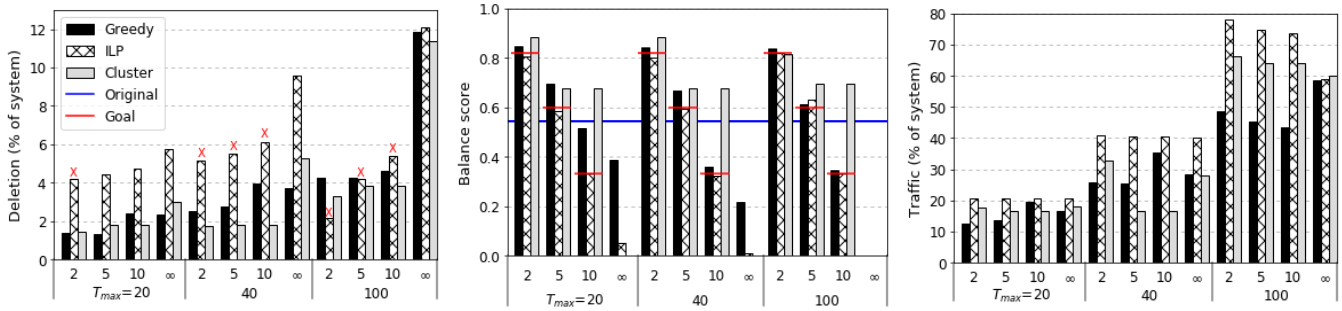


Figure 8: UBC-500 system with $k = 13$ and different load balancing margins.

The sample size affects each algorithm differently. Greedy achieves a higher deletion on the larger sample (by up to 238%), as it identifies more opportunities for capacity reduction. In contrast, ILP suffers from the increase in the problem size: it spends more time on finding a feasible solution and has less time for optimization, and thus its deletion on the larger sample is smaller. We repeated the execution of ILP on the large ($k = 8$) sample with a longer timeout—twelve hours instead of size—but the increase in deletion was minor. This confirmed the observation made for GoSeed [40], that it is more effective to reduce the sample size than to increase the runtime of the ILP solver. The relaxed ILP instance is much simpler, and thus relaxed ILP does not suffer such degradation. Cluster returns similar results for both sample sizes. The differences in the accuracy of the sample are masked by its randomized process.

All the algorithms return a more balanced system for the larger sample ($k = 8$), because the load-balancing constraint is enforced on more blocks, and thus more accurately. At the same time, as we expected, their runtime was higher by several orders of magnitude, as the large sample included $2^5\times$ more blocks than the small one. We note that Greedy is so much faster than ILP and cluster, that its runtime on the large sample is considerably shorter than their runtime on the small one. Thus, if the sample is generated on-the-fly for the purpose of constructing the migration plan, it is possible to execute Greedy on a larger sample for a better migration plan.

Effect of load balancing and traffic constraints. Figure 8 shows the deletion, balance, and traffic consumption of all the algorithms on the UBC-500 system with different values of T_{max} and μ. The results on this system shows the highest sensitivity to these constraints due to the lower similarity between the files. The deletion achieved by all the algorithms increases as T_{max} increases, and their traffic consumption increases accordingly. Removing the load-balancing constraint also allows for more deletion, as we observed in Figure 4. At the same time, the precise value of the load balancing margin, μ, has a much smaller effect on the achieved deletion, although in most cases, a lower margin does guarantee a more balanced system. Increasing the margin increases the runtime (not shown) of Greedy, as a result of more space-reduction iterations, as discussed above. The runtime of ILP and Cluster is not affected by the precise value of μ.

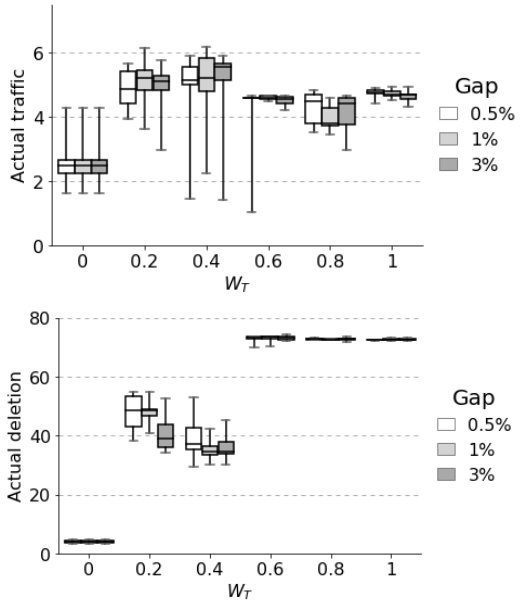


Figure 9: The distribution of migration traffic (top) and reduction in system size (bottom) in the set of plans returned by Cluster for Linux-all with $k = 13$.

Effect of randomization on Cluster. Figure 9 shows the range of deletion values and traffic usage of the migration plans generated by Cluster for Linux-all with $k = 13$. Each bar shows the 25th, 50th, and 75th percentiles, and the whiskers show the minimum and maximum values achieved with different random seeds for each combination of *gap* and W_T. Recall that Cluster picks the plan with the highest deletion that complies with the traffic and load-balancing constraints.

Our results show that different random seeds can result in large differences in the deletion and traffic: up to 84% and 400%, respectively, when W_T and *gap* are fixed. At the same time, W_T cannot predict the actual traffic used by the migration plan, which is why we repeat the clustering process for a range of values. Indeed, different W_T values result in very different deletions. For a given W_T, the range of deletion and traffic values generated by different *gaps* are similar. Thus, as no *gap* consistently outperforms the others, executing the clustering with one or two gaps instead of three will likely have a limited effect on the results while significantly reducing the runtime.

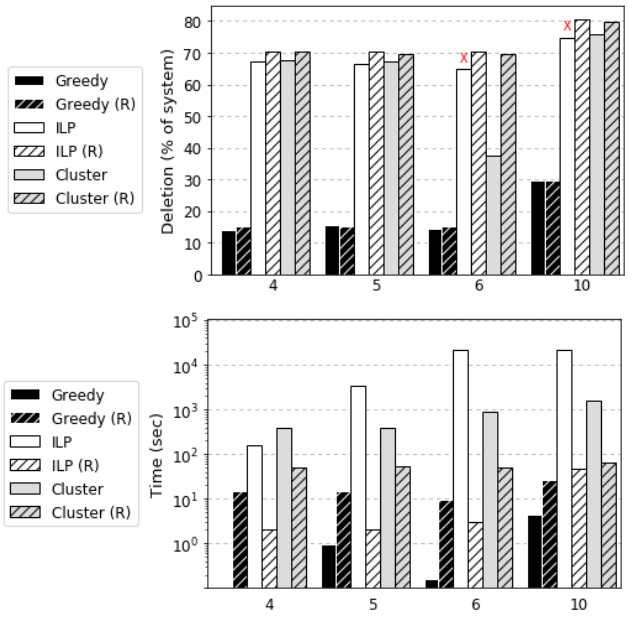


Figure 10: Linux-skip with different numbers of target volumes with $T_{max} = 100, k = 13, \mu = 2\%$.

We compared these results to final plans generated from 5 and 15 seeds (90 and 270 runs, respectively). The comparison, omitted due to space considerations, showed that using more than 5 random seeds carries diminishing returns. Thus, in practice, it is possible to halt the algorithm when additional runs do not improve the best solution so far.

Effect of the number of volumes. Figure 10 shows the deletion and runtime of our algorithms on the Linux-skip system when the number of volumes is reduced ('4'), increased ('6'), or is larger overall ('10'). Due to the high similarity between the Linux versions, the same deletion is achieved when the number of volumes remains five, or when a volume is added or removed (the reduced performance of Cluster is an outlier for $\mu = 2\%$). When the initial number of volumes is 10, there are more duplicates in the system. This provides more opportunities for deletion, which is indeed higher.

The number of volumes affects the problem's complexity, affecting each algorithm differently. Greedy requires less time when a volume is added or removed (compared to a problem where the number of volumes remains the same), because most of its traffic is spent on the faster load-balancing step. The runtime for a system with 10 volumes is much longer than for a system with only five volumes because there are more volume pairs and thus more file remap options to consider in each iteration. The ILP problem complexity increases with every additional volume and thus its runtime increases until it reaches the timeout. The clustering process could, potentially, stop at an earlier stage when more clusters are needed. However, as the number of clusters increases the load balancing constraint is encountered at an earlier stage, causing the clustering to restart more often when the number of volumes is higher. Nevertheless, all our algorithms successfully generated migration plans for a varying number of volumes, most of them within less then an hour.

8 Conclusions and Future Challenges

We formulated the general migration problem for storage systems with deduplication, and presented three algorithms for generating an efficient migration plan. Our evaluation showed that the greedy approach is the fastest but least effective, and that the clustering-based approach is comparable to the one based on ILP, despite ILP's premise of optimality. While the ILP-based approach guarantees a near-optimal solution (given sufficient runtime), clustering lends itself to a range of optimizations that enable it to produce such a solution faster.

All our approaches can be applied to more specific cases of migration, presenting additional opportunities for further optimizations in the future. For example, thanks to its short runtime, we can use Greedy to generate multiple plans with different traffic constraints. These plans are points on the Pareto frontier [55], i.e., they represent different tradeoffs between the conflicting objectives of maximizing deletion and minimizing traffic. The multiple executions in the clustering algorithm provide a similar set of options.

Applying our approach in a live deduplicated system introduces several challenges, such as collecting and generating the system's snapshot as input to the algorithms, efficiently updating the metadata, determining the migration schedule, and adjusting it if new files are added to the system during this process. We leave these challenges for future work.

Acknowledgments

We thank the reviewers and our shepherd, Dalit Naor, for their feedback and suggestions. We thank Aviv Nachman for help with the ILP approach, Nadav Elias for the Linux snapshots, and Danny Harnik for insightful discussions. This research was supported by the Israel Science Foundation (grant No. 807/20).

References

[1] Cluster analysis. https://en.wikipedia.org/wiki/Cluster_analysis. Accessed: 2020-10-24.

[2] CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer. Accessed: 2018-10-24.

[3] The fastest mathematical programming solver. http://www.gurobi.com/. Accessed: 2018-10-24.

[4] GLPK (GNU Linear Programming Kit). https://www.gnu.org/software/glpk/. Accessed: 2018-10-24.

[5] Introduction to lp_solve 5.5.2.5. http://lpsolve.sourceforge.net/5.5/. Accessed: 2018-10-24.

[6] Linux Kernel Archives. https://mirrors.edge.kernel.org/pub/linux/kernel/.

[7] SNIA IOTTA Repository. http://iotta.snia.org/tracetypes/6. Accessed: 2018-10-24.

[8] Source code of migration algorithms. https://github.com/roei217/DedupMigration. Accessed: 2022-02-22.

[9] SYMPHONY development home page. https://projects.coin-or.org/SYMPHONY. Accessed: 2018-10-24.

[10] Traces and snapshots public archive. http://tracer.filesystems.org/. Accessed: 2018-10-24.

[11] Bhavish Aggarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. EndRE: An end-system redundancy elimination service for enterprises. In *7th USENIX Conference on Networked Systems Design and Implementation (NSDI 10)*, 2010.

[12] 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.

[13] Eric Anderson, Joseph Hall, Jason D. Hartline, Michael Hobbs, Anna R. Karlin, Jared Saia, Ram Swaminathan, and John Wilkes. An experimental study of data migration algorithms. In *5th International Workshop on Algorithm Engineering (WAE 01)*, 2001.

[14] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, 2002.

[15] Bharath Balasubramanian, Tian Lan, and Mung Chiang. SAP: Similarity-aware partitioning for efficient cloud storage. In *IEEE Conference on Computer Communications (INFOCOM 14)*, 2014.

[16] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS 09)*, 2009.

[17] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[18] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in SAN cluster file systems. In *2009 Conference on USENIX Annual Technical Conference (USENIX 09)*, 2009.

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

[20] Wei Dong, Fred Douglis, Kai Li, Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[21] 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.

[22] 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 *7th Conference on File and Storage Technologies (FAST 09)*, 2009.

[23] 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.

[24] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[25] Ron Gabor, Shlomo Weiss, and Avi Mendelson. Fairness enforcement in switch on event multithreading. 4(3):15–es, September 2007.

[26] Michael Greenacre and Raul Primicerio. *Hierarchical Cluster Analysis*. Fundación BBVA, Bilbao, 2013.

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

[28] 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.

[29] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[30] 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 (SYSTOR 12)*, 2012.

[31] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[32] 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.

[33] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *7th Conference on File and Storage Technologies (FAST 09)*, 2009.

[34] Xing Lin, Guanlin Lu, Fred Douglis, Philip Shilane, and Grant Wallace. Migratory compression: Coarse-grained data reordering to improve compressibility. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.

[35] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: Online data migration with performance guarantees. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, 2002.

[36] Udi Manber. Finding similar files in a large file system. In *USENIX Winter 1994 Technical Conference (WTEC 94)*, 1994.

[37] Keiichi Matsuzawa, Mitsuo Hayasaka, and Takahiro Shinagawa. The quick migration of file servers. In *11th ACM International Systems and Storage Conference (SYSTOR 18)*, 2018.

[38] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[39] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *18th ACM Symposium on Operating Systems Principles (SOSP 01)*, 2001.

[40] Aviv Nachman, Sarai Sheinvald, Ariel Kolikant, and Gala Yadgar. GoSeed: Optimal seeding plan for deduplicated storage. *ACM Trans. Storage*, 17(3), August 2021.

[41] P. C. Nagesh and Atish Kathpal. Rangoli: Space management in deduplication environments. In *6th International Systems and Storage Conference (SYSTOR 13)*, 2013.

[42] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[43] 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 *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13)*, 2013.

[44] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnala. 99 deduplication problems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[45] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, 2012.

[46] Zhen Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, and Erez Zadok. A long-term user-centric analysis of deduplication patterns. In *32nd Symposium on Mass Storage Systems and Technologies (MSST 16)*, 2016.

[47] 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.

[48] Nguyen Tran, Marcos K. Aguilera, and Mahesh Balakrishnan. Online migration for geo-distributed storage systems. In *2011 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[49] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *ACM/IEEE Conference on Supercomputing (SC 06)*, 2006.

[50] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79:258 – 272, 2014. Special Issue: Performance 2014.

[51] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[52] Zhichao Yan, Hong Jiang, Yujuan Tan, and Hao Luo. Deduplicating compressed contents in cloud storage environment. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[53] zhichao Cao, Hao Wen, Fenggang Wu, and David H.C. Du. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.

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

[55] Eckart Zitzler, Joshua Knowles, and Lothar Thiele. *Quality Assessment of Pareto Set Approximations*, pages 373–404. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

Appendix

Formal formulation of constraints and objective. The ILP formulation for migration with load balancing consists of 13 constraint types.

1. All Variables are Boolean: $x_{fst}, c_{bst}, d_{bv} \in \{0,1\}$
2. A file can be remapped to at most one volume: for every file f in volume s, $\sum_{t \in V} x_{fst} \leq 1$.
3. A block can only be deleted or copied from a volume it was originally stored in: for every two volumes s,t; if $b \notin s$ then $c_{bst} = d_{bs} = 0$.
4. A block can be deleted from a volume only if all the files containing it are remapped to other volumes: for every volume s and for every file f such that $f \in s$, $d_{bs} \leq \sum_t x_{fst}$.
5. A block can be deleted from a volume only if no file containing it is remapped to this volume: for every two volumes s,t, every file f such that $f \in s$ and $f \notin t$, and every block b such that $(b,f,s) \in I_S$, $d_{bt} \leq 1 - x_{fst}$.
6. View all the blocks in the volume intersections as having been copied: for every two volumes s,t and for every block $b \in Intersect_{st}$, $c_{ist} = 1$.
7. When a file is remapped, all its blocks are either copied to the target volume, or are initially there (as part of the intersection): for every two volumes s,t and every block b and file f such as $(b,f,s) \in I_S$, $x_{fst} \leq \Sigma_{v \in V} c_{bvt}$.
8. A block can be copied to a target volume only from one source volume: for every block b and volume t, $\Sigma_{s \text{ such that } b \notin Intersect_{st}} c_{bst} \leq 1$.
9. A block must be deleted if there are no files containing it on the volume: for every two volumes s,v and all files $f_s \in s$, $f_v \in v$ and all blocks b where $b \in f_s$ and $b \in f_v$, $d_{bs} \geq 1 - \{\Sigma_{f_s}(1 - \Sigma_v x_{f_s sv}) + \Sigma_{fv}(x_{f_v vs})\}$.
10. A block cannot be copied to a target volume if no file will contain it there: For every volume t and every block $b \notin t$, $\Sigma_s c_{bst} \leq \Sigma_s \Sigma_{f \in s \wedge b \in f} x_{fst}$
11. A file cannot be migrated to its initial volume: for every file f and volume v, $x_{fvv} = 0$
12. *Traffic constraint*: the size of all the copied blocks is not larger than the maximum allowed traffic: $\sum_{s \in V} \sum_{t \in V} \sum_{b \notin Intersect_{st}} c_{bst} \times size(b) \leq T_{max}$.
13. *Load balancing constraint*: for each volume v, $(w_v - \mu) \times Size(S') \leq Size(v') \leq (w_v + \mu) \times Size(S')$, where $Size(v')$ is the volume size after migration, i.e., the sum of its non-deleted blocks and blocks copied to it: $Size(v') = \sum_{b \in v}(1 - d_{bv}) \times Size(b) + \sum_{s \in V, \sum_b \notin Intersect_{sv}} c_{bsv} \times Size(b)$. $Size(S')$ is the size of the system after migration: $Size(S') = \sum_{v \in V} Size(v')$.

- *Objective*: maximize the sum of sizes of all blocks that are deleted minus all blocks that are copied. This is equivalent to minimizing the overall system size: $Max\left(\sum_{b \in B} Size(b) \times \sum_{s \in V} \left[d_{bs} - \sum_{t \in V, b \notin Intersect_{st}} c_{bst}\right]\right)$.

DUPEFS: Leaking Data Over the Network With Filesystem Deduplication Side Channels

Andrei Bacs† Saidgani Musaev† Kaveh Razavi‡ Cristiano Giuffrida† Herbert Bos†

† *VUSec, Vrije Universiteit Amsterdam* ‡ *ETH Zurich*

Abstract

To reduce the storage footprint with increasing data volumes, modern filesystems internally use *deduplication* to store a single copy of a data deduplication record, even if it is used by multiple files. Unfortunately, its implementation in today's advanced filesystems such as ZFS and Btrfs yields timing side channels that can reveal whether a chunk of data has been deduplicated. In this paper, we present the DUPEFS class of attacks to show that such side channels pose an unexpected security threat. In contrast to memory deduplication attacks, filesystem accesses are performed asynchronously to improve performance, which masks any potential signal due to deduplication. To complicate matters further, filesystem deduplication is often performed at large granularities, complicating high-entropy information leakage. To address these challenges, DUPEFS relies on carefully-crafted read/write operations that show exploitation is not only feasible, but that the signal can be amplified to mount byte-granular attacks *over the network*. We show attackers can leak sensitive data at the rate of ∼1.5 bytes per hour in a end-to-end remote attack, to leak a long-lived (critical) OAuth access token from the access log file of the nginx web server running on ZFS/HDD. Finally, we propose mitigations where read/write operations exhibit the same time-domain behavior, irrespective of the pre-existence of the data handled during the operation.

1 Introduction

Modern filesystems such as ZFS [9] and Btrfs [47] rely on deduplication to reduce the storage footprint for achieving scalable storage systems [17, 38, 59]. The idea is both simple and attractive: if two files both contain some data that is exactly the same, we can save storage space by storing the corresponding data once and maintaining shared references for the two files. Superficially, such functionality resembles its memory deduplication counterpart, where operating systems and hypervisors reduce the memory footprint by merging pages with the same content into a single shared copy-on-write (COW) page. Unfortunately, memory deduplication presents security risks as researchers have shown that it is possible to leak even high-entropy data by detecting when memory is shared [7, 10, 22]. In response, cloud providers and operating system vendors have simply disabled memory deduplication to stop these attacks [10, 29]. In contrast, filesystem deduplication is still commonly deployed everywhere [39, 55]. The question we ask in this paper is whether similar or even more significant security risks exist for filesystem deduplication.

At first sight, the answer appears to be an easy *no*. After all, memory and filesystem deduplication may have the same high-level objective and modus operandi, but their behavior is fundamentally different. In particular, filesystem operations tend to be asynchronous for efficiency. As an example, a write to a file, regardless of deduplication, is first absorbed in memory and will not prompt a write to the disk until much later. Asynchronous operations, achieved through many layers of caching, invariably blind any deduplication-related signals. Besides such fundamental differences, filesystem deduplication also differs in other important *practical* aspects. For instance, to reduce overhead, the granularity of filesystem deduplication (often as large as 128 KB) vastly exceeds that of memory deduplication (typically 4 KB). For an attacker, large deduplication granularity requires non-trivial massaging when leaking data at a desired byte granularity. Due to these complexities, state-of-the-art storage-based deduplication attacks are limited to exploiting cloud application-level deduplication for cross-user file fingerprinting [25, 40].

In this paper, we present DUPEFS, a class of attacks showing that, despite these challenges, exploiting inline filesystem deduplication is feasible. To this end, we describe novel primitives to leak data via filesystem deduplication and analyze their properties. Using these primitives, we build a number of DUPEFS attacks, including one leaking arbitrary data at byte granularity. Moreover, we show that we can expand the threat model of such fine-grained attacks from local-only (as done by memory deduplication) to fully remote attacks. This is possible since, in production systems, filesystems are often shared between multiple remote parties, either directly (e.g.,

a shared file server), or indirectly (e.g., when multiple clients cause a web server to log accesses). In addition, access times to filesystem storage are generally higher than accesses to memory, and, as we will show, even amenable to further amplification by an attacker aware of filesystem internals. This enables remote attacks, where a malicious client leaks secret data from another remote victim client *across the network*.

To craft DUPEFS primitives and obtain secret file data of other users, the attacker generates a carefully-chosen sequence of file operations, specifically tailored to the target filesystem's low-level implementation. The attacks rely solely on timing and storage information available to unprivileged users. We demonstrate the practicality of the side channel with concrete attacks against two popular filesystems, ZFS [9] and Btrfs [47], from different vantage points: local (attacker's code on the victim machine), LAN (attacker across the local-area network), and WAN (attacker across the Internet). For instance, we present an end-to-end DUPEFS attack against an nginx web server running on ZFS/HDD during off-hours. In this scenario, we can leak a (critical) long-lived OAuth access token from the server's access log file over LAN or even WAN at a rate of around 1.5 and 1 byte per hour, respectively.

Finally, we discuss possible mitigations. In particular, we propose to drastically reduce the timing side channel by making filesystem operations that interact with the deduplication subsystem pseudo-constant-time. The goal is to eliminate remote exploitability in a practical way, preserving space savings and avoiding a complete filesystem redesign.

Contributions. We make the following contributions:

- We analyze filesystem deduplication side channels and show that despite the asynchronous disk accesses and large deduplication granularities, attackers can mount byte-level data leak attacks across the network.
- We introduce DUPEFS's novel attack primitives and demonstrate their feasibility in end-to-end attacks to leak data even across the Internet.
- We describe and analyze mitigations for such attacks.

2 Background

A filesystem is the operating system component that controls the storage and retrieval of data to and from storage devices. Compared to DRAM, accessing storage devices is slow. To hide the latency, modern filesystems use a number of optimizations. In particular, *deduplication* finds identical copies of data and stores them as a single shared data *record* of predetermined size. The duplicates are replaced by references to the single record and thus the filesystem reduces the data footprint. More broadly, deduplication is a generic optimization that is used at different levels of the storage hierarchy, including caches [35, 54], cloud services [5, 33], and most importantly, directly in the filesystems themselves [30, 45]. In this paper, we are concerned with the latter.

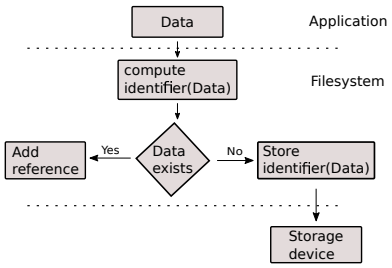


Figure 1: Write path with deduplication.

Basic write workflow. Figure 1 presents a high-level overview of the steps performed by a deduplicating filesystem for an application-issued *write* operation. Upon receiving data from a client application, the filesystem calculates a unique identifier for the data (e.g., by calculating a hash over the content), which it checks against a database of existing identifiers. If none of the existing identifiers match, the new data is written to storage and the identifier added to the deduplication database. If the data exists, the filesystem updates its metadata with a new reference to the existing data and returns control to the application without writing the data to the storage device.

Deduplication mode. In this paper, we are interested in *inline* deduplication, where the filesystem automatically checks for duplicates during the I/O operation—e.g., when data is written. In contrast, offline (or out of band) deduplication is typically a manual process whereby a user runs a deduplication utility explicitly. While inline deduplication introduces some overhead with the identifier lookup, it is the deduplication commonly used in production since, in case of duplication, only a reference is immediately written to storage instead of the duplicate data, leading to space and time savings.

Data identifiers. To identify the content of a deduplication record, the filesystem uses a hash function. Some implementations use collision-resistant cryptographic hash functions such as SHA-256, while others rely on faster hash functions that are not collision-resistant, such as fletcher4 [19]. The same function used for data identification is also used to detect duplicates by computing hashes of candidate data to write and comparing it with the existing deduplication records [16, 31, 45]. Since hashing may incur collisions, some implementations include an additional step to verify that the data inside the matching deduplication records is identical [9].

Deduplication tables. A deduplicating filesystem keeps a history of previously written deduplication records to identify future duplicates. To this end, the filesystem stores the hash values of existing deduplication records as unique identifiers in a data structure called the deduplication table which can be kept in memory, on disk, or both. For inline deduplication, the filesystem accesses the table for every write operation.

Deduplication granularity. Filesystems store a large amount of data. Since the size of the deduplication table is proportional to the total amount of data, filesystems perform deduplication at a granularity (i.e., record size) that is a multiple of the data block size. As a result, a sufficient number of data blocks must be written to the filesystem to reach the deduplication record size before the deduplication checks happen.

3 Threat Model

We assume an attacker who has direct or indirect (possibly remote) access to the same filesystem as a victim, and the filesystem performs inline deduplication. We assume the filesystem to be free of bugs and that the configuration as well as the access control settings are all correct.

We consider different local/remote attack scenarios, with the attacker colocated with the victim on a given machine (*local*), across a local-area network (*LAN*), or across the Internet (*WAN*). The attacker wants to obtain secret data from the victim's files, even though the file permissions prevent direct access. In the local scenario, the attacker interacts with the filesystem through attacker-controlled (unprivileged) programs that write to and read from the underlying storage using low-level system calls such as *write()*, *read()*, *sync()*, *fsync()*. In the remote scenario, the attacker interacts with the filesystem through a program that is not under the attacker control. For example, in the case of a server program, this is possible through valid requests that lead to data being written to storage on the attacker's or victim's behalf. We assume there is no limit to the number of I/O operations that can be performed by running programs or by sending requests to a server program. Remote attacks, unlike local ones, require control over the victim's actions to perform successful attacks, e.g., the attacker forcing a victim web server to write a secret into a log file on the remote filesystem. We will discuss additional attack-specific assumptions in the corresponding sections.

4 Exploiting Filesystem Deduplication

In this section, we discuss two general primitives and the challenges to build attacks over deduplicating filesystems. In Section 7, we will discuss how to craft such primitives for modern file systems such as Btrfs and ZFS.

4.1 Primitives

The timed write primitive. Figure 1 shows that inline deduplication handles the writing of unique data differently from existing data on the write path. The common path consists of computing the data identifier and checking whether it is new. If so, the filesystem inserts both the new identifier and the data itself. In contrast, if the data existed already, it is sufficient to update the metadata with a new reference to existing data, which is considerably cheaper. This timing difference forms the basis for our *timed write primitive*. Similar to the COW timing primitive on memory deduplication [7, 10], this primitive allows attackers to leak whether certain data is present on the filesystem during a write operation. Unlike prior memory deduplication attacks, building filesystem-based timed write primitives is complicated, as we soon discuss.

The timed read primitive. Prior memory deduplication attacks [10, 42] show one can detect whether a memory page is deduplicated via a read-based cache timing attacks. However, this approach is not generally applicable to filesystem deduplication. Deduplicated data from different files end up in distinct physical memory pages as the page cache in popular operating systems such as Linux operates at the file level. To craft a filesystem-based timed read primitive, we observe that if a block of a file becomes deduplicated, its physical location on the disk differs from its surrounding blocks. We use this observation as a basis for our *timed read primitive*. Building it faces certain challenges which we discuss next.

4.2 Challenges

Modern filesystems perform many optimizations to improve performance and reliability, resulting in a number of challenges to craft our timing-based deduplication primitives.

C1. Performance. In filesystems, the I/O operations are mostly asynchronous to hide the latency of the underlying storage and other internal filesystem operations from client applications. For this reason, filesystems cache data which complicates the construction of a timing attack significantly. As we shall see, asynchronous operations may necessitate additional attack preparation steps that massage the cache before measuring time or attempting synchronous I/O.

C2. Reliability. To ensure that the system is in a sane state when it crashes, filesystems typically write metadata along with the user data to ensure the filesystem can be restored to a consistent state when catastrophe strikes. Even if data is deduplicated, the metadata still needs to be written to disk, which interferes with our timing channel. This makes building reliable timed write primitives particularly challenging.

C3. Capacity. To perform deduplication efficiently, filesystems need to maintain an in-memory digest of existing stored data. Given that a large number of digests may introduce unacceptable overhead, modern filesystems perform deduplication only across many blocks that are either temporally or spatially close to each other, clustered together in a deduplication record. This complicates building our primitives in two ways. First, detecting a deduplication event across many blocks is not trivial, especially for the timed write primitive. Second, the large deduplication granularity significantly increases the entropy of any target secret deduplication record.

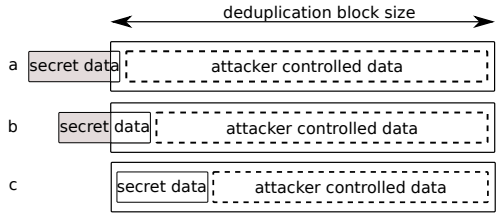


Figure 2: Leaking secret data using deduplication.

5 DUPEFS Overview

To mount DUPEFS attacks, we develop our general primitives into exploitation techniques for three classes of attacks: (i) *data fingerprinting*, (ii) *data exfiltration*, (iii) *data leak*.

5.1 Data fingerprinting

In a data fingerprinting attack, DUPEFS relies on the general timed read/write primitives to reveal the presence of existing known but inaccessible data, such as an inaccessible file of another user. Attackers may use fingerprinting to discover known but embarrassing/compromising content on the server, for instance for extortion purposes. Prior work has shown that timing the writes of a client application may be used for data fingerprinting, but always under the assumption that the client application plays an active role in the deduplication. That is, the client sends data to a cloud application server only if the server does not already have a copy of the data, yielding a timing side channel on write operations [25, 40, 57]. Of course, this is not the case with deduplicating filesystems such as ZFS and BtrFS. DUPEFS shows that similar attacks are still possible, without relying on client applications and application-level deduplication in any way. DUPEFS exploits deduplication performed entirely in the filesystem and is completely agnostic to the applications running on top of the storage stack.

5.2 Data exfiltration

In a data exfiltration attack, DUPEFS relies on the general timed read/write primitives to exfiltrate secret data from a system (or sandbox). The idea is to allow two colluding parties with direct/indirect access to the same system to communicate over a stealthy covert channel. For instance, the parties can use a small number of data blocks with predetermined values to encode messages of a communication protocol [25]. The parties can then exploit timing side channels to find which message was written by the other party. DUPEFS's covert channel can be used to exfiltrate data over LAN or WAN.

5.3 Data leak

In a data leak attack, DUPEFS can leak secret data from a remote system by relying on two exploitation techniques: *alignment probing* and *secret spraying*. The former reduces the entropy of a target secret and enables byte-granular attacks. The latter amplifies the signal and enables remote attacks over LAN/WAN. We first introduce such techniques, then present our example end-to-end data leak attack scenario.

Alignment probing. Figure 2 shows how to exploit alignment probing to leak secret data by carefully aligning known data and then probing for parts of the secret spilled next to it in the same deduplication record (Figure 2-a). By controlling how the data is written to storage, the attacker can stretch controlled data to fill the deduplication record minus one or more bytes of secret data (Figure 2-b). Next, the attacker issues multiple writes with possible guesses for the secret (now low-entropy) record to probe for the unknown byte values until she triggers deduplication (Figure 2-c). At that point, the attacker uses the timed read/write primitive to detect deduplication and hence the correct guess for the unknown byte values. Finally, the attacker repeats the process with multiple alignments until the entire secret is leaked. The attacker relies on the ability to make many instances of the secret appear at various offsets within chunks of otherwise known data. Compared to alignment probing techniques used in prior work in the context of memory deduplication [10], DUPEFS enables such techniques within the filesystem, which is more challenging given the difficulty of enforcing controlled alignment in the storage stack and the coarser block-level interface.

Secret spraying. With basic alignment probing, an attacker can leak part of a secret by timing an I/O operation on a single duplicated or non-deduplicated record. While this may be sufficient for local attacks, remote attacks over LAN/WAN require a stronger signal. To this end, DUPEFS relies on *secret spraying*, a novel deduplication-based exploitation technique for signal amplification. The key idea is to spray candidate secret values over many deduplication records and issue many writes for the corresponding guesses to exploit multiple deduplication events at once. In particular, $N/2$ secret deduplication records and $N/2$ probe deduplication records are carefully crafted with targeted mutations to ensure an attacker can time an I/O operation on $N/2$ deduplicated records (if the guessed probe values are correct) or N non-deduplicated deduplication records (otherwise). Using this technique, DUPEFS can amplify the original number of target deduplication events and thus the signal by a factor of $N/2$.

End-to-end attack. For our example end-to-end remote data leak attack, we target secret data stored in the access log file of a remote (nginx) web server running on top of ZFS/HDD. We specifically target a Single Sign-on (SSO) scenario based on the OAuth protocol [3], where a victim browser accesses an attacker-controlled website with a hidden *iframe* that repeatedly triggers security-sensitive HTTP requests from the victim's browser to an SSO-based service running the nginx web server. In particular, each request URL includes a 22-character OAuth *access token*, which is the target secret

stored in the web server's access log file. DUPEFS relies on the primitives and exploitation techniques introduced earlier to repeatedly interact with the web server and leak the token. Section 8 shows how we address all the aforementioned challenges to mount the attack over LAN or WAN. Before that, we provide necessary internal information about ZFS and Btrfs (Section 6), which we use to build the filesystem-specific timed read/write primitives (Section 7).

6 Deduplication in Modern File Systems

In this section, we discuss how modern filesystems such as ZFS and Btrfs perform basic I/O operations, with a focus on deduplication. In practice, deduplication operates at the granularity of multiple disk blocks, a unit that we generally refer to as deduplication record, but that Btrfs calls a *dedupe block* and ZFS calls *record*. To identify deduplication records, these filesystems use a hash function, typically SHA-256, and keep the metadata in hash tables, which, borrowing ZFS terminology, we will refer to as deduplication table (DDT).

ZFS. The Zettabyte File System (ZFS) [9] is a mature transactional copy-on-write filesystem that implements features such as volume management, deduplication, data compression, and snapshots. To support transactions, changes to on-disk data are first inserted in a transaction queue and processed later. Upon transaction completion, ZFS updates the metadata to reflect the changes and finalize the operation. ZFS implements inline deduplication, hence checks for data uniqueness are on the write path, as part of a transaction. ZFS keeps the deduplication table in memory (for ease of access) and on the disk (for reliability). A file in ZFS consists of aligned records of 128 KB in size and deduplication records are also 128 KB by default. It may take multiple transactions to fill a record, but when filled with data, the record becomes deduplicatable—prompting ZFS to look up its hash in the DDT.

Btrfs. The *B-tree* filesystem (Btrfs) [1, 47] is a modern Linux copy-on-write (COW) filesystem that implements features similar to ZFS and uses B-trees along with COW semantics to update the data on the disk. The B-trees are optimized for COW semantics and contain both data and bookkeeping information. The data in Btrfs are stored in *extents*. An extent consists of contiguous, aligned, on-disk data blocks, checksummed for integrity. Like ZFS, Btrfs is transactional. It collects data block changes in memory until the number of collected changes exceeds a threshold or a timeout occurs, at which point it flushes the changes to a new location on the disk. The filesystem state is kept in checkpoints that update the superblock, while extents store metadata such as the file creation checkpoint, the disk area corresponding to a file, the logical offset, and the number of data blocks in the extent. Deduplication in Btrfs works at the level of extents, which become candidate deduplication records when their size reaches the *deduplication record size* of 128 KB (default).

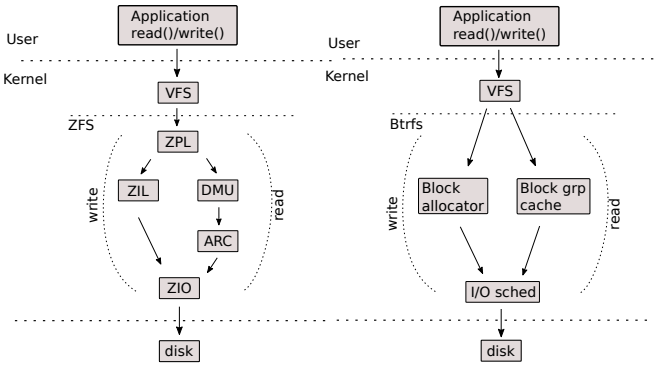


Figure 3: ZFS read/write paths

Figure 4: Btrfs read/write paths

6.1 Writes in Deduplicating Filesystems

Write operations in modern filesystems can be either synchronous or asynchronous. An asynchronous write operation does not block the application, which can resume its execution as soon as the data reaches the kernel. Meanwhile, the filesystem dispatches the operation through multiple layers of buffering and writes the data to storage later. In contrast, synchronous writes block and while the data still passes through intermediary buffers, the control returns to the application only after the data is written. When an application calls the *write* syscall, the data is moved from the application's buffers to kernel space, in the page cache of the Virtual File System (VFS), with its pages marked dirty. Next, the kernel dispatches the operation to the specific filsystem (e.g., ZFS or Btrfs).

Writes in ZFS. The left-side of Figure 3 describes the data path for such writes in ZFS. After placing the write in the ZFS Intent Log (ZIL) on disk, the kernel returns from the syscall. The write remains there until, at a later time, ZFS processes the ZIL by passing the data to the ZFS I/O (ZIO) layer and updating the deduplication table. In particular, ZFS uses an on-disk ZIL for reliability and an in-memory ZIL for efficiency. Write requests are committed to the on-disk ZIL in the following cases: the write is synchronous, the application calls the *fsync* syscall, or five seconds have elapsed. Finally, to prevent applications from overwhelming the filesystem, ZFS implements *write throttling*, which temporarily blocks aggressive writers to process outstanding writes.

With respect to the challenges in Section 4.2, challenge C1 stems from the ZIL introducing asynchronous behavior even for synchronous writes as a performance optimization, challenge C2 stems from metadata management in the ZIL and deduplication table for reliability reasons, and challenge C3 stems from the large 128 KB deduplication records that ZFS uses for capacity reasons. Finally, write throttling and ZIL flushing both introduce additional noise.

Writes in Btrfs. Figure 4 describes the equivalent data paths in Btrfs, with the write operation passing through the deduplication checks when the filesystem writes the data to disk [13].

In the default filesystem settings [14], Btrfs deduplicates data using the inline *deduplication record size* of 128 KB. After looking up the identifier in the deduplication table, it writes both data and metadata to disk if the data is new, or only the metadata if the data already exists.

The extents in Btrfs are contiguous on-disk data blocks and each file consists of one or more extents. In case of a large write, only the full extents are candidates for deduplication, while any remaining bytes become an extent with a smaller size. If the application subsequently appends data to the file, the new data is not merged with the small extent, but placed in a new extent. Btrfs does not support modifying or splitting extents and a write in an existing extent will trigger the creation of a new extent with the new data and an update of the file indexing information. For instance, when an application overwrites the first 100 bytes of a file, the copy-on-write behavior creates a new extent of one disk block which contains the new 100 bytes plus the rest of the first disk block of the original extent. When the file is read, Btrfs returns the first disk block of the new extent while taking the remainder of the data from the original extent.

With respect to the challenges identified in Section 4.2, we see again that the asynchronous transaction introduces challenge C1, the metadata handling for reliability introduces challenge C2, and the large deduplication records introduce challenge C3. Furthermore, partially filled extents, if any, and alignment issues complicate the attack.

6.2 Reads in Deduplicating Filesystems

When the kernel handles a *read* syscall, it retrieves the data either from the filesystem cache or from the disk.

Reads in ZFS. After the read syscall has passed through the VFS layer and the ZFS Posix Layer, ZFS checks if the data exists in the *Adaptive Replacement Cache*(ARC) and, if so (right-hand side of Fig. 3), returns the data to the application. Otherwise, it transfers control to the ZFS I/O (ZIO) layer which retrieves the data from the disk.

Reads in Btrfs. In Btrfs, after the read syscall has passed through the VFS layer, Btrfs performs a search in the "block group cache". In case of a miss, Btrfs retrieves the data from the disk. As mentioned, Btrfs may create new, partially-filled extents in case of file modifications and, in that case, the read may access more extents than one would expect.

For both filesystems the COW behavior creates an on-disk layout that is non-sequential for deduplicated data and typically sequential otherwise. Reading the contents of a file with deduplicated data involves random accesses on the disk which is usually measurably slower than sequential accesses. However the partially filled extents that result from file modifications also incur non-sequential accesses and generate noise.

7 Attack Primitives

This section introduces the general timed write/read primitives for ZFS and Btrfs to perform DUPEFS attacks. To exploit the timing side channel, an attacker writes data to the filesystem using carefully crafted patterns that massage the filesystem into an exploitation-friendly state. In particular, to handle *transactional behavior* potentially delaying write operations, the attacker performs multiple writes to flush each transaction. To handle *copy-on-write behavior* and the *coarse deduplication granularity*, the attacker writes a sufficient amount of data to trigger deduplication checks. To handle *caching behavior*, the attacker uses *sync* operations to force a cache flush where possible (locally on Btrfs) or massages the cache with enough I/O operations otherwise. Finally, to handle *concurrent operations* from other applications, the attacker repeatedly measures filesystem operations to confirm deduplication.

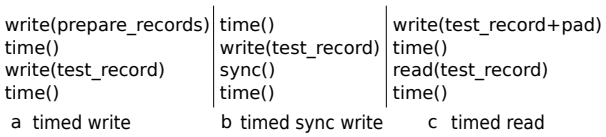


Figure 5: DUPEFS attack primitives

Overview. Figure 5 details our attack primitives in pseudocode. Each primitive describes I/O operations on multiple deduplication records to trigger deduplication checks. The test deduplication record is the record of 128 KB (by default) an attacker writes to the filesystem and then measures the impact of the operation in the time domain (by timing the write itself or subsequent operations). In the write operation, the test deduplication record can be padded with other deduplication records that help with filesystem massaging (e.g., flushing caches), alignment probing, and secret spraying. Depending on the attack, the content of the test deduplication record is known (e.g., data fingerprinting) or mostly known except for, say, 1 byte the attacker needs to guess (e.g., data leak).

In details, Figure 5-a presents a timed write primitive (available on ZFS). First, the attacker writes multiple controlled deduplication records to the filesystem in a *prepare phase*. This is to cause a transaction cache flush (absent *sync* support in ZFS) and prepare alignment. Next, the attacker issues (and times) a write of the test deduplication record.

Figure 5-b presents a timed synchronous write primitive (available on Btrfs). First, the attacker issues a *sync* operation to flush the caches. Next, the attacker issues (and times) a write of the test deduplication record. This primitive is available only in exploitation scenarios where the attacker can trigger *sync* operations. This is nontrivial in remote data leak attacks, where the only (unlikely) option is to lure a victim server program into issuing an explicit *sync* operation.

Figure 5-c presents a timed read primitive (available on Btrfs), which can probe for deduplication events after the

fact. First, the attacker writes a test deduplication record (plus padding) to the filesystem, which may trigger deduplication. Next, the attacker reads back the same data. If deduplication did happen, the (random-access) read will take longer than the (sequential-access) read for the nondeduplicated case.

In the next section, we analyze the timing side channel for the different primitives on an SSD (Corsair Force LS SSD S9FM02.6) and a magnetic HDD (model ST1000). The latter is obviously slower, but still a popular type of storage medium, especially in server environments [6]. Unless otherwise noted, we consider default configurations of ZFS and Btrfs, with a deduplication record size of 128 KB. We repeat experiments several times and find marginal deviations in results.

Timing differences on Btrfs. To verify the existence of the timing side channel on Btrfs, we evaluate Btrfs running on Linux (v4.20). In our experiments, we issue 500 *synchronous* write operations (thanks to the Btrfs-supported *sync*) of identical (deduplicated) deduplication records and 500 write operations of unique deduplication records. We obtain an average timing difference between deduplicated and unique write operations of 0.57 ms for the SSD and 24.5 ms for the HDD. Repeating the same experiment with asynchronous write operations leads to no statistically meaningful difference. As such, we do not further consider this configuration in our analysis. Nonetheless, this experiment confirms a realistic signal for our (synchronous) Btrfs write primitive on Linux.

On the same setup, we issue 500 read operations for deduplicated records and 500 read operations for nondeduplicated records. We obtain an average timing difference between deduplicated and nondeduplicated read operations of 0.7 ms for the SSD and 17.22 ms for the HDD. This experiment confirms a realistic signal for our Btrfs read primitive on Linux.

Timing differences on ZFS. To verify the existence of the timing side channel on ZFS, we first consider ZFS running in its natural FreeBSD (10.4) habitat. In our experiments, we perform asynchronous write operations using both identical (i.e., deduplicatable) and unique records. We write enough data in the 5 second time interval before the in-memory ZIL is flushed to disk and measure the time to complete the individual write operations.

We issue 500 write operations of identical (deduplicated) records and 500 write operations of unique records. We obtain an average timing difference between deduplicated and unique write operations of 0.04 ms for the SSD and 2.6 ms for the HDD. This experiment confirms a realistic signal for our ZFS write primitive on FreeBSD. While the SSD signal seems weak at first glance, this is just due to the larger default transaction cache size on ZFS. This simply means we need a larger number of writes for a strong signal. We confirm this by repeating the SSD experiment with a smaller transaction cache of 10 deduplication records and measuring a timing difference of 1.23 ms.

Due to different licensing models (CDDL vs. GPL), ZFS is not directly included in the Linux kernel. As a result, the Linux implementation contains more software layers that collectively dampen the signal for our timing side channel. To confirm this intuition, we re-run our last experiment on Linux (v4.20)-based ZFS and report an average timing difference between deduplicated and unique writes of 0.16 ms on HDD and no statistically meaningful difference on SSD. Similarly, ZFS' efficient read implementation does not yield a meaningful signal on HDD or SSD. As such, we do not further consider Linux/ZFS or ZFS-based read primitives in our analysis.

8 DUPEFS Exploitation

To illustrate the severity of DUPEFS, we exemplify attacks for *data fingerprinting*, *data exfiltration*, and *data leakage*.

8.1 Data fingerprinting

We exemplify a data fingerprinting attack using our Btrfs-based synchronous write primitive in a local exploitation scenario[1]. A local unprivileged attacker seeks to detect the existence of an inaccessible file (or deduplication record within a file) with known content. This is useful to detect specific system binaries or configuration files and fingerprint vulnerable programs running on the victim system. The attacker first prepares an oracle of target files with a size matching or larger than the deduplication record size. Next, the attacker runs an unprivileged program on the target system to repeatedly effect the timed write primitive for each file. Using the *syncfs* and *write* syscalls, the attacker synchronously writes each file to the victim filesystem and times the operation to detect deduplication indicating the presence of the file.

8.2 Data exfiltration / covert channel

We exemplify a data exfiltration attack using our Btrfs-based synchronous write primitive in a local exploitation scenario[2]. A local unprivileged attacker (or "sender") seeks to exfiltrate data from a sandbox over a covert channel. The receiver is an unprivileged colluding party running on the same system. For simplicity, the two communicating parties run a basic covert channel protocol and synchronize using the system clock.

First, the sender writes N deduplication records for each bit of data to a file. Each record is filled with a predetermined deduplication record prefix, a $[0...N-1]$ deduplication record ID, and the $[0-1]$ bit value. Next, the receiver uses the timed write primitive on another file in order to test for each unknown bit value across the same number of (N) deduplication records. The receiver uses the same record format as the sender and tests for 0-bit deduplication records. A signal (or its absence) determines the transfer of a 0-bit (1-bit) value.

[1] We have also reproduced the attack on the ZFS write primitive

[2] We have also reproduced the attack on the ZFS write primitive

After the receiver has stored the leaked bits, the protocol repeats with the sender writing new data-encoding deduplication records. For example, to exchange 1 byte of information using $N = 10$, the sender writes $10 * 8$ deduplication records to a file. The receiver then uses the timed write primitive with $10 * 8$ 0-bit test deduplication records on a separate file. Fast (deduplicating) writes on the first 10 deduplication records signal a 0 value for the first bit, slow (nondeduplicating) writes signal a 1 value for the second bit, and so on. We use multiple (N) records to transfer a single bit to amplify the signal and thus reduce the error rate of the covert channel. We use different deduplication record IDs to prevent the deduplication records written by the sender (or receiver) from deduplicating against themselves.

8.3 Remote data leak

We exemplify a data leak attack using our ZFS-based write primitive in a remote exploitation scenario[3]. The attacker seeks to leak an OAuth access token [3] from the access log of a remote nginx web server running on ZFS/HDD [4]. The nginx web server hosts a website (e.g., `https://someapp.com`) which the victim has granted access to an authenticated third-party service (e.g., `https://github.com`) via long-lived OAuth access tokens. At a high level, to implement the attack, the attacker lures a victim browser into an attacker-controlled website, which repeatedly (but transparently to the user) forces the browser to access the nginx web server with the secret OAuth access token encoded in the URL. As such, the secret is repeatedly spilled on the nginx access log stored on ZFS, enabling alignment probing and secret spraying. Meanwhile, the attacker concurrently and independently probes the nginx web server to leak the secret OAuth token one byte at the time. With the token, the attacker can gain access to the third-party service with the victim's credentials.

Attack scenario. The end-to-end attack scenario in our example has the following actors: the victim browser, a Single Sign On (SSO) server, an SSO client running nginx on top of ZFS, a third-party service the SSO client has been granted access to on the victim's behalf via SSO access tokens, and a malicious website under the control of the attacker (say `https://attacker.com`). Specifically, we target SSO access tokens from the OAuth 2.0 implicit grant access scheme [3] and assume such access tokens do not expire for the entire duration of the attack. This is a sensible assumption, as many third-party services use long-lived access tokens that never expire [2]. To mount the attack, the attacker-controlled website includes a hidden *iframe* which repeatedly forces the browser to connect to the SSO client with the secret OAuth token. This is legal behavior, but defenses against clickjacking, *X-Frame-Options* (XFO) in particular, may prevent such accesses from

[3] This primitive provided the best signal for remote attacks

[4] In our experiments, the HDD setup was necessary for a realistic signal across the Internet

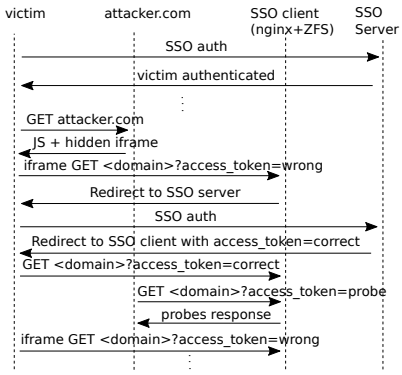


Figure 6: Data leak attack request sequence

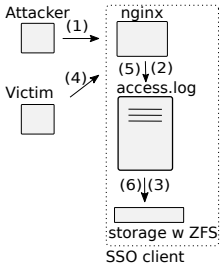


Figure 7: Data flows in the SSO client

an *iframe*. In our example attack scenario, we assume the victim server does not offer XFO (which is very common even in modern websites [26]), or the victim browser's XFO implementation is bypassable (e.g., Microsoft Edge [15]).

Attack workflow. Figure 6 presents all the requests exchanged between the victim browser, the attacker-controlled website, the SSO client, and the SSO server before and during the attack. Before the attack, the victim has already obtained an OAuth access token for the third-party service by authenticating with the SSO server. In the first stage of the attack, the victim browser is lured to the attacker-controlled website. The website serves the victim browser some attacker-controlled JavaScript and a hidden iframe. The latter issues an HTTP request to the SSO client using an incorrect access token.

Upon receiving the incorrect access token, the SSO client redirects the iframe of the victim browser to the SSO server to obtain a new valid access token. When the SSO server receives the request, it simply acknowledges that the victim browser is already authenticated and redirects the iframe again to the original SSO client. The redirect causes the iframe to issue an HTTP GET request with the correct (and secret) OAuth access token to the SSO client. Since the access token is encoded in the URL and the URL is logged in the access log of the SSO client's nginx web server by default, the request ultimately spills the target secret on the victim ZFS filesystem. At that point, the attacker can independently issue multiple GET requests to the SSO client in order to probe for the secret byte values of the access token. Meanwhile, the malicious JavaScript from the victim browser can reload the iframe and the entire sequence repeats, until the attacker obtains the secret access token using the timed write primitive for ZFS across the network.

Inside the SSO client. We now focus on the SSO client, which runs the nginx web server and stores its access log on the target ZFS filesystem. The requests that the attacker directly or indirectly sends to the SSO client reach nginx, which, in turn, uses system calls to write each HTTP request to the access log. The data flows for the attacker and victim inside the SSO client are shown as (1)-(3) and (4)-(6) in

Figure 7. The sequence of events above results in different classes of attacker-controlled entries being spilled into the access log: initial GET requests issued by the browser with an incorrect access token, second-stage GET requests issued by the browser with the correct access token, GET requests issued and timed by the attacker to probe for the secret bytes.

The attacker carefully massages the workflow above to interleave the different classes of GET requests and implement the required primitives. In particular, the attacker first issues a number of wrong-access-token requests causing nginx to carefully align the access log entries. By massaging the alignment (starting from a baseline of known access log alignment leaked with timed write primitive), the attacker can ensure the next GET request with the correct access token fills an entire access log deduplication record and spills the last byte of the access token into the next deduplication record. The attacker then performs additional GET requests to nginx to fill the rest of the deduplication record. At that point, the attacker can time specially-crafted GET requests to probe for each of the possible byte values. Figure 9 presents the access log file layout of nginx induced by the proposed attack patterns. Since OAuth uses 22-character access tokens, the entire process is repeated 22 times [3], leaking 1 byte value (from the base64 alphabet) of the access token per iteration.

Since we control both the browser- and the attacker-issued requests to nginx (using controlled iframe refreshes or independent client requests from an attacker-controlled machine), we can repeat the individual steps many times. This enables secret spraying for amplification (simply mutating the original wrong-access-token requests to cause the browser to send different variations of the correct-access-token requests to nginx) and repeated alignment probing (shifting the alignment by 1 byte every time) to leak all the bytes of the secret.

For every byte value of the secret access token probed, the attacker runs a number of P probes per byte using the pattern described in Figure 9 and measures the time to complete the individual network requests. Whenever the in-memory ZIL is flushed to disk a peak can be observed in the timing measurements of the attacker as feedback directly from the VFS layer. The attacker can analyze the real-time duration of the ZIL flush by observing the peaks in real time. The duration of the peak is larger when nondeduplicated data is flushed than when there is deduplicated data. To amplify the difference the attacker uses the secret spraying technique to submit large amounts of data to ZIL. Figure 8 shows an example of the width (duration) difference for 2 consecutive peaks, for both the deduplicated and nondeduplicted case, separated by a 5-second interval. The attacker picks the byte value that produces the peak with a duration that is below a threshold as the correct byte value of the secret access token. The threshold is discovered empirically by the attacker (e.g., a peak duration of at least 0.5s to signal the correct byte value for the data in Fig. 8) and depends on the network bandwidth.

To improve performance of the logging feature, nginx uses

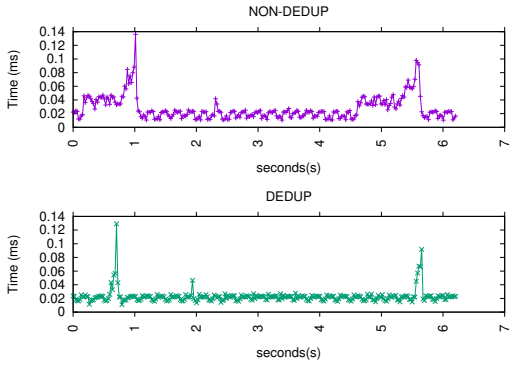


Figure 8: Dedup vs non-dedup peaks: the duration differs noticeably

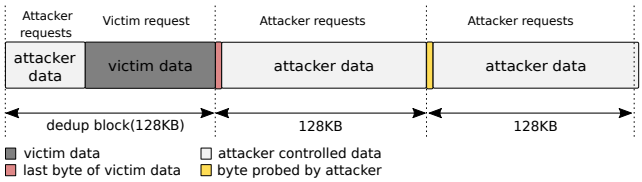


Figure 9: Crafted attack patterns in the nginx access log

an intermediary buffer which collects I/O operations before sending them to ZFS. The buffer is limited to 5 requests irrespective of their size and works as a FIFO queue: once a new request comes in, the oldest request is pushed out and submitted to the VFS layer to be written to disk. To deal with buffering, the attacker floods the intermediary buffer and controls when the write operation (including deduplication checks) is performed. In fact, nginx' use of the intermediary buffer helps the attacker, as it is easier to control alignment (within the buffer) and generate single large writes.

Note that the attacker controls most of the data that is written to the log, namely: the url of the requested resource, the user agent field, the request body, etc., which allows the attacker to reduce the entropy of probing to that of the secret. Some data, such as the timestamp, is not directly controlled by the attacker. However, the attacker learns it, at second granularity, from the Date header field in the HTTP reply—which is mandatory according to RFC2616. Based on the acquired timestamps, the attacker can synchronize the requests with the server, eliminating entropy.

9 Evaluation

We evaluate our DUPEFS attacks on a system equipped with an Intel Core i5-8250U CPU (4 CPU cores), 16 GB of RAM, an SSD (Corsair Force LS SSD S9FM02.6), and a magnetic HDD (Seagate ST1000). We run our target filesystem implementations in their default settings (with a deduplication record size of 128 KB) and on their natural operating system platforms, namely FreeBSD (10.4) for ZFS and Linux (v4.20) for Btrfs. ZFS, in particular, uses the default amount

Table 1: File fingerprinting

File	Type	Size	Success
config-4.11.3-200.fc25.x86_64	text	181 KB	70%
lena_color.gif	binary	223 KB	55%
libz3.so	binary	22 MB	99%
x86_64-redhat-linux-c++	binary	1 MB	99%

Table 2: Covert channel

N	Bit errors	Time	BR	BER	I/O
20	13	375s	0.320 bps	10.83%	76.8MB
40	14	746s	0.160 bps	11.66%	153.6MB
60	12	1591s	0.075 bps	10.00%	230.4MB
100	6	1873s	0.064 bps	5.00%	384.0MB
120	3	2387s	0.050 bps	2.50%	460.8MB

of memory for dirty data (10% of the RAM size) and its reliability configuration using sync=*always*, which ensures that the in-memory ZIL is also saved to disk—enabling recovery after a crash. We repeat all experiments multiple times on a quiescent server and report the mean values.

9.1 Data Fingerprinting

We evaluate our data fingerprinting attack using the Btrfs-based write primitive over HDD. We have reproduced these experiments on SSDs and using the ZFS write primitive, observing similar results, which we omit for space reasons. For our experiments, we consider an unprivileged attacker interested in probing for a number of sensitive files (larger than the deduplication record size of 128KB) on the running sytem.

Table 1 presents our results. We consider 3 different binary files (a picture, a shared library, and a binary executable) and one text file (the kernel configuration file) for our analysis. Note that, most of the contents of the kernel configuration file is predictable, given that each line normally refers to a configuration OPTION in one of the following 4 variants:

```
1. #CONFIG_OPTION is not set
2. CONFIG_OPTION=y
3. CONFIG_OPTION=m
4. CONFIG_OPTION=n
```

DUPEFS reliably figerprints individual (128 KB) fragments of the target files. The table presents success rates for fingerprinting the entire file. DUPEFS can reliably fingerprint the target data except the last sub-128 KB chunk of a file. Thus the small (181 KB and 223 KB) files have lower success rates.

9.2 Data Exfiltration

We now evaluate our data exfiltration attack, again using the Btrfs-based write primitive over HDD. As we shall see, other configurations again yield comparable results. For our experiments, we consider two unprivileged colluding parties running on the same machine. Both parties exchange information using the covert channel protocol introduced earlier.

Table 3: LAN 1 byte data leak

Success	Attack time/byte	Probes/byte val	I/O
50%	19.2 min	200	4.9 GB
80%	25.6 min	300	7.3 GB
92%	42.6 min	400	9.8 GB
96%	78.9 min	800	19.6 GB

Table 4: WAN 1 byte data leak

Success	Attack time/byte	Probes/byte val	I/O
64%	24.5 min	200	4.9 GB
87%	38.4 min	300	7.3 GB
94%	59.7 min	400	9.8 GB
94%	110.9 min	800	19.6 GB

To evaluate the channel, we transfer chunks of 15 bytes from the sender to the receiver, measuring the bit rate and bit error rate. We repeat the measurements for different numbers of deduplication records (N, determining the number of probes per bit) to investigate the throughput/reliability tradeoff.

Table 2 presents our findings, including the amount of I/O involved in the transfer. Our results show that the bit rate starts from 0.32 bit/s for 20 probes per bit with a bit error rate of 10.83% and drops to 0.05 bit/s for 120 probes per bit with a bit error rate of 2.5%. We also reproduced these results on SSDs and using the ZFS write primitive, with a proportional signal, matching the trend detailed in Section 7. Our results confirm the covert channel can be used for realistic data exfiltration attacks. Note that the bit errors in the covert channel can be compensated by running a simple error correction protocol.

9.3 Data leak

We now evaluate our remote data leak attack, using the ZFS-based write primitive. In many environments, this would be the most worrying attack. The goal of the attacker is to leak the access token, as used commonly on the web, from an SSO client *across the network*. The SSO client runs nginx version 1.14.0_12,2 with the default settings, logging HTTP requests to the access log, over ZFS/HDD. We consider 2 locations for the attacker: one where the attacker is on a wide area network (WAN), far from the server, and one where the attacker is in the same local network (LAN). In the WAN attack, the attacker is located 12 hops away from the victim with an RTT of 2 ms, measured using traceroute with TCP SYN probes. In the LAN scenario, the attacker is located 1 hop away from the victim with an RTT of 0.1ms. The attacker probes for OAuth secrets of 22 bytes encoded in base64.

To evaluate the attack success rate and attack time, we vary the number of probes per byte value from 200 to 800. Tables 3 and 4 present the success rate (out of 50 attempts) and the total attack time for 1 byte in a LAN and WAN setting.

LAN attack. Table 3 presents the success rate to discover 1 byte over a LAN, the time needed to leak 1 byte given the number of probes/byte value used, and the amount of I/O

(in GB) used in this attack scenario. The attacker can tune the attack to obtain a desired attack performance-reliability tradeoff. Given a success rate of 92%, the attacker, using the configuration of 400 probes/byte value, leaks 1 byte over the network in roughly 42 min and the full 22-character OAuth access token in around 15 hours.While high success rates require substantial amounts of I/O, such attacks are already within reach of attackers today and will be even more so as the speed of file systems and networks increases.

WAN attack. Table 4 presents the success rate and time needed to guess 1 byte over a WAN, given the number of probes per byte value used, and the corresponding amount of I/O. As shown, the attacker has different options to select the reliability-performance tradeoff for a desired success rate. For example, for a success rate of 94%, the attacker can use 400 probes/byte value, resulting in approx 1h to leak 1 byte and around 21h to leak the full 22-character OAuth access token.

Noise. As well-established in literature [18, 20, 21, 42], the signal progressively degrades in the presence of noise (i.e., concurrent I/O workloads). As a result, in noisy environments, the attack, when not conducted during off-peak/idle times, would require more repetitions and hence more time [20]. For example, in a LAN setting, we generated concurrent load by continuosly reading data from `/dev/random` and writing it to disk. The attack used 800 probes per byte value. The attack duration was ≈91 min (up from 78.9 min), the success rate dropped to 90% (down from 96%), and the I/O performed was 19.6 GB by the attacker and 6 GB by the script.

10 Mitigation

Similar to prior side-channel attacks, DUPEFS attacks are not very stealthy and could be detected by an intrusion detection system (IDS) monitoring I/O activity. Nonetheless, as observed in literature [50], it is difficult to design such an IDS to guarantee no false negatives and no false positives in practice. As such, we now consider more principled mitigations that can provide security-by-design guarantees.

An ideal implementation of filesystem deduplication would save space and have constant-time behavior. In other words, all the deduplication-aware I/O operations need to implement a *same-behavior* policy [42]. This essentially translates to each operation traversing the storage stack in the same amount of time regardless of whether data handled by the operation has been deduplicated or not. In practice, a strict same-behavior policy is neither desirable—as it would hurt space savings—nor practical—as it would not only require a redesign of the filesystem, but also of the physical storage devices. Our goal here is instead to discuss a practical, *pseudo-same-behavior*, mitigation strategy that drastically reduces the (I/O-based) signal and deters remote attacks.

A mitigation for the write path would change the behavior described in Figure 1 for the case when the deduplication checks conclude that the data exist to update the reference and then still perform the write operation to the disk. The duplicate data is simply overwritten. To investigate the practicality of this strategy, we have experimented with Btrfs implemented in the Linux kernel (v4.20) [12].

Write path. In Btrfs, the *submit_compressed_extents* function contains the program point where the write code path diverges in a deduplication-dependent way and induces different behavior in the time domain. Inside this function the block allocation is followed by the *dedupe_hash_hit* check which determines whether to finalize deduplication or else write a nondeduplicated block to disk. To bring the implementation as close to the same-behavior policy as possible with few code changes, we propose a patch to also perform the write operation on the else branch of *dedupe_hash_hit*, simply overwriting existing on-disk data. With a 5 LOC change, we preserve space savings, only slow down deduplicated write paths (mirroring the execution time of non-deduplicated write paths), and eliminate the classic deduplicated write path side channel. We have verified the proposed strategy is sufficient to cripple the SSD/HDD signal for remote attacks.

To verify the performance impact of our proposed mitigation we ran microbenchmarks on a system with an SSD, using 5,000 synchronous write operations with deduplicated data (worst-case scenario)—with and without our mitigation enabled. When the mitigation is enabled, the median performance overhead is as low as 6.7% compared to the mitigation disabled case. Note that performing redundant I/O reduces device longevity compared to the `deduplication=on` baseline (but is equivalent to the `deduplication=off` baseline).

Read path. For this path, the mitigation has to enforce pseudo-same-behavior for disk access patterns. For this purpose, we need to patch the *btrfs_readpages* function, which reads the extents of a file. Since a strict same-behavior policy would require random access for each read operation (with possible performance loss), our strategy here is to introduce time jitter on the read path. We implemented this strategy with a 2 LOC change. We have verified (by running similar microbenchmarks as done for the write path) that even low jitter values are sufficient to cripple the SSD/HDD signal for remote attacks, while introducing no observable performance impact. Note that applying the same jitter-based mitigation on the write path is, in contrast, ineffective (as we have experimentally confirmed), since the write-path signal is too strong to be efficiently eliminated using jitter.

Limitations. With less than 10 LOC changed in the large Btrfs codebase, we believe our mitigation proposal is practical and has a chance at mainline inclusion. Nevertheless, we emphasize these changes only seek to deter remote attacks but cannot completely eradicate the signal for local attacks. For instance, the write path mitigation only enforces a same-behavior policy from the disk perspective. It does *not* eradicate all the code differences on the write path. We believe this

limitation still offers a good compromise, since, on a local setting, there are already more powerful side channels (e.g., cache side channels) to mount practical end-to-end attacks.

Another limitation is the mitigation operating only in the time domain (similar to prior secure memory deduplication systems [42]). There may be other side channels that escape our same-behavior policy in other domains. For instance, tools reporting free disk space information to unprivileged users may re-enable very reliable (local) attacks. Free disk space or similar leaky filesystem information should be restricted to privileged users to deter practical side-channel attacks.

11 Related Work

Deduplication is used to efficiently store data in different types of memory, ranging from desktop computers [4, 39] to caches [35, 54] and cloud services [5, 33].

Deduplication Attacks. Many recent efforts investigate the security of deduplication from both an offensive and defensive perspective [27, 28, 34, 40, 44, 46]. Existing storage-based attacks exploit deduplication in the cloud application layer, mostly to detect the presence of particular files. Harnik et al. [25] describe deduplication-based attacks to identify files on the cloud side by observing the amount of data transferred by the client. Mulazzani et al. [40] exploit the hashing mechanism of the Dropbox storage provider to obtain information about the existence of a file. Access to it can be obtained by providing the hash to the service. The attacks exploit the application-level cross-user deduplication performed by Dropbox. In contrast, DUPEFS targets low-level deduplication in modern filesystems, enabling application- and cloud-agnostic attacks leaking arbitrary byte-granular data.

Memory deduplication is a technique used by modern hypervisors or operating systems to reduce main memory usage. Memory deduplication attacks can locally fingerprint applications [22, 52], operating systems [43], or defeat ASLR [7]. Dedup Est Machina is a more advanced memory deduplication attack [10], which can read arbitrary data from the local system's memory using alignment probing and other memory-specific exploitation techniques. In contrast, DUPEFS repurposes alignment probing to exploit filesystem deduplication and combines it with secret spraying to enable byte-granular data leak attacks across the network for the first time.

Deduplication Defenses. Rabotka [46] identifies 4 classes of countermeasures against traditional storage-based deduplication attacks: encryption enforced by the client [37, 51] or by a third party [56], noise added by probabilistic uploads [24, 25, 60], proof of ownership [24, 58], and obfuscation enforced by an intermediate gateway in the network [27]. All these mitigations are only applicable to cloud application scenarios where the client plays an active role in the deduplication. As such, they are ineffective against DUPEFS' attacks based on filesystem deduplication. VUSion [42] proposes a memory deduplication redesign based on same-behavior (i.e., constant-time sensitive operations) and other principles to cripple both side-channel and Rowhammer attacks. In contrast, we show enforcing a pseudo-same-behavior policy—sufficient to deter remote attacks—is feasible with small changes rather than a complete filesystem redesign.

Network Side-channel Attacks. Many prior efforts propose remote network side-channel attacks. Early attacks leak sensitive (cryptographic) data but only target vulnerable server applications [8, 11, 41]. More recently, NetSpectre [48] and NetCAT [32] exploit cache side channels over the network. The former targets a vulnerable (or cooperative) server application containing specific gadgets, while the latter assumes specialized hardware (Intel DDIO and RDMA)—similar to state-of-the-art network-based Rowhammer attacks [36, 53]. Page cache attacks [23] can exploit the operating system's page cache to implement a covert channel between cooperating parties over the network. In contrast to all these attacks, DUPEFS can target arbitrary noncooperative applications running on top of a commodity hardware/software stack and can leak sensitive byte-granular data over LAN/WAN. In concurrent work, Schwarzl et al. [49] showcase similar remote attacks exploiting memory (rather than) storage deduplication and operate byte-by-byte disclosure at comparable speeds.

12 Conclusion

In this paper, we showed that deduplication in commodity filesystem implementations poses a nontrivial security threat. Specifically, we presented evidence that such implementations yield timing side channels that can be abused to remotely leak arbitrary data at byte granularity. To substantiate our claims, we presented DUPEFS, a class of filesystem deduplication-based attacks for remote data fingerprinting, exfiltration, and disclosure. Our end-to-end data leak attack demonstrates DUPEFS can disclose sensitive data from a remote server program even across the Internet. Finally, we investigated mitigations and showed that implementing a pseudo-same-behavior policy for all the I/O operations in the time domain is practical without a full filesystem redesign.

Disclosure

We have disclosed our findings to the affected parties.

Acknowledgements

We thank our shepherd, Carl Waldspurger, and the anonymous reviewers for their comments, as well as Ilias Diamantakos for early signal testing. This work was supported by the EU's Horizon 2020 programme under grant agreement No. 825377 (UNICORE), Intel Corporation through the Side Channel Vulnerability ISRA, and NWO through project "Intersect".

References

[1] btrfs Wiki. https://btrfs.wiki.kernel.org/index.php/Main_Page.

[2] Github: Creating a personal access token for the command line. https://help.github.com/en/github/authenticating-to-github/creating-a-personal-access-token-for-the-command-line.

[3] The oauth 2.0 authorization framework. https://tools.ietf.org/html/rfc6749.

[4] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *ACM Trans. Storage*, 2007.

[5] Frederik Armknecht, Jens-Matthias Bohli, Ghassan O. Karame, and Franck Youssef. Transparent Data Deduplication in the Cloud. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 886–900, New York, NY, USA, 2015. ACM.

[6] Backblaze. Backblaze hard drive stats q2 2019. https://www.backblaze.com/blog/hard-drive-stats-q2-2019, 2019.

[7] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: Silently Breaking ASLR in the Cloud. WOOT'15, 2015.

[8] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, The University of Illinois at Chicago, 2005.

[9] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. https://pdfs.semanticscholar.org/27f8/1148ecbcd04dd97cebd717c8921e5f2a4373.pdf, 2003.

[10] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. SP'16, 2016.

[11] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[12] Btrfs Contributors. Linux fork for in-line dedupe. https://github.com/littleroad/linux/tree/dedupe_latest, 2019.

[13] btrfs Wiki. Dedupe design notes. https://btrfs.wiki.kernel.org/index.php/Design_notes_on_dedupe.

[14] btrfs Wiki. User notes on dedupe. https://btrfs.wiki.kernel.org/index.php/User_notes_on_dedupe.

[15] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. A tale of two headers: A formal analysis of inconsistent click-jacking protection on the web. In *USENIX Security*, 2020.

[16] Feng Chen, Tian Luo, and Xiaodong Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[17] Zhuan Chen and Kai Shen. Ordermergededup: Efficient, failure-consistent deduplication on flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 291–299, Santa Clara, CA, February 2016. USENIX Association.

[18] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. CCS'14, 2014.

[19] John Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 30(1):247 – 252, January 1982.

[20] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[21] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. NDSS'17.

[22] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed Javascript. ESORICS'15. 2015.

[23] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page cache attacks. In *CCS*, 2019.

[24] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[25] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security Privacy*, 8(6):40–47, November 2010.

[26] Mohammadreza Hazhirpasand, Mohammad Ghafari, and Oscar Nierstrasz. Tricking johnny into granting web permissions. In *Proceedings of the Evaluation and Assessment in Software Engineering*, pages 276–281. 2020.

[27] O. Heen, C. Neumann, L. Montalvo, and S. Defrance. Improving the resistance to side-channel attacks on cloud storage services. In *2012 5th International Conference on New Technologies, Mobility and Security (NTMS)*, 2012.

[28] J. Hur, D. Koo, Y. Shin, and K. Kang. Secure data deduplication with dynamic ownership management in cloud storage. *IEEE Transactions on Knowledge and Data Engineering*, 2016.

[29] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Jackpot stealing information from large caches via huge pages. Cryptology ePrint Archive, Report 2014/970, 2014. https://eprint.iacr.org/2014/970.

[30] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 7:1–7:12, New York, NY, USA, 2009. ACM.

[31] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *Trans. Storage*, 6(3):13:1–13:26, September 2010.

[32] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Netcat: Practical cache attacks from the network. In *S&P*, 2020.

[33] W. Leesakul, P. Townend, and J. Xu. Dynamic Data Deduplication in Cloud Storage. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pages 320–325, 2014.

[34] Jin Li, Xiaofeng Chen, Fatos Xhafa, and Leonard Barolli. Secure deduplication storage systems supporting keyword search. *J. Comput. Syst. Sci.*, 2015.

[35] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. Cachededup: In-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, Santa Clara, CA, 2016. USENIX Association.

[36] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. Nethammer: Inducing rowhammer faults through network requests. *arXiv preprint arXiv:1805.04956*, 2018.

[37] Jian Liu, N. Asokan, and Benny Pinkas. Secure deduplication of encrypted data without additional independent servers. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[38] Dirk Meister, Andre Brinkmann, and Tim Süß. File recipe compression in data deduplication systems. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 175–182, San Jose, CA, 2013. USENIX.

[39] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. *Trans. Storage*, 2012.

[40] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. Dark Clouds on the Horizon: Using Cloud Storage As Attack Vector and Online Slack Space. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[41] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. Cache time-behavior analysis on aes. *Selected Area of Cryptology*, 2006.

[42] Marco Oliverio, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Secure Page Fusion with VUsion. SOSP'17.

[43] R. Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. IPCCC'11, 2011.

[44] P. Puzio, R. Molva, M. Önen, and S. Loureiro. Cloudedup: Secure deduplication with encrypted data for cloud storage. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, 2013.

[45] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies*, FAST '02, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[46] Vladimir Rabotka and Mohammad Mannan. An evaluation of recent secure deduplication proposals. *J. Inf. Secur. Appl.*, 2016.

[47] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[48] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *ESORICS*, 2019.

[49] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote memory-deduplication attacks. In *NDSS*, 2022.

[50] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *CCS*, 2004.

[51] Z. Sheng, Z. Ma, L. Gu, and A. Li. A privacy-protecting file system on public cloud storage. In *2011 International Conference on Cloud and Service Computing*, 2011.

[52] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication As a Threat to the Guest OS. EUROSEC'11, 2011.

[53] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *USENIX ATC*, 2018.

[54] Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh. Last-level Cache Deduplication. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 53–62, New York, NY, USA, 2014. ACM.

[55] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.

[56] J. Xiong, Y. Zhang, S. Tang, X. Liu, and Z. Yao. Secure encrypted data with authorized deduplication in cloud. *IEEE Access*, 2019.

[57] Jia Xu, Ee-Chien Chang, and Jianying Zhou. Weak Leakage-resilient Client-side Deduplication of Encrypted Data in Cloud Storage. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 195–206, New York, NY, USA, 2013. ACM.

[58] Jia Xu, Ee-Chien Chang, and Jianying Zhou. Weak leakage-resilient client-side deduplication of encrypted data in cloud storage. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, 2013.

[59] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 121–128, Boston, MA, February 2019. USENIX Association.

[60] P. Zuo, Y. Hua, C. Wang, W. Xia, S. Cao, Y. Zhou, and Y. Sun. Mitigating traffic-based side channel attacks in bandwidth-efficient cloud storage. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

FusionFS: Fusing I/O Operations using $CISC_{Ops}$ in Firmware File Systems

Jian Zhang[*] *Yujie Ren*[*] *Sudarsun Kannan*
Rutgers University

Abstract

We present FusionFS, a direct-access firmware-level in-storage filesystem that exploits near-storage computational capability for fast I/O and data processing, consequently reducing I/O bottlenecks. In FusionFS, we introduce a new abstraction, $CISC_{Ops}$, which combines multiple I/O and data processing operations into one fused operation and offloads them for near-storage processing. By offloading, $CISC_{Ops}$ significantly reduces dominant I/O overheads such as system calls, data movement, communication, and other software overheads. Further, to enhance the use of $CISC_{Ops}$, we introduce MicroTx, a fine-grained crash consistency and fast (automatic) recovery mechanism for both I/O and data processing operations. Finally, we explore efficient and fair use of in-storage compute resources by proposing a novel Completely Fair Scheduler (CFS) for in-storage compute and memory resources across tenants. Evaluation of FusionFS against the state-of-the-art user-level, kernel-level, and firmware-level file systems using microbenchmarks, macrobenchmarks, and real-world applications shows up to 6.12x, 5.09x, and 2.07x performance gains, and 2.65x faster recovery.

1 Introduction

Modern high bandwidth and low-latency storage technologies such as NVMe SSDs [50] and 3D-Xpoint [6] have significantly accelerated I/O performance leading to better application performance. Yet, the combination of software and hardware I/O overheads that include system calls, data movement, and communication cost in the application and the OS, and the storage hardware latency (e.g., PCIe) continue to be an Achilles heel in fully exploiting storage hardware capabilities.

A recent focus is to reduce software indirections by moving filesystems to userspace and avoiding system calls and kernel traps for data and metadata updates [22, 60, 49, 27, 35]. Although effective, the dominating I/O overheads such as data and metadata movement cost, host and device communication cost (e.g., PCIe latency), and indirect costs like polling or interrupts remain. Henceforth, we refer to the combination of above overheads, which includes system calls, as dominating I/O overheads.

Another design point to reduce I/O overheads is the reincarnation of near-storage processing [46]. Vendors are introducing computational storage devices (**CSD**) that embed in-storage processors that range from ARM cores [11], FPGAs [49, 52, 30, 44, 47, 14] to RISC-V processors [51]. To reduce I/O cost and offload computations to CSD, recent research has explored application customization techniques [14], software runtimes [47], system software [49], and databases [46].

More broadly, these techniques can be classified into systems that focus on (1) in-storage compute offloading and (2) in-storage filesystems and key-value stores designed to accelerate I/O and storage management. First, in-storage compute offloading systems (which includes a majority of current CSD solutions) such as the seminal ActiveStore [46] for databases and recent approaches like PolarDB [14] focus on data processing by rewriting application logic to offload computation. While beneficial, these systems either lack storage management or delegate management to the host file system [14]. The former leads to a lack of data and metadata integrity, crash consistency, durability, or managing in-storage resources across tenants. In contrast, the latter incurs high I/O overheads for basic I/O operations and fails to utilize the full potential of CSDs. For example, in key-value stores like LevelDB [5], one could offload data compression to a CSD, but basic I/O operations would still incur system calls, data, metadata (e.g., inode, extents), and journal movement between key-value store, file system, and storage.

In contrast, in-storage management designs like CrossFS [44], DevFS [30], and Insider [47] offload filesystems and key-value stores [49] inside the storage firmware for direct-I/O, bypassing the OS. Unfortunately, these designs lack near-storage processing capability leading to substantial data movement and failing to manage in-storage resources such as device compute and memory or handle multi-tenancy.

We envision an ideal near-storage design that co-designs

[*]The authors contribute equally to this paper.

and combines storage management and data processing by rethinking I/O abstractions to reduce dominant I/O overheads, such as system calls, data and metadata movement, and host to device communication latency. Importantly, the design must ensure (storage) correctness, handle crash consistency, and achieve fairness across tenants.

We propose **FusionFS**, an near-storage file system design to exploit device compute and memory resources for reducing dominant I/O overheads and improving application performance. FusionFS provides fine-grained crash consistency, fast data recovery, and improves system efficiency by providing in-storage compute and memory fairness across tenants.

Towards the above goals, in FusionFS, we revisit I/O abstractions and take inspiration from seminal RISC (reduced instruction set computers) and CISC (complex instruction set computers) architectures. In FusionFS, a RISC operation is a simple POSIX file system operation (e.g., read, write, open) that can be directly offloaded to an in-storage filesystem (StorageFS), bypassing the OS. In contrast, our proposed CISC operations (hereafter referred to as **CISC$_{Ops}$**) are aggregated I/O and data processing operations offloaded as one operation to StorageFS for processing.

For generating CISC$_{Ops}$, we capture frequent I/O (e.g., *file-open-write-close*) and I/O + data processing sequences on a file (e.g., *append-checksum-write*) and combine them to one CISC operation. Intuitively, *aggregating I/O and data processing sequences and offloading them for near-storage processing* significantly reduces dominant overheads (system calls, data movement, and device and host communication costs). Note that CISC$_{Ops}$ support a combination of I/O and data processing operations and differ from traditional POSIX I/O vectors that are homogeneous (e.g., *readv*, *writev*).

Supporting in-storage RISC and CISC$_{Ops}$ introduces new challenges in terms of (1) applications changes, (2) crash consistency, and (3) resource management.

Application Support. FusionFS strives to reduce application changes by requiring minimal changes. First, a user-level library file system (UserLib) enables applications to use POSIX-like extensions for data processing or pack their custom command vectors supported by an in-storage file system (StorageFS). Optionally, FusionFS also provides mechanisms to transparently combine multiple I/O operations (without data processing) into a CISC$_{Ops}$ and offload them for in-storage processing, when feasible.

Fine-grained Crash-Consistency and Fast Recovery. In FusionFS, for traditional filesystem operations, we support journaling inside StorageFS. However, questions arise when packing multiple I/O and data processing operations in a CISC$_{Ops}$: (1) in what granularity should FusionFS support crash consistency? (2) how to exploit in-storage compute to accelerate recovery? For answering these questions, in FusionFS, we explore macro-transactions (MacroTx) and micro-transactions (MicroTx). MacroTx uses an all-or-nothing approach that only commits and recovers an entire CISC$_{Op}$ including the data processing state, whereas MicroTx supports crash consistency of partially committed CISC$_{Ops}$. Further, to reap the benefits of MicroTx, we go a step beyond current filesystems and use in-storage compute to support operational logging and automatic recovery by finishing partially completed CISC$_{Ops}$.

In-storage Resource Fairness. Next, offloading simple I/O operations and CISC$_{Ops}$ across tenants could exceed the in-storage compute (device-CPUs) and memory (device-RAM) resources. Therefore, there is a need for efficient and fair allocation of resources in ways that do not starve operations or tenants. Hence, in FusionFS, we borrow ideas from the Linux CPU scheduler, Completely Fair Scheduler (CFS) [1], to design a device-CPU and device-RAM CFS scheduler for enabling resource fairness and to reduce starvation.

End-to-end Evaluation. We evaluate FusionFS on a wide range of microbenchmarks, macrobenchmarks (Filebench [56]), and applications like LevelDB [5], Snappy compression [19], and Linux file encryption [2]. FusionFS, by using CISC$_{Ops}$ reduces dominant I/O overheads leading to 6.12X gains over the NOVA kernel file system [61], 6.12X over the user-level SplitFS, and 1.65X over the firmware-level CrossFS design. Application workloads like LevelDB [5] and Snappy compression [19] show gains up to 6.12X and 2.43X over user-level SplitFS. To highlight the benefits of CISC$_{Ops}$ as a general principle for kernel file systems, we extend ext4-DAX with CISC$_{Ops}$ and showcase the gains. Next, the proposed fine-grained crash consistency (MicroTx) combined with automatic recovery accelerates filesystem recovery by 2.65X. Further, CISC$_{Ops}$ support for LevelDB's restart-after-failure code accelerates recovery by 3.58X. Finally, the CFS-based device-CPU and RAM management reduce unfairness and improve storage efficiency.

The source code of FusionFS is available at `https://github.com/RutgersCSSystems/FusionFS`

2 Background and Related Work

Hardware Near-storage Processing Advancements. Although modern solid-state and nonvolatile memory storage devices have significantly accelerated I/O performance, software and hardware data access cost continues to be expensive. This has motivated hardware vendors to move away from legacy storage controllers with wimpy device cores for handling firmware functionalities (e.g., FTLs [33]) and support powerful in-storage compute. For example, ARM is introducing CSDs with Cortex-R82 64-bit 16-core processors (yet to be commercially available) [11] . In contrast, products like Newport CSDs with 16GB device-RAM, 1.5GHz 16 core processors, and TCP/IP stack support run Linux OS inside the CSD [20]. Finally, FPGA-based CSDs, such as SmartSSD [21], LSM-FPGA [14], ScaleFlux's CSD [52], implement fixed functions (e.g., filtering, compression, and encryption) and continue to evolve.

Software Innovation and Limitations. Software inno-

Properties	KernFS	UserFS	DeviceFS	Computing Offload	FusionFS
Direct-I/O	✗	Partial	✓	✗	✓
Reduce data copy	✗	Partial	Partial	Partial	✓
Reduce PCIe cost	✗	✗	✗	Partial	✓
In-storage Mgmt.	✗	✗	✓	✗	✓
In-storage process	✗	✗	✗	✓	✓
Durability	Data	Data	Data	Data	Data and compute
Resource Mgmt.	✓	✗	✗	✗	✓
Security	✓	Partial	✓	Partial	Same as KernFS

Table 1: Capabilities and Limitations of State-of-the-art Storage Approaches. The last column shows our proposed FusionFS.

vations for modern storage can be broadly categorized as (a) kernel file system (KernFS) and user-level file system (UserFS), (b) in-storage firmware file systems (DeviceFS) and key-value stores (DeviceKV), and (c) computational offloading (comp. offload) solutions mainly for processing. In Table 1(b), we qualitatively compare these designs.

Host-level KernFS and UserFS: Modern KernFS designs for fast storage devices reduce software indirections (e.g., page cache) and guarantee fundamental properties like crash consistency, security, and data sharing [58, 61]. Yet, the I/O overheads such as system calls, data movement, communication latency, and concurrency bottleneck continue to be a problem. An alternative trend is the re-introduction of UserFS designs aimed to bypass the OS (e.g., Strata [35], SplitFS [28], and others [42, 57, 37, 38]). While effective for applications that execute in isolation, a lack of a trusted computing base (e.g., OS) makes it challenging to handle security, data sharing, or multitenancy [30, 37, 38]. In contrast, hybrid designs like SplitFS [28], depend on the OS for metadata management, which could increase I/O overheads. Importantly, most UserFS designs fail to reduce data movement between the host and the storage and do not utilize in-storage compute.

Device-level File Systems (DeviceFS): As an alternative design point, prior work explored offloading file systems [30, 44, 45] and key-value stores [29, 49] inside CSDs and providing applications with direct-I/O. However, these designs generally lack data processing capability. DevFS [30] offloads file system into the firmware, whereas CrossFS [44] exploits parallel I/O queues for I/O scaling. CrossFS and DevFS reduce system calls, but data movement and communication costs remain. Prior solutions have also explored offloading key-value stores inside CSDs [49, 29, 13], which could benefit a specific class of applications that do not require file systems. Unfortunately, issues like high I/O overheads (e.g., data movement) and lack of near-storage processing and resources fairness remain in these designs.

In-storage Computation: In-storage computation systems primarily offload specific functions to the CSD. For example, seminal systems such as CASSM [54], RARES [36], and Active-Storage [46] offloaded database search and scan operations on slow hard drives. Recently, to benefit from fast storage, runtimes like LSM-FPGA [62], PINK [25], KEVIN [34] and others [11, 30, 44, 48] redesign and offload database compaction to FPGA-based CSD, whereas Newport OS deploys a specialized OS for offloading functions [20]. However, these systems lack storage management, do not handle critical storage properties like data durability, security, and sharing, and depend on the host-level file system.

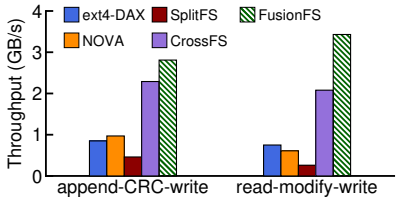


Figure 1: Analysis. Y-axis shows cumulative throughput when using 16 applicaiton threads.

Batching Operations. To reduce I/O cost, several I/O batching strategies have been proposed. Traditional file systems support vectored I/O, but it is restrictive and only supports data plane operations (e.g., writev, readv). Notably, all operations packed in a vector must be the same. Further, vectored I/O only provides coarse-grained all-or-nothing durability. Next, Chen et al. propose NFSv batch remote NFS I/O operations of the same type to reduce network latency [15], whereas TC-NFS [55] extended NSFv to support compound transactions. In contrast, FusionFS designs $CISC_{Ops}$ to reduce dominating I/O overheads by organically aggregating non-similar I/O sequences that could contain data-plane, control-plane, and processing operations. FusionFS also provides fine-grained durability and recovery without compromising in-storage resource fairness. We also showcase the benefits of $CISC_{Ops}$ for traditional file systems (§ 5).

3 Motivation

To motivate the need for reducing dominating I/O overheads like kernel/userspace crossing, data movement cost, and communication cost between host and device, we study the performance of state-of-the-art designs: KernFS ext4-DAX [58] and NOVA [61] (a log-structured design) designed for fast NVMs; hybrid UserFS SplitFS [27]; DeviceFS CrossFS [44]; our proposed FusionFS.

We use two workloads modeled after real-world applications: (1) an I/O-intensive *read-modify-write* that opens a 12GB file, continuously reads 4K blocks, updates, and writes them back depicting databases, key-value stores, and others; (2) an I/O + processing-intensive *append-checksum-write* (hereafter referred to as *append-CRC-write*) workload that appends data, computes checksum, and writes the data, replicating the behavior of several applications like key-value stores (LevelDB [7]), web-servers [56]. For brevity, we show results for 16 thread configuration of the benchmarks and show thread sensitivity in § 5. Because state-of-the-art systems use NVM as storage, we use a machine with 512 GB DC Optane NVM for storage, 64 CPUs, and 32 GB DRAM [6].

In Figure 1, the y-axis shows the throughput. First, kernel-level ext4-DAX provides direct access without data copies to page cache but incurs significant system call and data

copy cost for *read-modify-write* workload leading to substantially lower throughput. With its multicore-friendly and log-structured design, NOVA reduces some I/O overheads (e.g., avoiding double writes for a journal), but other overheads remain. Next, hybrid user-level SplitFS memory-maps storage to userspace and replaces reads/writes with loads/store operations. SplitFS reduces system calls, but metadata updates require frequent OS interaction. We also observe high OS locking and pre-paging costs for supporting user-level memory-map for 16 threads, leading to poor performance in both workloads. CrossFS, an emulated firmware-level file system design, reduces system calls and only metadata movement between filesystem and storage, resulting in higher performance. In contrast, FusionFS eliminates data movement significantly as well as host and device interaction by offloading both I/O and data processing. In § 5, we show the breakdown of FusionFS benefits for these workloads.

4 FusionFS Design

We next discuss FusionFS's design principles, followed by system architecture, mechanics for supporting $CISC_{Ops}$, support for fine-grained durability and fast recovery, permission management, and in-storage resource management.

4.1 Principles

1. Co-design in-storage management and data processing to eliminate dominating I/O overheads. We design a near-storage file system that combines storage management and data processing to reduce dominating I/O overheads such as system calls, data movement, and communication costs.

2. Design abstractions to reduce host and device interactions. We design $CISC_{Ops}$, a novel approach to fuse identical and nonidentical I/O and data processing operations. $CISC_{Ops}$ aggregate a sequence of I/O and processing operations and utilize device-CPUs to reduce data movement and communication between the host and the storage. We also explore an application-explicit and transparent approach (without data processing).

3. Exploit in-storage compute for fine-grained crash consistency and faster recovery. We design fine-grained crash consistency, micro-transactions (MicroTx), which persist all operations (including intermediate processing state) and reduce data loss in case after a failure. MicroTx uses an operational log and device-CPUs for automatic recovery by completing unfinished $CISC_{Ops}$ for faster recovery.

4. Manage in-storage resources for fairness and performance efficiency across tenants. We design a completely fair device-CPU and device-RAM scheduler (CFS) for fairness and for avoiding starvation across tenants.

4.2 System Architecture

To support direct-I/O design, FusionFS designs a user-level library (UserLib) and in-storage (StorageFS) components that work in tandem to offload I/O and computation without compromising correctness, crash consistency, recovery, security, and resource fairness. Figure 2 shows the high-level design of FusionFS. Applications issue traditional POSIX I/O requests or $CISC_{Ops}$ adding them to an inode-queue. StorageFS checks permission for each I/O request, dequeues, and schedules for processing, but importantly also provides fine-grained durability and recovery.

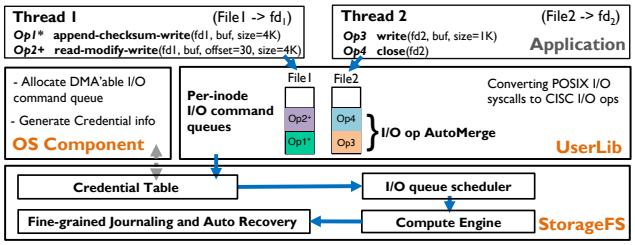


Figure 2: FusionFS High-level Design. Figure shows the high-level design of FusionFS with the UserLib, the StorageFS, and the OS components. For thread1, Op1 and Op2 show a $CISC_{Op}$ with data processing, whereas Op3 and Op4 show simple I/O. StorageFS shows the in-device structure with durability, permission, and scheduling components.

4.2.1 User and Device Components

We next discuss the details of user-level UserLib and in-storage StorageFS layers.

UserLib. This is an untrusted per-process user library to interface applications to the in-storage file system using host-CPUs. Beyond supporting POSIX I/O through interception and $CISC_{Ops}$, UserLib also sets up I/O queues, prepares requests, and handles errors.

I/O Commands. For regular POSIX I/O, application changes are not required, whereas $CISC_{Ops}$ require some changes to either use pre-defined UserLib $CISC_{Ops}$ or construct one. Table 2 shows select pre-defined Compute + I/O and I/O-only $CISC_{Ops}$. In § 4.3, we discuss principles and mechanics of constructing $CISC_{Ops}$ and the limited support for application-transparent I/O-only $CISC_{Ops}$.

inode-queue. To exploit modern NVMe's 64K hardware I/O queues and increase parallelism across files, we use a dedicated per-file I/O queue, referred to as inode-queue. The inode-queues buffer requests and intermediate data to be processed by StorageFS. The queues are created using a DMA'able memory region when opening a file [30, 44]. The OS maps the DMA regions, which can be accessed by the host and the device layers. By default, each inode-queue has 32 entries for inserting I/O commands and 2MB of data buffers, but the number of entries is configurable depending on host DRAM availability.

StorageFS. It is the heart of FusionFS design responsible for traditional file system functionalities like metadata and data management and permission control (§ 4.5). StorageFS houses compute engine (§ 4.3), supports traditional and fine-grained data and metadata journaling (§ 4.4) and recovery (4.4.1), and implements resource schedulers (§ 4.6). StorageFS is designed for general-purpose device-CPUs (e.g., ARM CSD) [11] and implements simple in-memory and on-disk filesystem structures such as a super-block, bitmap blocks, inode blocks, and data blocks (see Figure 2). For in-

Type	Ops. Sequence	Overheads			
		Data move	Syscall	Comm.	Resource
I/O-only	open, read, write	Hi	Hi	Med	Lo
I/O-only	open, read, close	Med	Hi	Hi	Lo
I/O-only	write, fsync	Med	Hi	Med	Lo
I/O-only	read, update, write	Med	Hi	Hi	Lo
Compute+I/O	write, CRC, write	Med	Hi	Hi	Med
Compute+I/O	read, CRC	Med	Med	Med	Med
Compute+I/O	read, compress, write	Hi	Med	Hi	Hi
Compute+I/O	read, encrypt, write	Hi	Med	Med	Hi

Table 2: Frequently used select I/O-only and I/O+Compute operation sequences and their overheads. **Hi**gh, **Med**ium, and **Lo**w indicate the relative magnitude of overheads. The columns, Data move, Syscall, Comm., and Resource denote data movement, system call, communication between host and device, and compute and memory usage, respectively.

storage computation, StorageFS parses through all operations in a $CISC_{Op}$ vector, executes, and returns operation-specific return codes. Internally, the StorageFS compute component currently supports several data processing operations like checksum, compression, encryption, and decryption, beyond just parsing and sorting operations. To address the lack of programmable hardware, StorageFS prototype is currently implemented as a device driver with separate CPUs, memory regions, and disk with carefully emulated hardware parameters. We next discuss the details of each StorageFS component.

4.3 Operation Types and $CISC_{Ops}$ Mechanics

We first discuss operation types to offload, followed by the mechanics for offloading.

4.3.1 Operation Types

Applications generally access storage using (1) simple I/O operations to store or read state, (2) issue a sequence of I/O operations, or (3) access, process, and store data. The processing could vary from operations like compression, encryption, checksumming, or complex transformations that search, sort, or even execute ML operations (e.g., add, multiply, XOR). Reducing I/O overheads across all such operation types is critical.

Offloading Simple I/O Operations. For basic POSIX I/O, UserLib intercepts system calls and adds them to inode-queues. We extend the NVMe commands to add new operation codes for representing filesystem operations (e.g., read, write, open, close). After adding a request (command), UserLib sets a doorbell for the StorageFS to start processing, which sets the request's commit flag after completion. All blocking (e.g., *read*) and non-blocking (*write*) data plane operations are added to inode-queues, whereas metadata-intensive operations (e.g., *mkdir*) use a separate global metadata I/O queue. For error-prone operations like file and directory rename [12], FusionFS uses global file system locks.

$CISC_{Op}$ Operations. We next discuss UserLib support to identify and aggregate identical and non-identical I/O and data processing operations.

Identical and Non-identical I/O Operations. We observe that in several applications, I/O operations are executed in sequence or pairs. For example, Figure 3(a) shows a widely-used NoSQL database and webserver sequence that opens, writes, syncs, and closes the file when inserting values (i.e., open()->write()->sync()->close()) or when reading data [5]. The figure also shows overheads for each operation, which includes system calls between application and the OS (S), data movement (D), metadata movement (M) such as inode, and host-storage communication (PCIe or memory bus) cost (C). Note that the direct-access (DAX) filesystems for NVM incur one less data copy compared to the block-level file system. We observe several such sequences contributing to I/O overheads as listed in Table 2. In contrast, $CISC_{Ops}$ aggregates and offloads such sequences to StorageFS reducing I/O overheads (see Figure 3(b)).

I/O and Data Processing Operations. For supporting data processing + I/O operations, as opposed to full application redesign [14] for CSDs, we aim to reduce application changes for organically supporting I/O and their related pre and post-processing to reduce I/O overheads. Specifically, as shown in Table 2, we observe that applications frequently fetch I/O data to perform operations like checksum generation (CRC), compression/decompression, encryption, search, sort, and ML operation pairs (e.g., XOR, multiplication). For example, as shown in the code snippet in Figure 3(c), NoSQL persistent stores like LevelDB avoid expensive file commits or propagation of corrupted data by adding CRC for integrity check. After each file system append() system call, the CRC is computed and appended to the actual data. This sequence incurs 2 system calls, 4 data and metadata copies (2 for DAX file systems) , and 2 PCIe operations in OS file systems (see Figure 3(a)). The above sequence repeats for all data reads or replication to other nodes to check data integrity. In contrast, an *append-CRC-write* $CISC_{Ops}$ offloaded to StorageFS significantly reduces I/O overheads to 1 data movement and a PCIe operation without incurring system calls (see Figure 3(b)).

4.3.2 Mechanics for Application-explicit Support

With the explicit approach, applications can use $CISC_{Ops}$ pre-constructed by UserLib or construct custom $CISC_{Ops}$.

$CISC_{Op}$ Command Structure: Each $CISC_{Op}$ is a vector of commands in an extended NVMe format [59] for supporting multiple operations and added to inode-queue for processing. For example, Figure 3(d) shows the code snippet for packing a *append-CRC-write* $CISC_{Op}$ in LevelDB. Each $CISC_{Op}$ vector element contains an operation code (*opc*), input and output address to specify DMA'able address from which data must be fetched or stored (*in*), I/O offset (*slba*), block count (*nlb*), and return code (*retcode*), and a journal commit flag (*commit*). The number of elements in the $CISC_{Op}$ is configurable, and by default, can pack *32* operations to fit in a DMA-able page. Furthermore, FusionFS could also combine multiple $CISC_{Ops}$ to batch operations.

Specifying Dependency. Applications or UserLib developers can specify inter-dependencies across operations in a $CISC_{Op}$. For example, as shown in Figure 3(e), for the *append-CRC-write*, the CRC depends on the input of previous *append* and the bytes actually written to storage, which is unknown until *append* completes. The input dependencies can be specified

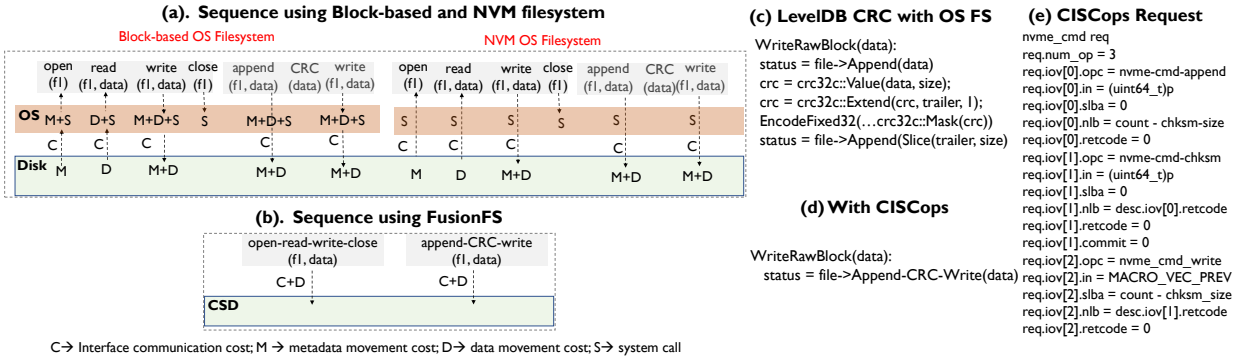


Figure 3: Comparison of I/O Overheads. (a) and (b) compare data movement (D), communication cost (C), and system call (S) using traditional storage (top) and envisioned $CISC_{Ops}$ design that bypasses OS (bottom); (c) and (d) show CRC (checksum) code sequence in vanilla LevelDB (top) and proposed ($CISC_{Ops}$); (e) shows packing a $CISC_{Ops}$ request. For NVM direct-access filesystems, the data copies are reduced to one per operation. Metadata caching could reduce I/O.

using the DMA address (e.g., *req.iov[1].in = (uint64_t)p* for checksum input) and the bytes to process (*req.iov[1].nlb = desc.iov[0].retcode*).

Concurrency and Ordering. FusionFS supports out-of-order processing for concurrency by default (e.g., vectored writes or combination of reads and writes without conflicts) as well as in-order processing (e.g., *append-CRC-write*). To prevent out-of-ordering, the $CISC_{Op}$ structure allows specifying an "order" field to execute the operation sequentially (e.g., *req.iov[1].order = MACRO_VEC_PREV*), which are otherwise parallelized in the presence of free device cores.

4.3.3 StorageFS Compute Engine

Inside the storage, StorageFS implements a generic parser to disassemble $CISC_{Ops}$ and either execute them with file system logic for basic I/O or use the compute engine for data processing. We have currently added new data processing functionalities to the compute engine, which would require firmware upgrade [53]. However, we will explore alternative dynamic code injection techniques (e.g., eBPF [4]).

4.3.4 Partial Support for Automatic Offloading

FusionFS enables a partial support for transparently generating, merging, and offloading $CISC_{Ops}$ for a group of I/O-only operations without data processing, referred to as *AutoMerge*. This is useful when application changes are not feasible. AutoMerge primarily reduces system calls, host-to-device interaction, and overheads such as command generation, I/O queuing, and scheduling but not necessarily data movement. AutoMerge can either merge non-dependent operations on the same file by different threads or asynchronous operations. UserLib parses all pending operations in a file's inode-queue to generate $CISC_{Ops}$. For example, consider multiple *append-CRC-write* to different blocks of the same log file across threads or a sequence of asynchronous writes. AutoMerge can aggregate two non-dependent operations – *append* in a *append-CRC-write* with *write* in previous *append-CRC-write* – to generate $CISC_{Ops}$. We study the benefits and implications of AutoMerge in § 5.

Limitations. While our AutoMerge provides simple batching, it currently lacks support for offloading data processing operations (a harder problem). Similarly, it is ineffective for single-threaded applications with blocking I/O. We plan to explore automatic data processing offloading (a harder challenge) and other limitations in our future work.

4.4 Supporting Durability and Fast Recovery

We next discuss the support for basic journaling and fine-grained crash-consistency support for offloaded POSIX I/O and $CISC_{Ops}$, respectively, followed by the support for automatic recovery after failure by utilizing in-storage compute.

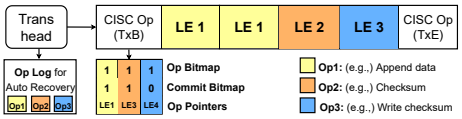


Figure 4: Micro-transaction (MicroTx) Design. Figure shows an example of *append-CRC-write*.

4.4.1 Traditional Journaling

For POSIX I/O, StorageFS supports traditional REDO journaling. The journal log-entry headers contain a unique transaction ID followed by the metadata log (with a pointer to actual data) and a transaction commit bit. Before a transaction commits, first the data is updated in place, followed by metadata logging.

To support crash consistency for $CISC_{Ops}$, FusionFS first provides an "all-or-nothing" macro-transaction (MacroTx), which wraps an entire $CISC_{Op}$ into one transaction and recovers only if the entire $CISC_{Op}$ is committed (to the log) before the failure. This simple approach resembles crash-consistency support in today's vectorized I/O. However, this approach risks losing I/O data and computation state when $CISC_{Ops}$ do not complete.

4.4.2 Durability & Automatic Recovery with MicroTx

We overcome the limitations of MacroTx by designing MicroTx that provides fine-grained durability of all I/O and processing operations of a $CISC_{Op}$, which we refer to as micro-ops. Each micro-op can be independently committed and recovered after a system failure. Figure 4 illustrates a MicroTx structure with *append-CRC-write*, where a log-entry for append, checksum, and write micro-ops are independently committed. Each $CISC_{Op}$ log entry uses a bitmap in *TxB* to represent the number of micro-ops (micro-op bitmaps) and the address offset of log entries for each micro-op. *TxB* also

contains a commit bitmap to mark and track committed micro-ops, and *TxE* represents a bit to indicate the completion of an entire $CISC_{Op}$. Because some compute operations (e.g., compression) could have a state larger than the available log entry size (e.g., 48 bytes by default), users can configure log-entry size during filesystem mount. We will explore dynamic log entry sizes in our future work.

Next, we utilize MicroTx and device-CPUs to design an automatic recovery mechanism and redo incomplete $CISC_{Ops}$. This is in contrast to current OS file systems that rely on applications to redo failed operations, which increases recovery time and developer efforts. For auto-recovery (optionally enabled during filesystem mount), MicroTx additionally uses operational log (shown in Figure 4) to write $CISC_{Ops}$ and the input data similar to data logging before processing a request. After logging, a "commit" flag is set and used as a receipt by an application. In case of a failure, MicroTx's recovery first recovers all committed micro-ops, followed by recovering $CISC_{Op}$ and input data using the operational log and then executes all incomplete micro-ops in $CISC_{Ops}$. In Table 10c, we show the benefits of automatic recovery in reducing recovery and restart time after a system crash or failure by reducing I/O costs. Importantly, MicroTx and automatic recovery could reduce application/developer effort to check and redo incomplete I/O operations.

4.4.3 Error Handling

To handle errors (e.g., insufficient disk space), for application-explicit $CISC_{Ops}$, when using an all-or-nothing MacroTx, FusionFS aborts the entire sequence and also updates the return code for the operation that caused the failure. In contrast, when using MicroTx, all operations starting from the erroneous micro-op are aborted with an error return code. However, FusionFS could potentially execute all subsequent non-dependent operations in the sequence. For example, in a $CISC_{Op}$ with 10 writes, a large write (say, the 6th write) could fail due to the lack of disk space, but subsequent smaller writes (7 to 10) could succeed. For the transparent AutoMerge, because applications expect independent execution of micro-ops, we allow execution of all I/O operations.

Potential (infrequent) errors could occur during automatic recovery (say after a system crash and restart). With MicroTx and operational log enabled, we retain the journaled micro-ops, abort the erroneous operation, and use operational log to report the failure with file name, operation type, and error type, which is later checked by UserLib to identify errors. Beyond our current design, a careful exploration of opportunities to reduce $CISC_{Op}$ aborts and failures, and correctness issues is critical.

4.5 Permission Checking and Data Sharing

We aim to match the security guarantees of OS file systems for both POSIX I/O and $CISC_{Ops}$ by satisfying the following assertions: (1) only processes with the right permission can access a file or directory; (2) the file system metadata is updated only by a trusted entity; (3) for inter-processes file sharing, only legal writers can update the data.

FusionFS achieves these goals by utilizing trusted OS but without compromising direct I/O. First, the OS shares credential information of a process with StorageFS during process initialization, and StorageFS maintains a per-process credential table similar to prior designs [44]. Second, inode-queues and their DMA'able memory regions (e.g., when opening files) are created by the OS only when requested by a process with the right credentials. Third, access to inode-queues (i.e., DMA'able memory pages) is protected by virtual memory protection, preventing illegal access by a malicious process. Finally, the OS shares credentials with StorageFS. For all requests in the inode-queue, StorageFS checks permission for operations packed in a $CISC_{Op}$ by comparing against the credential list before processing, thereby avoiding partial execution. For example, in a *read-compress-write* $CISC_{Op}$ issued to a read-only file, FusionFS' permission manager does not allow partial $CISC_{Ops}$ execution.

Data Sharing. Supporting direct-I/O via user-level library and inode-queues complicates secure inter-processes file sharing. When a file is shared and accessed across readers and writers, the inode-queues used for dispatching requests are also shared. Unfortunately, a reader process (with read-only permission) could accidentally or maliciously corrupt I/O or data processing requests issued by writers. To overcome these complexities, we employ the following design. First, all legal writers (without readers) can concurrently access and update a file, similar to OS file systems. Second, similar to KernFS and UserFS designs, applications are responsible for ordering updates to an inode-queue (e.g., using lease-based locks [35]). However, in the presence of readers (a file opened with read-only permission), to prevent corruption of writer requests in the inode-queue, FusionFS detects file opened with read-only permission and delegates the trusted OS to add I/O commands from both writers and readers after permission checks.

4.6 Resource Management

We introduce in-storage compute and memory-centric scheduling to enable in-storage resource fairness across tenants, avoid starvation, and improve performance efficiency.

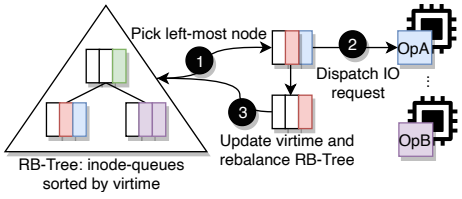


Figure 5: FusionFS scheduler high-level overview

4.6.1 FusionFS Compute-Centric Scheduling

The number of application threads that issue POSIX I/O and $CISC_{Ops}$ could exceed available CSD compute cores. Specifically, compute-intensive data processing (e.g., checksum, compression) will increase in-storage compute use, leading to workload imbalance, starvation, and impacting the

performance of POSIX I/O across tenants. Unfortunately, conventional I/O schedulers and prior in-storage schedulers are I/O centric (e.g., Linux *blk-queue*) and fail to consider device-CPU usage.

In-storage CFS Compute Scheduler. In FusionFS, we take inspiration from the OS CFS CPU scheduler and explore its use for device-CPU scheduling [1]. We account for I/O and data processing operations that use device-CPUs. Figure 5 provides a high-level overview. At a high level, for CPU fairness, StorageFS scheduler selects and dispatches requests from inode-queues (of processes) with the least CPU usage (i.e., virtual CPU runtime). However, unlike OS schedulers, in-storage StorageFS lacks process state, which makes bookkeeping challenging, specifically for keeping track of virtual runtime (*virtime*) usage of device-CPUs by each process. We overcome this by using the inode-queues of an application that buffers I/O requests to track and bookkeep device-CPU usage. For each I/O or data processing request dispatched from an inode-queue, FusionFS increments the *virtime*, and selecting a request from an inode-queue with the least *virtime*.

Internally the scheduler maintains a global red-black tree (RB-tree) that stores reference to inode-queues, sorted by their *virtime* (see Figure 5). Initially (after mount), the scheduler uses a round-robin approach to pick an inode-queue, but once requests are dispatched, their *virtime* are updated, and the tree is rebalanced. Note that the scheduler always dispatches requests from the left-most RB-tree node that points to inode-queue with the least *virtime*. While we currently implement a simple RB-tree, we will explore alternative device-optimized (i.e., firmware) data structures [47].

Fairness Across Tenants. We employ a two-pronged approach to prevent a greedy process (tenant) from increasing compute share by increasing inode-queues and starving other tenants. First, the trusted OS tags each inode-queue with a process ID, and StorageFS maintains the overall *virtime* of each process using the sum of all its inode-queue's *virtime*. The scheduler always selects a request from an inode-queue with the smallest process-level and inode-queue-level *virtime*. Second, an administrator can limit the number of inode-queues per process.

Request Termination and Preemption. To handle misbehaving or long-running requests with large inputs, StorageFS first terminates the request for avoiding starvation, then sets error codes for the request, finally clearing the transaction states (logs). Our ongoing work is exploring request preemption, which requires committing intermediate state of CISC_{Ops} using MicroTx and switching to other operations. However, preemption introduces correctness challenges. For example, preempting an *append-compress-write* after the compress operation could lead to incorrect read operations on the same file other threads.

4.6.2 Memory-Centric Scheduling

Beyond device-CPUs, efficient management of device-RAM is critical for fairness. Although modern CSDs are equipped with 4-16GB of memory, a combination of in-storage data processing, filesystem operations, and FTL's logical to physical block translations could increase memory contention and starvation across clients [16, 43]. For example, offloading memory-hungry file compression with large inputs could starve or block other CISC_{Ops} (e.g., *append-CRC-write*) or POSIX I/O from other tenants even when free compute cores are available.

We overcome the above challenges by extending the CFS to share device memory capacity efficiently. First, we implement a simple slab allocator for allocation and deallocation. Next, we enhance the CFS scheduler with memory usage (*memuse*) accounting for each process and inode-queues and maintaining a memory-specific RB-tree with per-inode *memuse*. When device CPUs are not a bottleneck, the scheduler selects a process and inode-queue with the least *memuse*; this avoids blocking or failing other requests. Finally, in § 5, we evaluate the benefits of FusionFS' memory-centric scheduling with the multitenant workload. An ideal scheduler must provide multi-resource fairness across compute, memory, and other resources (e.g., using Dominant Resource Fairness [23]), which we will explore in our future work.

4.7 Emulation and Application Changes.

Due to a lack of a programmable storage device, FusionFS is implemented as a device driver. We us Intel Optane Memory [6] for storage similar to prior work [30, 44, 26]. We emulate device-CPU and device-RAM by using dedicated CPUs and memory managed by StorageFS. To emulate PCIe latency, we add 900-1000ns delays [40] for all interactions between the host and the device. Finally, the storage and the device-RAM bandwidth could vary across vendors. To understand the implications, we use DRAM thermal throttling and study the impact [32, 24]. Note that bandwidth throttling works only for DRAM technology and in older Intel Haswell architectures. Therefore, we use a multi-socket DRAM-based system to emulate and vary memory and storage bandwidths.

We built FusionFS by extending CrossFS' direct-I/O support and adding CISC_{Ops}, fine-grained journaling, fast and automatic recovery, efficient and fair in-storage resource management, and optimizations to improve device-compute scalability. UserLib and StorageFS components add 3K and 11K LOC, and the data processing operations like compression, checksum, and decryption functions add 2.4K lines of code. Finally, to use CISC_{Ops}, LevelDB requires < 38 LOC changes replacing the CRC logic, whereas file encryption and snappy compression require < 21 LOC changes.

5 Evaluation

Our evaluation answers the following questions:

- How effective is FusionFS and its CISC_{Ops} abstraction in reducing I/O overheads across microbenchmarks and macrobenchmarks?
- How sensitive are FusionFS gains towards device-CPU frequency and memory and storage bandwidth?

CPU	Intel Xeon(R) Gold 3.1GHz, dual-socket, 64-core
DRAM	96GB DDR4 2666 MT/s
NVM	256GB Intel Optane DC PMM (2*128GB)
Device-CPU	4-cores, Fast (2.7GHz) and Slow (1.2GHz) CPUs
Device-RAM	2GB dedicated for device operations
PCIe Latency	900us

Table 3: Experiment Platform and Setup. The device-CPUs and device-RAM are emulated through DVFS and thermal throttling.

$CISC_{Ops}$	**ext4-DAX**	**ext4-DAX-$CISC_{Ops}$**	**FusionFS**
append-CRC-write	0.85 GB/s	1.18 GB/s	2.81 GB/s
read-modify-write	0.75 GB/s	0.84 GB/s	3.43 GB/s

Table 4: $CISC_{Ops}$ under traditional ext4 file systems.

- Is $CISC_{Ops}$ effective for traditional OS filesystems?
- Can MicroTx and auto-recovery improve durability and accelerate recovery?
- How effective is the CFS-based compute and memory scheduler in improving resource fairness across tenants?
- What is the overall impact of FusionFS on real-world applications?

5.1 Experimental Setup and Methodology

Due to the lack of programmable CSD, we carefully emulate our FusionFS with the parameters shown in Table 3. Our Optane NVM storage provides 8 GB/sec read and 3.8 GB/sec write bandwidth. We compare. For StorageFS processing, we reserve 4 CPUs [50]. We also study the impact of varying CPU speeds using fast (2.7GHz and default) and slow (1.2GHz) CPUs resembling ARM-based CSDs [11]. For device-RAM, we reserve 2 GB memory managed by StorageFS. We study the impact of device-RAM bandwidth using a 64-core CloudLab machine. For PCIe latency, we add 900ns [40] delay before a request is processed.

Methodology. We compare FusionFS against the state-of-the-art KernFS – ext4-DAX [58] and NOVA [61], hybrid UserFS – SplitFS [28], and DeviceFS – CrossFS [44]. Note that some file systems do not support macro-benchmarks and applications evaluated in this paper. The throughput shown for workloads (in GB/s) combines data (payload) I/O and processing times.

5.2 Benchmark Analysis

We evaluate microbenchmarks and Filebench macrobenchmark [56] to understand $CISC_{Ops}$ benefits and implications.

5.2.1 Microbenchmark

We evaluate I/O data and metadata intensive *file-open-write-close* in Figure 6a, I/O and data processing intensive *append-CRC-write* in Figure 6b, and data plane heavy *read-modify-write* benchmark in Figure 6c. The *file-open-write-close* is modeled after NoSQL databases, file-servers, and web-servers that operate on several files. The workload opens a file, performs a 2MB write, and closes the file, repeating this for 10K files. Next, the *append-CRC-write* benchmark (as discussed extensively in this paper) is used for providing integrity in NoSQL databases [5, 3], key-value stores [10], and others. Each thread appends a 4KB block, computes the checksum, and writes the checksum on a 12GB file. The data plane-intensive *read-modify-write* mimics several widely-used applications [5, 3, 10, 56] by reading a random 4KB block, modifying with random text, and writing back data blocks on a 12GB file.

Workload	**Syscall I/O**	**Direct-I/O**	**Direct-I/O + $CISC_{Ops}$**	**Direct-I/O + $CISC_{Ops}$ + CFS-sched**
append-CRC-write	1.15 GB/s	2.23 GB/s	2.74 GB/s	2.81 GB/s
read-modify-write	0.75 GB/s	1.91 GB/s	2.85 GB/s	2.98 GB/s

Table 5: Breakdown of FusionFS incremental gains.

# of threads	**1**	**2**	**8**	**16**
ext4-DAX	0.26 GB/s	0.66 GB/s	0.99 GB/s	0.85 GB/s
FusionFS-AutoMerge	0.26 GB/s	0.93 GB/s	1.66 GB/s	2.70 GB/s
FusionFS	0.27 GB/s	0.95 GB/s	1.94 GB/s	2.81 GB/s
FusionFS-Batch	0.29 GB/s	1.05 GB/s	2.06 GB/s	2.98 GB/s

Table 6: FusionFS Optimizations. (append-CRC-write).

Methodology. We vary the number of benchmark threads in the x-axis, and the y-axis shows the throughput (GB/sec), and the threads use separate files. We compare ext4-DAX, NOVA, SplitFS, CrossFS, and FusionFS. Additionally, to understand the impact of slower device-CPUs, we also evaluate in-storage StorageFS to use 1.2GHz device-CPUs (*CrossFS-slow-device-cpu* and *FusionFS-slow-device-cpu*).

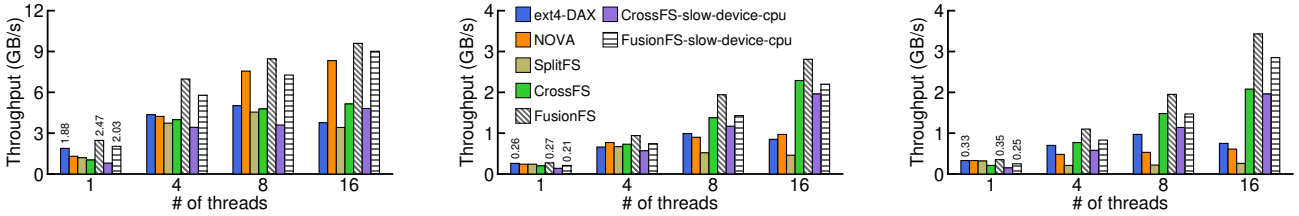
Observation. As shown in Figure 6a, in KernFS designs ext4-DAX and NOVA, each I/O operation incurs system call, data copy, and the device communication latencies, resulting in high I/O overheads. However, NOVA performs better than ext4-DAX due to its log-structured design and per-CPU (multicore-friendly) block management. Next, SplitFS, a hybrid UserFS, memory-maps staging files to userspace and performs load and store operations. However, SplitFS uses the OS for metadata operations. Specifically, we observe increased kernel overheads that increase with workload size and thread count from metadata operations, internal data copies, and block lookup and pre-paging (MAP_POPULATE) cost for the userspace mmap'ed files that stage I/O.

In contrast, in-storage CrossFS bypasses the OS and avoids system calls but suffers from data copies between the host and the device (for read and write I/O), PCIe latency (hardware), and the software cost to allocate, enqueue, and dequeue requests. The blocking *reads()* also stall the host CPUs.

Finally, FusionFS merges the open->write->close into a *file-open-write-close* $CISC_{Ops}$, avoids system calls, reduces a data copy between the application and the OS, and the PCIe latency, all leading to up to 4.58X gains over ext4-DAX, 6.12X over SplitFS, and 1.65X over CrossFS. Table 5 shows the incremental benefits of FusionFS' design optimizations.

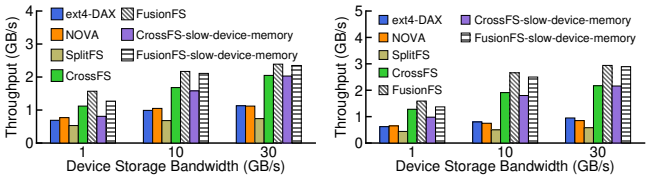
Next, as shown in Figure 6b, for *append-CRC-write*, similar to *file-open-write-close*, prior approaches lack in-storage compute capability for CRC, resulting in two system calls (except CrossFS) and a data copy. In contrast, FusionFS improves performance over ext4-DAX, SplitFS, and CrossFS by up to 3.3X, 6.1X, and 1.3X, respectively. Finally, for *read-modify-write*, FusionFS outperforms all other systems.

Device Compute Speed. In Figure 6, we show *FusionFS-slow-device-cpu* and *CrossFS-slow-device-cpu* configurations using slower device compute by throttling them to 1.2GHz. FusionFS outperforms CrossFS and, importantly,



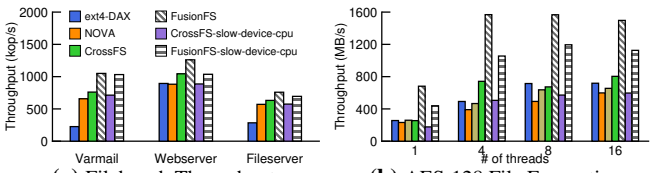
(a) Open-Write-Close, 10K files, 2MB each file **(b)** Append-CRC-Write **(c)** Read-Modify-Write

Figure 6: Microbenchmarks. Shows aggregated throughput. Threads operate on separate files. CrossFS and FusionFS use 4 device cores.



(a) Append-CRC-Write **(b)** Read-Modify-Write

Figure 7: Sensitivity to Device Bandwidth. The x-axis shows the device storage bandwidth. For CrossFS and FusionFS, we show results when using slow device memory throttled to same bandwidth as storage.



(a) Filebench Throughput. **(b)** AES-128 File Encryption

Figure 8: Macrobenchmark. (a) shows the Filebench performance with three workloads. (b) varies the number of threads when encrypting target files

other filesystems that use 2.5× faster host CPUs, highlighting the importance and benefits of reducing I/O overheads.

Impact of Memory and Storage Bandwidth. The bandwidth of device storage and device-RAM could vary across vendors. To understand the sensitivity, in Figure 7, we use DRAM thermal throttling to study the impact. As discussed earlier, we use a 64-core dual-socket CloudLab machine [17] that uses DRAM as storage. In Figure 7, we vary the storage bandwidth from 1GB to 30GB (maximum without throttling) along the x-axis. The bars *CrossFS-slow-device-memory* and *FusionFS-slow-device-memory* represent the case where both CrossFS and FusionFS use slower but the same device-RAM and storage bandwidth by pinning them to a slower NUMA socket.

Observation. FusionFS consistently delivers higher gains. For example, when varying the storage bandwidth without slower device-RAM results in 1.35x and 3.09x gains over CrossFS and ext4-DAX. Next, when the device-RAM bandwidth is also reduced, FusionFS' throughput reduces but is still higher than other systems. *The results show that dominating I/O overheads add more constraints for storage-intensive workloads, and reducing them is critical.*

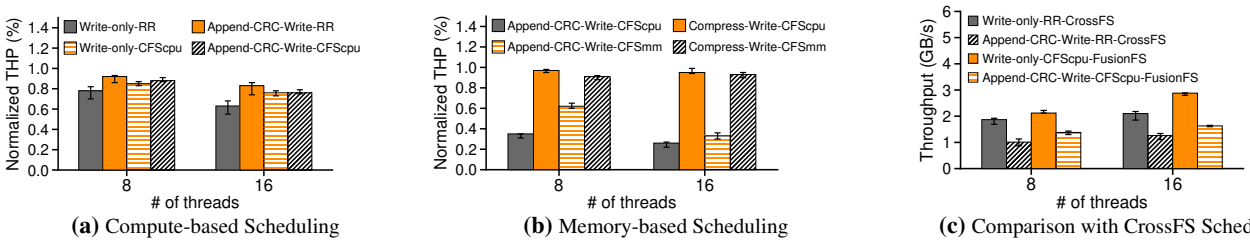
Effectiveness of CISC$_{Ops}$ for ext4-DAX. To understand the effectiveness of CISC$_{Ops}$ as a general principle for all file systems, we extend ext4-DAX to support *append-CRC-write* and *read-modify-write* CISC$_{Ops}$ and compare the throughput against vanilla ext4-DAX and FusionFS in Table 4. The vanilla ext4-DAX incur two system calls and two data moves (see Figure 3a) for both workloads. Next, ext4-DAX-CISC$_{Ops}$ reduces system calls by half with one data copy leading to better throughput. In contrast, FusionFS eliminates all system calls and one data copy leading to 4.08x higher throughput.

Optimizations: Application Explicit Batching and Transparent CISC$_{Ops}$. First, FusionFS can batch multiple, non-dependent CISC$_{Ops}$ as a vector to eliminate data copy, system call, and other software overheads, such as enqueuing and dequeueing requests. The number of requests to batch depends on the available DMA memory used as a data buffer for each inode-queue (a configurable parameter in FusionFS). Table 6 shows the performance for explicitly batching 10 CISC$_{Ops}$ (*FusionFS-Batch*) and varying the number of threads. As shown, FusionFS with batching shows 5.02x and 1.45x gains, respectively, compared to not batching.

Next, as discussed in §4.3.4, FusionFS provides partial support for transparently generating and offloading CISC$_{Ops}$ by aggregating non-dependent and pending I/O requests in an inode-queue, mainly for asynchronous I/O or requests across multiple threads but without offloading data processing. As shown in Table 6, *FusionFS-AutoMerge* provides gains over ext4-DAX for higher thread counts, but as expected, the application-explicit approach outperforms all cases.

5.2.2 Macrobenchmark - Filebench

To validate the microbenchmark gains, we next evaluate FusionFS for the widely-used Filebench [56] in Figure 8a. We use the *fileserver*, the *webserver*, and the *varmail* workloads. The fileserver opens a file, randomly appends 16K bytes, and closes the file. In FusionFS, these operations are aggregated to one *file-open-write-close* and offloaded using a temporary file's inode-queue. The webserver opens a file, reads the whole file, and closes it, which is aggregated to *open-read-close*. Finally, varmail issues a combination of file create, write, sync, and close operations and open file, read, and close file operations, which are aggregated to *open-write-close* and *open-read-close*, respectively. The I/O sequences are repeated on thousands of files by 16 worker threads. The workloads are metadata-heavy issuing file create, delete, directory update operations that contribute to 69%, 63%, and 64% of the overall I/O in the varmail, the fileserver, and the webserver workloads, respectively. FusionFS and CrossFS outperform other file systems by eliminating system calls, reducing data movement, communication, and software costs such as queuing delays, delivering 36% gains for webserver workload over ext4-DAX. Furthermore, despite NOVA's multicore parallelism friendly design, reducing I/O overheads is



(a) Compute-based Scheduling **(b)** Memory-based Scheduling **(c)** Comparison with CrossFS Scheduler

Figure 9: Scheduler. (a) shows the normalized throughput (THP) for each workload relative to no-sharing of device-CPU resources. (b) shows the throughput factor relative to the no-sharing of device-memory resources. The device memory budget is set to 2GB. (c) compares CPU scheduling against CrossFS.

critical, as showcased by *FusionFS-slow-device-cpu* gains.

5.2.3 Macrobenchmark - File Encryption

We next study FusionFS on widely used Linux encryption and decryption service, Cryptsetup [2]), which is used by several applications (e.g., OpenSSL [8]). Cryptsetup uses widely-used AES-128 [39] symmetric block cipher to encrypt files. The application threads read data from a 4GB file, encrypt, and write back the encrypted data. In Figure 8b's x-axis, we vary the application's thread count. For FusionFS, we replace the read, encrypt, write sequence with *read-encrypt-write* CISC_{Ops}. FusionFS, by aggregating operations, reduces I/O cost and outperforms its counterparts achieving up to 2.48X gains over NOVA.

5.3 Crash Consistency and Recovery

We next study FusionFS' crash consistency and recovery capabilities. We use (1) a *append-CRC-write* workload and (2) a *vector-write* (ten writes) benchmark. For both, we inject failures at different points to test durability, as shown in Table 10a. For the *append-CRC-write*, we add three failure states (*F1*, *F2*, and *F3*), whereas for the vectored write, we inject a failure before any writes (*F4*), a failure between writes 4 and 5 (*F5*), and a failure after all the writes complete (*F6*). In Table 10b, we show crash-consistency correctness of committed and uncommitted operations for different file systems. Note that FusionFS offers the basic all-or-nothing (*MacroTx*), and an optional (and optimistic) auto-recovery, *AutoRec* that uses *MicroTx* and operational log for fast recovery.

First, as shown in Table 10a, for the *append-CRC-write*, for the case when the failure happens before *append* commits (*F1*), ext4-DAX, NOVA, SplitFS, CrossFS, and FusionFS's MacroTx provide crash consistency excluding the uncommitted append (C). In contrast, FusionFS's AutoRec provides operational logging; hence during restart, it can recover and re-execute *append-CRC-write* to completion if a valid operational log entry exists, as indicated by the (C/R) state. Similarly, for a failure after checksum (*F2*), AutoRec recovers both append and checksum's state from MicroTx, and finishes writing, providing better recoverability after failure. Next, *F4-F6* shows the failure points for the vectored 4KB write workload. When a failure occurs at *F5* (partial writes of a vector), all file system approaches, excluding FusionFS' AutoRec, do not recover due to their all-or-nothing approach restoring the file system to a consistent state. In contrast, AutoRec uses MicroTx with fine-grained commits, recovers partially completed writes in a vector (C/R), and finishes the vectored write.

Recovery Time. To study the impact on recovery time, in Table 10c, we run the *append-CRC-write* workload with 16 threads that issue 16MB appends and inject failures at crash points *F1* (before append) and *F2* (after checksum). For ext4-DAX without data atomicity, applications must re-execute the entire operation sequence and incur system calls and data movement costs, also increasing recovery time. In contrast, NOVA and SplitFS provide atomic appends. We assume applications when using NOVA file system keep a record of appends and only re-execute checksum and write operations during the restart. This reduces data movement and system call costs. Next, MacroTx must re-execute the entire sequence, but offloading as CISC_{Ops} reduces cost. Finally, AutoRec uses MicroTx to automatically recover state at *F1* and *F2* and uses the operational log to re-execute and complete the CISC_{Op} without application interaction. Consequently, this provides up to 2.65X gains over ext4-DAX.

Latency Impact. To understand the performance impact of AutoRec, in Table 10d, we compare the average latency of *append-CRC-write* and *vector-write*. First, MacroTx reduces the average latency of all operations, including a substantial reduction for vectored writes by reducing I/O overheads. Next, MicroTx provides fine-grained durability (i.e., commit for each operation of a CISC_{Op}), but the latency increase over MacroTx is negligible because it reuses the same journal blocks, reducing the block allocation cost. In contrast, the optional AutoRec's operational logging with request commands and input, marginally increases the latency.

5.4 Device Compute and Memory Fairness

We next evaluate the effectiveness of FusionFS CFS-based scheduling in providing storage compute and memory fairness across tenants using workloads that are bottlenecked by (a) device-CPU and (b) device-RAM. We consider I/O intensive random *write-only* benchmark, compute-intensive *append-CRC-write*, and compute + memory-intensive *read-compress-write* workloads. We compare FusionFS CFS schedulers against round-robin I/O scheduler (*RR-scheduler*) as proposed in recent studies [44].

Device Compute + I/O Scheduling We first analyze the effectiveness of FusionFS's compute-centric CFS by co-running *append-CRC-write* with I/O-intensive *write-only* workload performing 4KB writes. In Figure 9a, the x-axis

$CISC_{Ops}$	Crash Condition
append-CRC-write	Before append completes (**F1**), after checksum calculation (**F2**), after checksum write (**F3**)
vector-write (10 writes)	before first write completes (**F4**), between writes 4 and 5 (**F5**), after all writes complete (**F6**)

(a) $CISC_{Ops}$ Failure (F) Condition.

Systems	Crash No.					
	F1	**F2**	**F3**	**F4**	**F5**	**F6**
ext4-DAX, NOVA, SplitFS, CrossFS	*C*	*C*	*C/R*	*C*	*C*	*C/R*
FusionFS-MacroTx	*C*	*C*	*C/R*	*C*	*C*	*C/R*
FusionFS-AutoRec	*C/R*	*C/R*	*C/R*	*C/R*	*C/R*	*C/R*

(b) Consistency (C) and Recovery (R) after crash. **C** and **R** denotes successful crash-consistency and recovery after failure.

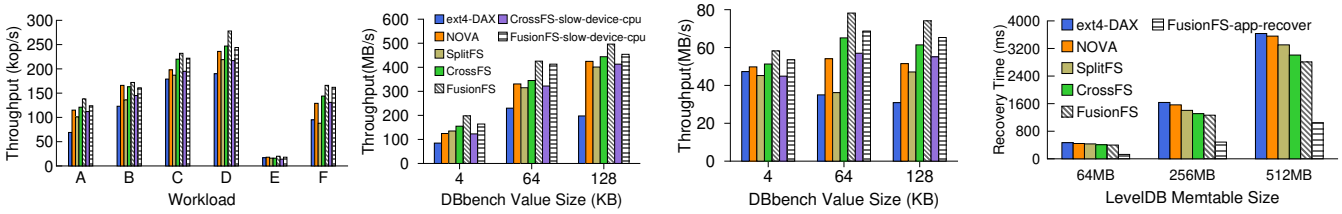
No.	ext4-DAX	NOVA	SplitFS	CrossFS	MacroTx	AutoRec
F1	75.5	55.6	23.9	43.7	30.4	28.5
F2	74.3	55.1	24.6	41.3	29.4	8.3

(c) Recovery time (ms) *append-CRC-write* running 16 threads with 16MB I/O size.

Operation	ext4-DAX	NOVA	SplitFS	CrossFS	MacroTx	AutoRec
append-CRC-write	18.4	17.3	16.5	16.9	15.6	23.4
vector-write	44.6	41.1	35.3	39.1	29.2	46.6

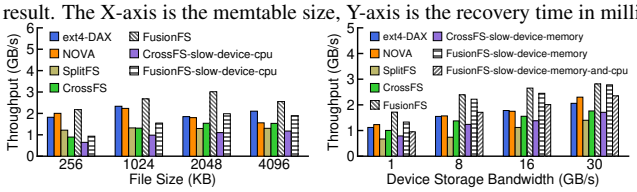
(d) Average latency (μs) for each $CISC_{Op}$.

Figure 10: Crash consistency and Fast Recovery.



(a) *YCSB* benchmark **(b)** Random Reads **(c)** Random Writes **(d)** Recovery benchmark.

Figure 11: LevelDB Evaluation. (a) shows *YCSB* benchmark result, (b) and (c) show *db_bench* benchmark results, (d) shows LevelDB Recovery benchmark result. The X-axis is the memtable size, Y-axis is the recovery time in milliseconds.



(a) Snappy Throughput **(b)** Sensitivity to Device Bandwidth

Figure 12: Snappy Compression Throughput. (a) varies file size and use 16 threads and 100K files. (b) varies storage bandwidth and use 16 threads, 100K files and 2MB each file.

varies application thread count for all workloads, the y-axis shows normalized throughput, and errors lines on the bars show max and min throughput variation across threads. The application threads operate on separate files. Further, we also compare the performance against the state-of-the-art CrossFS with round-robin scheduling in Figure 9c.

Observations. First, when using the baseline round-robin *RR-scheduler*, StorageFS picks a request from either the inode-queues of the *append-CRC-write* or the *write-only* workloads and dispatches them for processing. When using RR-scheduler, *append-CRC-write-RR* with higher compute needs could unfairly delay or starve I/O-intensive *write-only-RR* requiring short bursts of CPU for executing the file system logic, journals, and checkpointing; this impacts the throughput. In contrast, FusionFS' CFS scheduler accounts for both workloads' virtual device CPU usage time, equally prioritizes the *write-only* and the *append-CRC-write* workloads, consequently improving the throughput of *write-only-CFScpu* workload by 1.34x. The throughput of *append-CRC-write-CFScpu* reduces marginally. Finally, our CFS scheduler achieves higher gains over CrossFS for 8 and 16 thread configurations.

Memory-based Scheduling To understand the effectiveness of device memory-centric scheduling, we study capacity-intensive *read-compress-write* (shown as Compress-Write) co-scheduled with *append-CRC-write* workloads in Figure 9b. We limit the device-RAM budget for in-storage processing to 2*GB*. We compare *append-CRC-write-CFScpu* and *compress-write-CFScpu* against *append-CRC-write-CFSmm*, and *compress-write-CFSmm* scheduler that treats memory capacity as a first-class citizen towards $CISC_{Ops}$ scheduling.

Observations. The CFS CPU scheduler lacks memory capacity awareness. Consequently, the memory-intensive compression workload often stalls *append-CRC-write* despite the availability of device-CPUs. Moreover, stalling leads to side-effects such as frequent polling to check for free device memory availability. In contrast, FusionFS's CFS enables fairness based on each workload's memory usage, thereby equally prioritizing *append-CRC-write* and *read-compress-write* workloads. As shown, *append-CRC-write-CFSmm*'s throughput improves by 1.76x, whereas *compress-write-CFSmm*'s throughput only reduces by 15%.

5.5 Real-World Applications

We next study the benefits of FusionFS for real-world applications, LevelDB [7] and Snappy Compression [19]. Beyond performance, we also explore recovery and restart performance through simple changes to LevelDB's recovery and restart code.

For LevelDB, we modify the *append->checksum->write* sequence designed to avoid frequent commits (*fsync*) for SST files and WAL and replace them with *append-CRC-write* $CISC_{Ops}$. Similarly, we offload read operation using *read-checksum*. We evaluate the random write workload in Figure 11c and the random read workload in Figure 11b using the widely-used *db_bench* for 1 million key-value pairs and 16 application threads. The value size is varied from 4KB to 128KB. Note that recent LevelDB versions use 64K internal buffer for smaller appends. Further, we also evaluate YCSB [18] cloud benchmark using workloads A-F with varying read/write ratios issued with Zipfian distribution [31].

Observations. First, FusionFS provides considerable gains for both random write and read workloads. For random writes, smaller appends in *append-CRC-write* (e.g., 4K) are buffered, resulting in one system call instead of two calls for the larger values (64K). FusionFS gains stem from a combination of reduced system call, data movement, and communication costs. Offloading CRC to the device helps in the better utilization of host-CPUs for other work across application threads. We also observe that, in contrast to CrossFS, FusionFS CFS is more effective in multiplexing 4 device cores compared to CrossFS, which uses linked lists to schedule requests. For random reads, beyond offloading CRC to the device-CPUs and efficient scheduling, FusionFS reads only data without CRC bytes. In summary, FusionFS achieves gains between 1.23x-2.23x and 1.81x- 2.51x, respectively, over ext4-DAX.

Next, YCSB uses Zipfian access pattern, and therefore, the application-level caching is beneficial for all approaches. Despite caching, FusionFS provides high gains for write-intensive C, D, and F workloads. Furthermore, we believe adding more $CISC_{Ops}$ to other parts of the application (e.g., SST compaction) [14] would further increase the gains.

Restarts using Application-Customized $CISC_{Ops}$. We next study the benefits of application-customized $CISC_{Ops}$ in reducing restart cost using LevelDB. We observe that LSMs such as LevelDB [5], and others (e.g., Redis [10]), persist in-memory state to a write-ahead log (WAL) before updating the data file (e.g., SST files). LevelDB reads and checks the integrity of key-value pairs during restarts using the checksum and then sorts and writes them to disk files (SST files). All of these operations consume high I/O costs and data movement. Notably, the restart cost increases with the memory buffer (i.e., memtable size) and the WAL size.

Observations. Figure 11d shows the recovery cost for all prior systems, FusionFS (without application-customized restart), and FusionFS-app-recover with customized recovery. FusionFS offloads just the *read-checksum* achieving up to 1.17x faster restarts. In contrast, for *FusionFS-app-recover*, we reduce restart costs by enabling developers to construct and offload a custom *read-checksum-sort-write* $CISC_{Ops}$. For each key-value pair, the offloaded operation validates checksum, sorts them using a RB-tree in StorageFS, and writes them directly to the SST file. This results in up to 2.69x and 3.58x faster recovery over FusionFS and ext4-DAX, respectively.

Snappy Compression. Next, we evaluate FusionFS on the widely-used snappy file compression [19]. Figure 12a shows results for compressing 100K files using 16 threads. We vary the file sizes along the x-axis, and the y-axis shows the throughput in terms of bytes compressed. For FusionFS, we add a *open-read-compress-write* $CISC_{Ops}$.

Observations. First, we observe that NOVA performs well with its multi-core friendly structures and log-structured file system. Second, SplitFS avoids system calls but suffers from high kernel activity and pre-paging cost for staged mmap() files when opening 100K files. Third, CrossFS lacks $CISC_{Ops}$ and incurs higher overhead from creating file descriptor queues and data movement. In contrast, FusionFS avoids data copy overheads by aggregating I/O operations and offloading compression to device-CPUs resulting in 1.63x and 1.67x gains over ext4DAX and NOVA, respectively. However, for large 4MB files, the high compression cost dominates I/O costs, thereby reducing throughput for all cases.

Sensitivity to Device-CPU Speeds. We evaluate LevelDB (in Figure 11) and snappy compression (in Figure 12a) using FusionFS and CrossFS that use slower device-CPU (*slow-device-cpu*). Despite using slower CPUs, FusionFS gains significantly over other designs, specifically for I/O-intensive workloads. For example, LevelDB's *FusionFS-slow-device-cpu* provides 1.79x gains over ext4-DAX.

Sensitivity to Storage and Device Memory Bandwidth. Finally, in Figure 12b, we study the impact of device-RAM and storage bandwidth for memory and I/O-intensive snappy compression by throttling memory on a dual-socket CloudLab machine that uses DRAM for storage. In the x-axis, we vary the storage bandwidth from 1GB/s to 30GB/s, and the values 8GB and 16GB emulate the bandwidth of PCIe Gen4 and Gen5 SSDs [9]. Further, for CrossFS and FusionFS, we also study the impact of memory bandwidth (*CrossFS-slow-device-memory* and *FusionFS-slow-device-memory*) and the impact of slow CPU and memory bandwidth (*FusionFS-slow-device-memory-and-cpu*).

Observations. For all storage bandwidths, FusionFS provides considerable gains. For extremely low device-memory bandwidth (say, 1GB), FusionFS throughput is similar to KernFS and UserFS that use faster host DRAM. However, real devices are expected to have higher bandwidth (8GB and above) [11, 41], for which FusionFS shows considerable gains.

6 Conclusion

We designed and evaluated FusionFS, an in-storage firmware design that combines file system and data processing capabilities in modern CSDs. To reduce the impact of system calls, data copy, and other software and hardware overheads, we introduced $CISC_{Ops}$, which fuses multiple I/O and compute operations and offloads them to storage. Evaluation of FusionFS with several benchmarks and applications show significant performance benefits.

Acknowledgements

We thank Joo-young Hwang (our shepherd) for insightful comments to improve the quality of this paper. We also thank anonymous reviewers and the members of RSRL for their valuable feedback. We are grateful to Rutgers Panic Lab for the infrastructure support. This research was supported by funding from NSF grant CNS-1910593. This work was partially carried out on the experimental platform funded by NSF grant CNS-1730043.

References

[1] Completely Fair Scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[2] Cryptsetup. https://linux.die.net/man/8/cryptsetup.

[3] Facebook RocksDB. http://rocksdb.org/.

[4] Filesystem sandboxing with eBPF. https://lwn.net/Articles/803890/.

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

[6] Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.

[7] LevelDB Source Code. https://github.com/google/leveldb.

[8] OpenSSL. https://www.openssl.org/docs/man1.1.1/man1/openssl-genrsa.html.

[9] PCI Express. https://en.wikipedia.org/wiki/PCI_Express.

[10] Redis. http://redis.io/.

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

[12] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 69–86, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Tim Bisson, Ke Chen, Changho Choi, Vijay Balakrishnan, and Yang-suk Kee. Crail-kv: A high-performance distributed key-value store leveraging native kv-ssds over nvme-of. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2018.

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

[15] Ming Chen, Dean Hildebrand, Henry Nelson, Jasmit Saluja, Ashok Subramony, and Erez Zadok. vNFS: Maximizing NFS performance with compounds and vectorized I/O. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–314, Santa Clara, CA, February-March 2017. USENIX Association.

[16] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *J. Syst. Archit.*, 55(5-6):332–343, May 2009.

[17] Cloudlab. https://docs.cloudlab.us/cloudlab-manual.html.

[18] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[19] Jeff Dean, Sanjay Ghemawat, and Steinar H. Gunderson. Snappy Compession. https://github.com/google/snappy.

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

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

[22] 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.

[23] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, Boston, MA, 2011.

[24] HewlettPackard Quartz. https://github.com/HewlettPackard/quartz.

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

[26] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. YourSQL: A High-Performance Database System Leveraging in-Storage Computing. *Proc. VLDB Endow.*, 9(12):924–935, August 2016.

[27] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. SOSP '19: Symposium on Operating Systems Principles, New York, NY, USA, 2019. ACM.

[28] 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.

[29] 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 '19, page 144–154, New York, NY, USA, 2019. Association for Computing Machinery.

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

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

[32] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 521–534, New York, NY, USA, 2017. ACM.

[33] Jesung Kim, Jong Min Kim, S. H. Noh, Sang Lyul Min, and Yookun Cho. A Space-Efficient Flash Translation Layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

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

[35] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[36] Chyuan Shiun Lin, Diane CP Smith, and John Miles Smith. The Design of a Rotating Associative Memory for Relational Database Applications. *ACM Transactions on Database Systems (TODS)*, 1(1):53–65, 1976.

[37] Jing Liu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Sudarsun Kannan. File Systems as Processes. In *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

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

[39] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009.

[40] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

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

[42] Simon Peter, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system as control plane. In *Proc. 11th USENIX Conf. Oper. Syst. Des. Implement*, volume 38, pages 44–47, 2013.

[43] Moinuddin K. Qureshi and Gabe H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 235–246, Washington, DC, USA, 2012. IEEE Computer Society.

[44] Yujie Ren, Changwoo Min, and Sudarsun Kannan. Crossfs: A cross-layered direct-access file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154. USENIX Association, November 2020.

[45] Yujie Ren, Jian Zhang, and Sudarsun Kannan. CompoundFS: Compounding I/O Operations in Firmware File Systems. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020.

[46] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[47] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drivei. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, Renton, WA, July 2019. USENIX Association.

[48] Samsung. NVMe SSD 960 Polaris Controller. http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/NVMe_SSD_960_PRO_EVO_Brochure.pdf.

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

[50] Samsung. Samsung NVMe SSD 960 Data Sheet. http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_960_PRO_Data_Sheet_Rev_1_1.pdf.

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

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

[53] StorageReview.com. Firmware Upgrade. http://www.storagereview.com/how_upgrade_ssd_firmware.

[54] Stanley Y. W. Su and G. Jack Lipovski. CASSM: A Cellular System for Very Large Data Bases. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, Framingham, Massachusetts, 1975.

[55] Wei Su, Akshay Auror, Ming Chen, and Erez Zadok. Supporting transactions for bulk NFSv4 compounds. In *Proceedings of the 13th ACM International Systems and Storage Conference (SYSTOR '20)*, Haifa, Israel, June 2020. ACM.

[56] Tarasov Vasily. Filebench. https://github.com/filebench/filebench.

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

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

[59] NVM Express Workgroup. NVMExpress Specification. https://nvmexpress.org/resources/specifications/.

[60] 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, pages 427–439, New York, NY, USA, 2019. ACM.

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

[62] 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 *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 225–237, Santa Clara, CA, February 2020. USENIX Association.

InfiniFS: An Efficient Metadata Service for Large-Scale Distributed Filesystems

Wenhao Lv[†] Youyou Lu[†] Yiming Zhang[‡] Peile Duan[↟] Jiwu Shu[*†‡]

[†]*Department of Computer Science and Technology, BNRist, Tsinghua University*
[‡]*School of Informatics, Xiamen University* [↟]*Alibaba Group*

Abstract

Modern datacenters prefer one single filesystem instance that spans the entire datacenter and supports billions of files. The maintenance of filesystem metadata in such scenarios faces unique challenges, including load balancing while preserving locality, long path resolution, and near-root hotspots.

To solve these challenges, we propose INFINIFS, an efficient metadata service for extremely large-scale distributed filesystems. It includes three key techniques. First, INFINIFS decouples the *access* and *content* metadata of directories, so that the directory tree can be partitioned with both metadata locality and load balancing. Second, INFINIFS designs the *speculative path resolution* to traverse the path in parallel, which substantially reduces the latency of metadata operations. Third, INFINIFS introduces the *optimistic access metadata cache* on the client-side, to alleviate the near-root hotspot problem, which effectively improves the throughput of metadata operations. The extensive evaluation shows that INFINIFS outperforms state-of-the-art distributed filesystem metadata services in both latency and throughput, and provides stable performance for extremely large-scale directory trees with up to 100 billion files.

1 Introduction

Modern datacenters for fast-expanding businesses often contain huge numbers of files, which can easily exceed the capacity of one single instance of current distributed filesystems [23, 27, 35, 36, 39, 42]. Currently, a datacenter is often divided into relatively smaller clusters, each of which runs a distributed filesystem instance separately. However, it is more desirable to manage the entire datacenter with one single filesystem instance, which provides global data sharing, high resource utilization, and low operational complexity. For example, Facebook introduced the Tectonic distributed filesystem to consolidate small storage clusters into one single instance that contains billions of files [28].

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

Scalable and efficient metadata service is crucial for distributed filesystems [14, 22, 23, 25, 28, 31–33, 36, 41]. As modern datacenters often contain tens or even hundreds of billions of files, using one extremely large-scale filesystem to manage all the files brings severe challenges to the metadata service. First, directory tree partitioning is challenging to achieve both high metadata locality and good load balancing, as the directory tree expands and the workloads are diverse. Second, the latency of path resolution could be high, as the file depths are deep in extremely large-scale filesystems. Third, the overhead of coherence maintenance for client-side metadata cache becomes overwhelming, as extremely large-scale filesystems usually need to serve a large number of concurrent clients.

This paper presents INFINIFS, an efficient metadata service for extremely large-scale distributed filesystems. In order to address the challenges mentioned above, INFINIFS distributes the filesystem directory tree and accelerates metadata operations with the following designs.

First, we propose an *access-content decoupled partitioning* method to achieve both high metadata locality and good load balancing. The key idea is to decouple the access metadata (name, ID, and permissions) and content metadata (entry list and timestamps) of the directory, and further partition these metadata objects at a fine-grained level. Specifically, we first group each directory's access metadata with its parent, and content metadata with its children, thereby achieving high metadata locality. Then, we partition these fine-grained groups to different metadata servers with consistent hashing on directory IDs, thereby ensuring good load balancing.

Second, we design a *speculative path resolution* to traverse the directory tree in parallel, which substantially reduces the latency of metadata operations. The key idea is to assign a predictable ID to each directory, so that clients can speculate on the IDs of all intermediate directories, then send lookups for multi-component paths in parallel.

Third, we introduce an *optimistic access metadata cache* to alleviate the near-root hotspots, which achieves scalable path resolution. The key idea is to cache directory access metadata on the client-side to absorb the frequent lookups on near-

root directories, and to invalidate cache entries lazily on the metadata servers with low overhead. Specifically, the directory `rename` and directory `set_permission` operations will send the cache invalidation notification to metadata servers instead of numerous clients, so that each server can validate the cache staleness lazily when processing client metadata requests.

In summary, this paper makes the following contributions:

- We identify the challenges that impair the performance of metadata services in the large-scale scenario (§2).
- We propose a scalable and efficient distributed metadata service, INFINIFS, featured with access-content decoupled partitioning (§3.2), speculative pathname resolving (§3.3), and optimistic access metadata cache (§3.4).
- We implement and evaluate INFINIFS to demonstrate that INFINIFS outperforms the state-of-the-art distributed filesystems in both latency and throughput of metadata operations, and provides stable performance for extremely large-scale directory trees with up to 100 billion files (§5).

2 Background and Motivation

In this section, we first explain why having one single filesystem instance that spans the entire datacenter is desirable (§ 2.1). Then, we discuss the unique challenges for efficient metadata services in such scenarios (beyond billions of files) (§ 2.2). Finally, we analyze the characteristics of metadata access in real datacenter workloads (§ 2.3).

2.1 Large-Scale Filesystem

The filesystem typically provides users with a hierarchical namespace (i.e., a directory tree) to manage files. In the directory tree, each file/directory possesses metadata information. Metadata operations typically involve two critical steps, i.e., path resolution and metadata processing. When a user accesses a file with a pathname, e.g., `/home/Alice/paper.tex`, metadata is accessed as follows: First, the filesystem executes path resolution to locate the target file and check whether the user has the proper permissions; then, the filesystem executes metadata processing to update corresponding metadata objects atomically.

The metadata service is the scalability bottleneck for large-scale distributed filesystems [22, 32, 36]. The distributed filesystem is an important infrastructure component of datacenters [2, 12, 35, 39]. As the number of files inside a datacenter grows rapidly, the metadata service becomes the scalability bottleneck of the distributed filesystems. Currently, datacenters usually consist of a constellation of filesystem clusters. For example, the Alibaba Cloud maintains nearly thousands of Pangu distributed filesystems to collectively support up to tens of billions of files in the datacenter [2]. Facebook also needs many HDFS clusters to store datasets in one single datacenter [28], because each HDFS cluster supports at most 100 million files due to the metadata limitation [36].

However, a large-scale filesystem spanning the entire datacenter is more desirable. For example, Facebook introduced the Tectonic distributed filesystem to consolidate small storage clusters into one single instance [28]. One single large-scale filesystem per datacenter outperforms a constellation of small filesystem clusters in the following aspects:

- *Global data sharing.* One single large-scale filesystem provides a global namespace, enabling better data sharing across the datacenter. In contrast, storing different datasets in separate clusters is inefficient, requiring dedicated data placement and causing data movement. It also complicates the logic of computing service, because related data might be split among separate filesystems.
- *High resource utilization.* Global data sharing can eliminate duplicated data among separate clusters, thus improving disk capacity utilization. Further, one single filesystem allows better resource sharing. In contrast, in the constellation approach, the idle resources in one filesystem cluster could not be reallocated to other clusters.
- *Low operational complexity.* One single large-scale filesystem can significantly reduce the operational complexity, as there is only one system to maintain. In contrast, maintaining thousands of filesystem clusters in a large datacenter (such as Alibaba Cloud) is labor-intensive and error-prone.

2.2 Challenges of Scalable Metadata

One single large-scale filesystem spanning the entire datacenter needs to support billions of files and serve a large number of clients. This brings severe challenges for the metadata service of distributed filesystems, as discussed below.

Challenge 1: *Directory tree partitioning is challenging to achieve both high metadata locality and good load balancing, as the directory tree expands and workloads are diverse.*

Metadata locality is important for efficient metadata processing. A filesystem operation often processes multiple metadata objects. For example, a file creation first locks the parent directory to serialize with directory listing operations [25], then updates three metadata objects atomically, including the file metadata, the entry list, and the directory timestamps. With metadata locality, we can avoid distributed locks and distributed transactions, thus achieving low-latency and high-throughput metadata operations.

Load balancing is important to achieve high scalability. Metadata operations often cause load imbalance in the directory tree. This is particularly true for real-world datacenter workloads where related files are grouped into subtrees [4, 41]. Files and directories from the continuous subtree may be heavily accessed in a short period, causing a performance bottleneck on the metadata server that stores the subtree.

Existing partitioning strategies fail to achieve both high locality and good load balancing in the extremely large-scale scenarios. Managing all files in the datacenter causes the directory tree to expand rapidly in both depth and breadth. Further,

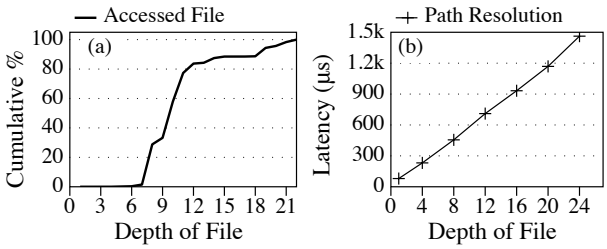


Figure 1: (a) The depth distribution of accessed files in a real-world distributed filesystem workload. (b) The latency of path resolution increases rapidly as the depth of files grows.

File Op	95.8%	Directory Op	4.2%
open/close	54.9%	readdir	93.3%
stat	12.9%	statdir	6.6%
create	10.0%	mkdir	0.1%
delete	12.4%	rmdir	0.1%
rename	9.7%	rename	0.0%
set_permission	0.1%	set_permission	0.0%

Table 1: Ratios of different operations in the real-world workloads from deployed systems. We show the relative ratios of file operations and directory operations separately.

the filesystem faces various workloads with different characteristics, since it backs all datacenter services. This is challenging for the existing directory tree partitioning strategies. Fine-grained partitioning, such as directly hashing metadata objects to servers [25, 37], can achieve load balancing. However, it sacrifices locality and frequently causes distributed locking, introducing expensive coordination overhead and leading to high latency and low throughput [8, 20]. Coarse-grained partitioning, such as grouping a continuous subtree onto the same server [4, 41], preserves locality and avoids cross-server operations. However, it is susceptible to skewed workloads, which lead to the load imbalance.

Challenge 2: *The latency of path resolution could be high, as the file depths are deep in extremely large-scale filesystems.*

The file depth becomes increasingly deeper in extremely large-scale filesystems. Previous filesystems usually assume that the depth of most files in the directory tree is less than 10 [7, 10]. However, we find that the depth of files rapidly increases when consolidating all services into one single filesystem. Figure 1(a) presents the depth distribution of accessed files in a real-world workload (§2.3). We can observe that almost half of the accessed files have a depth of more than 10.

The deep directory hierarchy has a high impact on the filesystem performance. We implement a naive path resolution mechanism based on the design of Tectonic [28], and evaluate the latency of path resolution as the depth grows. Figure 1(b) shows that the latency of path resolution increases linearly with the depth of files. Tectonic partitions directories to different metadata servers based on the directory ID, and consequently, resolving a path at a depth of N requires resolving the $N-1$ intermediate directories, which leads to $N-1$ sequential network requests.

Challenge 3: *The overhead of coherence maintenance for client-side metadata cache becomes overwhelming, as extremely large-scale filesystems usually need to serve a large number of concurrent clients.*

The path resolution needs to traverse the directory tree from the root and check the permissions of all intermediate directories inside the path sequentially. This causes the near-root directories to be read heavily, even for a balanced metadata operation workload. The filesystem throughput will then be bounded by the server that stores the near-root directories. We call this near-root hotspot in this paper. Many distributed filesystems depend on the client-side metadata cache to mitigate the near-root hotspot [13, 23, 30, 31].

We observe that previous client-side cache mechanisms do not work well in large-scale scenarios with numerous clients. For example, the lease-based mechanism grants a lease to every cache entry that will expire after a fixed duration. When the lease expires, corresponding cache entries become invalidated automatically. The lease mechanism is widely used by the NFS v4 [30], PVFS [13], LocoFS [23], and IndexFS [31]. However, the lease mechanism suffers from load imbalance caused by cache renewals at the near-root directories. This is because all clients have to repeatedly renew their cache entries of the near-root directories for path resolution. As the number of clients increases, such load imbalance at the near-root directories will eventually become a performance bottleneck and impair the overall throughput.

2.3 Characteristics of Real-World Workloads

To understand the characteristics of an extremely large-scale distributed filesystem, we analyze the relative frequency of metadata operations in a real-world workload. We trace metadata operations in production deployments from the Alibaba Cloud, one of the largest cloud providers. We capture workloads from three Pangu filesystem instances that support different services: data processing and analyzing service, object storage service, and block storage service. We merge the workloads from these different services to represent the workload of a large-scale filesystem that spans the entire datacenter. The relative frequencies of metadata operations are shown in Table 1, from which we can observe that:

- File operations account for ~ 95.8% of all operations.
- The directory `readdir` is the most frequent directory operation, accounting for ~ 93.3% of all directory operations.
- Directory `rename` and directory `set_permission` operations rarely occur, accounting for only ~ 0.0083% of all metadata operations.

These insights also refine our design of InfiniFS.

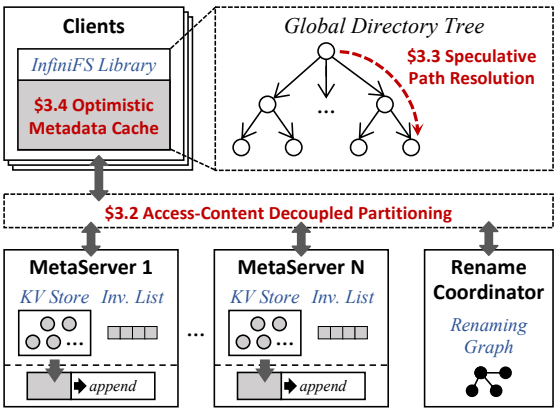


Figure 2: Architecture of INFINIFS.

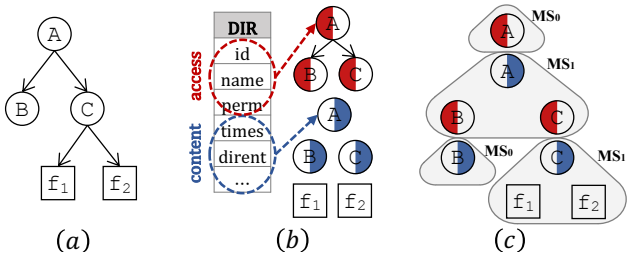


Figure 3: (a) Filesystem directory tree consists of the directory metadata (○) and the file metadata (□). (b) Directory metadata is decoupled into two parts: the access metadata (◐) which contains the name, ID, and permissions, and the content metadata (◑) which contains the timestamps, entry list (i.e., dirent), etc. (c) Directory access metadata is grouped with the parent. Directory content metadata is grouped with the children. These per-directory groups are partitioned to metadata servers (MS_n) by hashing directory IDs.

3 Design and Implementation

We design INFINIFS with three key ideas as below:

- **Decoupling directory metadata.** INFINIFS divides directory metadata into the access state (*name, ID, and permissions*) of the directory itself, and the content state (*entry list and timestamps*) related to the children, so it can be partitioned on a fine grain for load balancing, while still retaining good locality for the metadata processing of common operations such as `create`, `delete`, and `readdir`.
- **Speculating directory IDs in path resolution.** INFINIFS uses a predictable ID for each directory based on the cryptographic hash on the parent ID, the name, and a version number. It enables clients to speculate on directory IDs and launch lookups for multi-component paths in parallel.
- **Invalidating client-side cache lazily.** INFINIFS caches directory access metadata on the client-side to avoid hotspots near the root, thus achieving scalable path resolution. The client uses cache entries for path resolution, agnostic about their staleness. The metadata server lazily validates cache staleness when processing the client metadata requests.

3.1 Overview

INFINIFS is an efficient metadata service for extremely large-scale distributed filesystems. Figure 2 presents the architecture of INFINIFS, which contains the following components:

- *Clients.* INFINIFS provides a global filesystem directory tree that is shared by clients. Clients contact INFINIFS through the user-space library or the FUSE user-level filesystem. They traverse the directory tree via *speculative path resolution* (§3.3), which minimizes latency by predicting directory IDs and parallelizing lookups. Clients use the *optimistic metadata cache* (§3.4) during path resolution to mitigate the excessive read load on near-root directories.
- *Metadata Servers.* The filesystem directory tree is distributed across metadata servers via the *access-content decoupled partitioning* (§3.2), which achieves both high metadata locality and good load balancing. Each server manages metadata objects in the local *key-value store* (i.e., *KV Store*), which typically caches metadata in memory and logs updates in NVMe SSDs for high performance. Metadata servers use the *invalidation list* (i.e., *Inv. List*) to validate client metadata requests lazily.
- *Rename Coordinator.* A central *rename coordinator* is used to process the directory `rename` and directory `set_permission` operations. It checks concurrent directory renames with the *renaming graph* to prevent orphaned loops (§4.1), and broadcasts modification information to invalidation lists of metadata servers.

3.2 Access-Content Decoupled Partitioning

In this section, we first explain why to decouple access and content metadata. Then, we show how to achieve metadata locality via grouping and load balancing via partitioning. Finally, we show how to store metadata in key-value pairs.

Decoupling directory metadata. As discussed in §2.2, previous fine-grained and coarse-grained partitioning failed to achieve both high metadata locality and good load balancing at the same time. Essentially, the root cause is that they treat the directory metadata as a whole. Therefore, when partitioning the directory tree, they have to split the directory either from its parent or its children to different servers, which unintentionally breaks the locality of related metadata.

We analyze the composition of directory metadata and find that directory metadata consists of two independent parts: access and content. As shown in Figure 3(b), *access metadata* contains the directory name, ID, and permissions, which are used to access the directory tree. *Content metadata* contains the entry list, timestamps, etc, which are related to the children. Therefore, we propose to decouple directory metadata into

Metadata Objects	Key	Value	Partitioned by
Directory Access Metadata	*pid, name*	*id, permission*	*pid*
Directory Content Metadata	*id*	*entry list, timestamps, etc.*	*id*
File Metadata	*pid, name*	*file metadata*	*pid*

Table 2: INFINIFS stores metadata objects as key-value pairs. *pid: ID of the parent directory. id: ID of the directory.*

access and content, so as to group and partition two parts independently for both metadata locality and load balancing.

Grouping for locality. We group related metadata objects to the same metadata server to achieve high locality for the metadata processing phase. We first analyze the metadata requirements of all kinds of metadata operations, to determine the related metadata objects during metadata processing. We classify metadata operations into three categories as below:

1. Operations that only process the metadata of the target file/directory, such as `open`, `close`, and `stat`. For example, a file stat will only read the metadata of the target file.
2. Operations that process the metadata of the target file/directory and its parent, such as `create`, `delete`, and `readdir`. For example, a file creation will first insert the file metadata, then lock and update the entry list and timestamps of the parent directory.
3. `Rename` operation is special, as it processes the metadata of two files/directories and their parents.

We observe that most metadata operations (category 1 and 2) require metadata within the target file/directory and the parent during metadata processing. With decoupled directory metadata, we group each directory's content metadata with its subdirectories' access metadata and its files' metadata, as shown in Figure 3(c). In this way, we split the directory tree into independent per-directory groups for later partitioning, while retaining metadata locality for directory `readdir` and file `create/delete/open/close/stat/set_permission` operations. Based on the relative frequency of metadata operations (Table 1), these operations account for ~ 90% of all operations. Thus, INFINIFS achieves high locality for most metadata operations.

Partitioning for load balancing. We further partition the directory tree at a fine-grained level for good load balancing. Based on the locality-aware metadata grouping, we split the directory tree into independent per-directory groups, and then partition these groups to different metadata servers by hashing the directory ID. Such fine-grained hash partitioning effectively load-balances metadata operations [28].

We illustrate the partitioning in Figure 3(c). The metadata group, which contains content metadata of `C` and metadata of `f1` and `f2`, is partitioned to the metadata server 1. In this way, a file `create` under `C` only needs a local transaction in metadata server 1 to insert the new file metadata, then update the entry list and timestamps of `C`. And a `readdir` on `C` only involves metadata server 1 to first lock the directory entry list for isolation, then read the file names from the entry list.

INFINIFS also leverages consistent hashing [29] to map these fine-grained metadata groups to servers, so as to minimize the migration during cluster expanding or shrinking.

Storing. We implement access-content decoupled partitioning with the KV store as the backend storage. The key-value indexing schema is detailed in Table 2, which consists of three kinds of key-value pairs, two for directory access and content metadata, and one for file metadata. To resolve `/A/B/file` (let the IDs of `/`, `A`, and `B` be 0, 1, and 2), we first use $\langle 0, A \rangle$ as the key to get `A`'s access metadata, and find that `A`'s ID equals 1. Then, we use $\langle 1, B \rangle$ to get the `B`'s access metadata, and find that `B`'s ID equals 2. Finally, we use $\langle 2 \rangle$ to get the `B`'s content metadata, and $\langle 2, file \rangle$ to get the file's metadata.

3.3 Speculative Path Resolution

In this section, we first introduce how to generate predictable directory identifiers (§ 3.3.1). Then, we describe how to parallelize path resolution with speculation (§ 3.3.2).

3.3.1 Predictable Directory ID

Here, we present how INFINIFS generates and maintains directory IDs that can be predicted from pathnames later.

1) *Creating.* When creating a new directory, we generate the directory's ID by hashing its *birth triple*: ⟨❶ parent ID, ❷ directory name, ❸ name version⟩, as shown in Figure 4(a). The parent where a directory is created is referred to as the directory's *birth parent*. We use the version to guarantee the universal uniqueness of the birth triple.

2) *Renaming.* When renaming a directory to another location, only the key of its access metadata needs to be updated; its content metadata and ID remain unmodified, thus all descendants' metadata under the directory remain intact.

When a directory is renamed for the first time since creation, its birth parent will record a *rename-list* (RL): ⟨❶ directory name, ❷ name version⟩, and the directory itself will record a *back-pointer* (BP): ⟨❶ birth parent's ID, ❷ name version⟩. The RL of a directory records the subdirectories that were born in that directory but have been moved elsewhere. The BP of a renamed directory (i.e., a directory that has been moved elsewhere from its birth parent) points to its birth

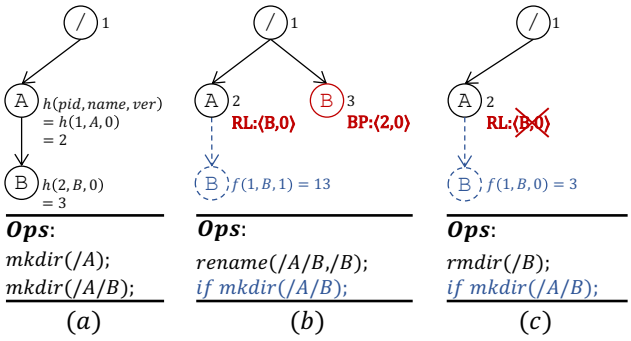


Figure 4: (a) Creating a directory generates its ID by hashing the parent directory ID, the directory name, and the name version. (b) Renaming a directory creates a back-pointer (BP) in itself and an entry in the rename-list (RL) of its birth parent. BP and RL are used for the uniqueness of the directory ID. (c) Deleting the renamed directory erases RL and BP.

parent. We determine the name version with the parent's RL in directory creation. For example, Figure 4(b) shows the `rename(/A/B,/B)` when `B` is renamed for the first time. The key to `B`'s access metadata is updated from `2:B` to `1:B`, while the ID of `B` remains unmodified. `A` records the RL $\langle B,0\rangle$ on the same server as its content metadata. `B` records the BP $\langle 2,0\rangle$ in its access metadata. At this moment, if creating a new `B` under `/A`, its name version should be 1, as the RL of `/A` indicates that there exists a renamed `B` that was born here.

When a directory is renamed again, only the key of its access metadata needs to be updated, while its content metadata, BP, and birth parent's RL remain unmodified. When deleting the birth parent of a renamed directory, the birth parent's access and content metadata are erased, but the birth parent's RL is retained. The RL is only removed through the BP when the renamed directories are deleted.

3) *Deleting.* When deleting a renamed directory, we use the BP of that directory to erase the RL of its birth parent, as shown in Figure 4(c). At this moment, if creating a new `B` under `/A`, its name version returns to the default zero.

ID uniqueness. Here, we first show the predictable directory ID is universally unique, then show hash collisions are very rare, and INFINIFS can detect and handle them properly. A directory ID is generated by hashing the birth triple, so if each birth triple is universally unique, the ID should be universally unique as well, unless a hash collision happens. The filesystem semantic mandates that no two directories inside the same parent have the same name at any time. Without directory renames, ⟨birth parent's ID, directory name⟩ is sufficient to be universally unique. When a directory is renamed elsewhere, a new directory with the same name is allowed to be created within the same parent. We use a version number, as elaborated before, to solve the directory rename problem, thus guaranteeing that each birth triple is universally unique.

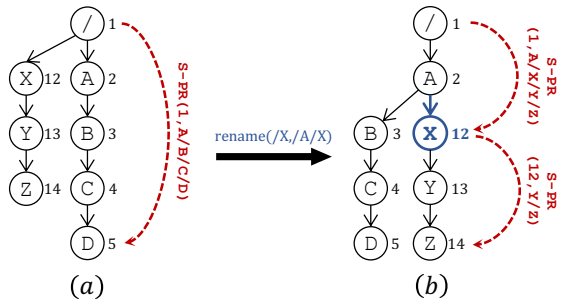


Figure 5: (a) Speculative path resolution (S-PR) reduces the latency of path resolution to nearly one network round-trip time, if correctly predicted. (b) After renaming the directory `/X` to be a child of `A`, resolving `/A/X/Y/Z` requires two rounds, taking about two network round-trip times.

We generate IDs using a cryptographic hash (e.g., SHA-256), so the chance of a collision is very small. As directory ID is the key of each directory's content metadata, we can easily detect a hash collision when inserting the new content metadata during directory creation. We handle a hash collision in the same way as a renaming by using the version number, RL, and BP. In INFINIFS, the collision and rename cases have the same effect on subsequent directory creations and can be distinguished through the RL entry format.

3.3.2 Parallel Path Resolution

Based on the predictable directory ID, a client can conduct path resolution in parallel with the following two steps:

1) *Predict directory IDs.* The client predicts the IDs of all intermediate directories by using the root ID of the path or subpath. It first reconstructs the birth triples with 0 as the version number, then recalculates the hashing results. With the speculated directory IDs, the client reconstructs the keys for all path components.
2) *Lookup in parallel.* The client sends lookup requests for all intermediate directories in parallel. Each lookup request will check the access permissions and compare the speculated ID with the ID stored in the metadata server. If the speculated ID does not match the one on the server, the lookup request returns the true ID to the client.
3) Steps 1) and 2) are repeated until the resolving completes.

Figure 5 shows the speculative path resolution mechanism. If one of the intermediate directories was once renamed to here, the speculated ID of the renamed directory will be wrong (i.e., $h(2,X,0) \neq 12$). However, the lookup request can find `X`'s access metadata using the correct key `2:X`, and returns the directory's true ID to the client. With the true ID of `X`, the client then continues to resolve the subpath under `X`.

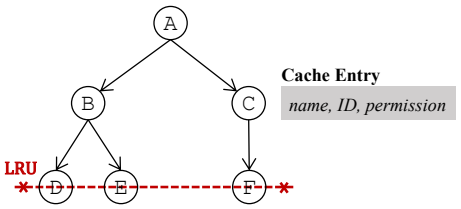


Figure 6: Organization of the access metadata cache on the client-side. Cache entries are organized as a tree. The leaf entries are linked as a least-recently-used (LRU) list.

3.4 Optimistic Access Metadata Cache

In this section, we first show how INFINIFS organizes directory access metadata at the client-side cache. Then, we introduce how clients use cache entries optimistically, and how metadata servers invalidate stale cache entries lazily.

Cache organization. INFINIFS caches only directory access metadata (i.e., directory name, ID, and permissions) on the client side. Cache hits will eliminate lookup requests to near-root directories, thereby avoiding hotspots near the root and ensuring scalable path resolution. As illustrated in Figure 6, INFINIFS organizes cache entries in a tree structure based on the filesystem hierarchy, and links the leaf entries as a least-recently-used (LRU) list. When cache replacement happens, the least recently used leaf entry will be evicted, ensuring that the near-root directories remain cached.

Lazy invalidation. A lot of cache entries become stale after the directory `rename` or directory `set_permission` operation. It is impractical to invalidate stale cache entries on all associated clients during each directory `rename` operation. Because the membership of clients is difficult to manage, and the number of clients can be huge, substantially outnumbering the number of metadata servers.

The lazy invalidation addresses this problem by broadcasting the invalidation information to metadata servers (the number of which is significantly less than clients), so that each server can validate cache staleness lazily when processing client requests. Specifically, a directory `rename` will contact the central rename coordinator to prevent orphaned loops (§4.1), then broadcast the rename information to metadata servers. A single coordination server should be sufficient to handle these infrequent operations, because directory `rename` operations rarely occur (accounting only for ~ 0.0083%) based on the real-world workload studies in §2.3.

As illustrated in Figure 7(a), INFINIFS handles the directory `rename` with the following procedures:

❶ INFINIFS sends the directory rename operation to the rename coordinator, to detect whether this directory rename produces orphaned loops with the in-flight ones. Then, the coordinator assigns each directory rename operation an incremental version number.

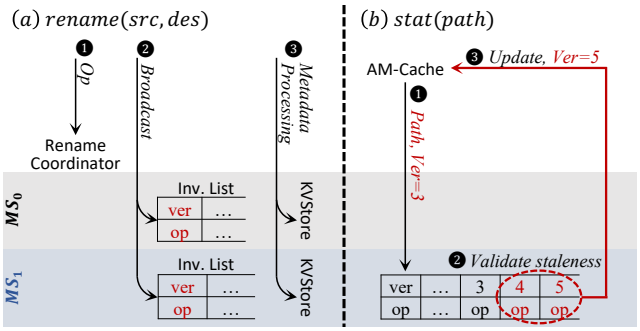


Figure 7: (a) Directory `rename` broadcasts the modification information to all metadata servers (MS_n). (b) Metadata server lazily validates cache staleness when clients issue metadata requests. *ver: version. op: rename operation.*

❷ INFINIFS serializes directory renames with other operations by locking the target directory, so that new accesses to the subtree are blocked until the rename completes. Then, it broadcasts the rename information with its version to metadata servers in parallel, and waits for acknowledgments. The metadata server maintains the rename information in the invalidation list sorted by the version.

❸ INFINIFS moves the directory's access metadata from the source server to the destination server, and updates the RL and BP if the directory is renamed for the first time.

As illustrated in Figure 7(b), INFINIFS validates cache staleness lazily at the server in the following manner:

❶ INFINIFS clients are agnostic about the staleness and optimistically utilize local cache entries during path resolution. Each client has a local version, indicating that its cache has been updated with rename operations before this version. When a client contacts a metadata server, it sends the request along with the pathname and version.

❷ The metadata server validates staleness by comparing the pathname against rename operations in the invalidation list. Only operations between the request's version and the latest version in the invalidation list need to be compared. If the server finds the requested pathname is valid, the request is processed and returned successfully.

❸ If the server finds the request is invalid, it aborts the request and returns the information of these new rename operations. The client then updates the cache and version.

4 Consistency

In this section, we introduce how INFINIFS prevents orphaned loops caused by directory `rename` operations (§4.1), and implements transactional metadata operations (§4.2), so as to guarantee the consistency of the filesystem directory tree.

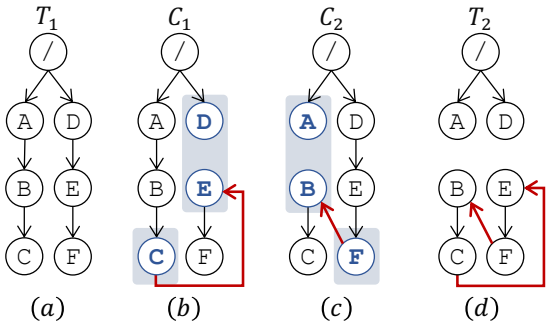


Figure 8: The orphaned loop caused by concurrent directory renames. At time T_1, $Client_1$ tries to rename E to be a child of C, and $Client_2$ tries to rename B to be a child of F, resulting in an orphaned loop at time T_2. The required metadata objects of each rename operation is colored in blue.

4.1 Orphaned Loop

Concurrent directory renames may cause orphaned loops, which lead to server data loss. Figure 8 illustrates such an orphaned loop caused by two directory renames. As shown in Figure 8(a), the filesystem directory tree should be connected and acyclic. In Figure 8(b) and 8(c), $Client_1$ attempts to rename `E` to be the child of `C`, while $Client_2$ attempts to rename `B` to be the child of `F`. The metadata objects required for two renames are completely independent of each other, therefore allowed to execute in parallel. But they lead to an orphaned loop that breaks the directory tree, as shown in Figure 8(d). No client can access any files within the orphaned loop.

InfiniFS addresses the orphaned loop problem by using a central *rename coordinator* to check each directory `rename` operation before its execution. The rename coordinator maintains a *renaming graph*, which tracks the source and destination paths of in-flight directory renames at the present moment. Before allowing a new directory rename operation to proceed, the rename coordinator first verifies whether it leads to an orphaned loop with the in-flight ones. The source and destination paths of a directory rename operation are kept in the renaming graph throughout the renaming procedure, and deleted after the rename operation completes.

Directory `rename` operations rarely occur (accounting for less than ~ 0.0083%) based on the real-world workload studies in §2.3. Therefore, a single rename coordinator should be sufficient to handle directory rename operations.

4.2 Transactional Metadata Operations

Operations in InfiniFS can be classified into three types:

1) *Single-server operations.* These operations include directory `readdir` and file `create/delete/open/close/stat/set_permission`, which are the most frequent operations. They process metadata objects within one single metadata server. Single-server operations leverage the transaction mechanism of the key-value storage backend to guarantee atomicity. When a metadata server restarts after a crash, it recovers the transactions of metadata operations and consults the rename coordinator to update its invalidation list.

2) *Two-server operations.* These operations include directory `mkdir/rmdir/statdir` and file `rename`. They process metadata objects across two metadata servers, requiring distributed transactions to guarantee correct behavior. InfiniFS adopts the two-phase commit protocol [21, 24, 31] for the atomicity of updates across servers. Since clients are unreliable and difficult to track, one of the two metadata servers is chosen to be the coordinator in the transaction. To recover from failures, write-ahead logging is used by both the coordinator and the participant to record the partial state of the transaction.

3) *Directory rename operations.* Directory `rename` and directory `set_permission` operations rarely occur. These operations are delegated to the rename coordinator, which detects orphaned loops, broadcasts the modification to all metadata servers, and processes the target directory metadata across two servers. These operations are implemented with distributed transactions similar to the two-server operations, with the difference that they choose the rename coordinator as the coordinator in the transaction, and broadcast modification at the beginning of the commit phase. If the rename coordinator crashes during the broadcast, it restarts and recovers the transaction by restarting the broadcast to ensure the modification is delivered to all servers at least once.

5 Evaluation

In this section, we use a number of microbenchmarks to evaluate InfiniFS, seeking to answer the following questions:

- How does InfiniFS compare to other distributed filesystems in the metadata performance? (§5.2)
- How do the design features employed in InfiniFS contribute to the overall performance? (§5.3)
- How does InfiniFS perform for extremely large-scale directory trees (up to 100 billion files)? (§5.4)
- How does InfiniFS compare to the lease mechanism in cache efficiency? (§5.5)
- What is the performance of directory `rename` and file `rename` in InfiniFS? (§5.6)
- What is the overhead of speculative path resolution in case of mispredictions? (§5.7)

5.1 Experimental Setup

5.1.1 Hardware Configuration

Experiments are conducted on-premises using 32 client nodes and 32 server nodes with the same hardware configurations,

CPU	Intel Xeon Platinum 2.50GHz, 96 cores
Memory	Micron DDR4 2666MHz 32GB × 16
Storage	RAMdisk
Network	ConnectX-4 Lx Dual-port 25Gbps

Table 3: Hardware configurations.

as shown in Table 3. Each node has two physical ports bonded to a single IP address. All nodes are connected in a network topology with two-level switches. The client nodes can run up to 2048 client processes in parallel, sufficient to saturate the metadata services of tested filesystems.

In our experiments, distributed filesystems are deployed on RAMdisks. Thus, evaluations show the pure performance of different designs, independent of the disk I/O speed. Actually, in production deployments (e.g., Pangu and HDFS), filesystem metadata is typically placed in DRAM and uses a standalone logger to persist metadata updates in NVMe SSDs (with tens of microseconds latency), and thus the latency of metadata operations is mainly affected by the network RPCs (with hundreds of microseconds latency) rather than the local storage devices (NVMe SSDs or RAMdisks).

5.1.2 Software Configuration

Each node runs CentOS 7 with Linux kernel version 4.9.151.

Compared Systems. We choose four state-of-the-art distributed filesystems for comparison, namely, LocoFS [23], IndexFS [31], CephFS [39], and HopsFS [25]. We use CephFS at version 12.2.13, deployed with multiple active MDS daemons, coexisting with OSD daemons on 32 server nodes. We use HopsFS at version 3.2.0.0, deployed with Mysql NDB cluster (version 7.5.3) in the diskless mode on 12 server nodes. Tectonic [28] was not compared because it is closed-source. The throughput of IndexFS is not included, because the metadata servers of IndexFS consistently terminate with errors.

InfiniFS is implemented with the Thrift RPC library [3] for network communication, and RocksDB [5] for the backend KV store. Thrift uses the Linux TCP/IP network stack, leading to a round-trip time of about 60 μs. For InfiniFS, LocoFS, and IndexFS, we set their metadata servers to the Thrift non-blocking server with 8 worker threads, metadata caches to 8MB, and KV stores in asynchronous write mode.

Benchmark. We use the *mdtest* [1] benchmark to evaluate the metadata performance of the aforementioned distributed filesystems. We use OpenMPI at version 3.0.6 to generate parallel mdtest processes across the client nodes. We modify the mdtest benchmark to use the client libraries provided by each distributed filesystem (e.g., libcephfs of CephFS).

Our experiments create files of zero length, like the previous works [16, 23, 25, 31, 43, 46], as we focus on insights into the metadata performance. For full-fledged distributed filesystems such as HopsFS and CephFS, we guarantee the fairness of comparison by ensuring their data paths were not accessed during experiments.

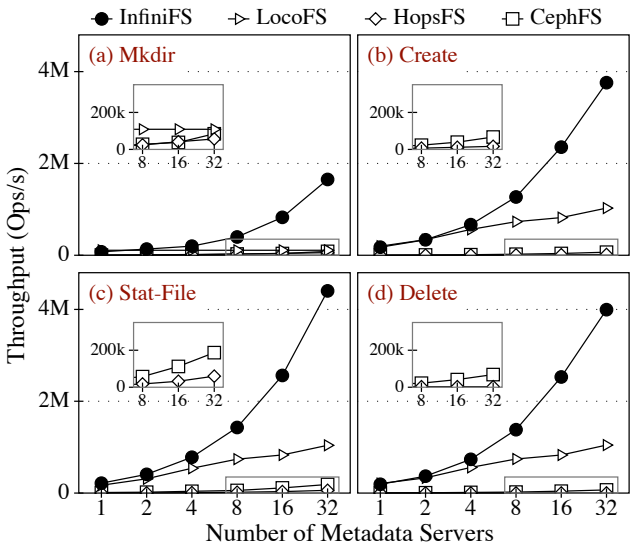


Figure 9: Throughput scalability of metadata operations (`mkdir`, `create`, `stat`, and `delete`). 500 million files.

5.2 Overall Performance

In this section, we compare the overall metadata performance of the aforementioned distributed filesystems. We use mdtest to generate a four-phase workload to measure the performance of directory `mkdir`, file `create`, file `stat`, and file `delete` operations, respectively. These clients operate on a shared directory tree with a depth of 10. Each client handles its own 0.1 million directories and 0.1 million files, which are evenly distributed in the shared directory tree. Clients are initiated at the beginning of each phase.

5.2.1 Throughput

In this section, we evaluate the throughput scalability of metadata operations in different distributed filesystems. We scale the number of metadata servers from 1 to 32, and measure the peak throughput that each filesystem can provide. To obtain the peak throughput, we gradually increase the number of clients until the throughput no longer increases. With 2048 client processes, clients process approximately half a billion files during the experiment.

Figure 9 shows the throughput scalability of directory `mkdir`, file `create`, file `stat`, and file `delete` metadata operations in different distributed filesystems. From the figure, we make the following observations:

1) InfiniFS presents near linear throughput scalability in `mkdir`, `create`, `stat`, and `delete` metadata operations, as the number of metadata servers scales from 1 to 32. InfiniFS achieves high scalability by optimizing the two critical

steps of metadata operations, i.e., path resolution and metadata processing. (a) For the path resolution, INFINIFS caches near-root access metadata at the client-side to absorb the read load on near-root directories, thus the near-root hotspot caused by path resolution will not impair scalability. Besides, INFINIFS partitions file/directory metadata across metadata servers by hashing directory IDs. The fine-grained hash partitioning strategy effectively load-balances metadata accesses, achieving high scalability. (b) For the metadata processing, INFINIFS decouples the directory metadata, then groups the directory access metadata with the parent and the directory content metadata with the children. In this way, metadata processing of file `create`, `stat`, and `delete` only accesses one single server requiring no cross-server coordination, thus, being scalable. Metadata processing of directory `mkdir` requires coordination with only two servers for atomicity, which also scales well with more servers.

2) According to Figure 9(a) and 9(b), the throughput of the `mkdir` operation is much lower than the throughput of the `create` operation in INFINIFS. This is because file creation is implemented using the local transaction protocol with no cross-server coordination, while directory creation requires two-phase locking and two-phase commit protocols. These distributed protocols require expensive coordination between servers, leading to lower throughput.

3) With one metadata server, the throughput of file `create` is 180K ops/sec in INFINIFS, which is slightly lower than LocoFS (200K ops/sec). This is because LocoFS uses a hash-based KV store, which provides higher performance than RocksDB while does not support the scan operation. LocoFS also decouples file metadata into two finer key-value pairs for high throughout. INFINIFS outperforms LocoFS as the number of metadata servers increases. With 32 servers, the directory `mkdir` and file `stat` operations of INFINIFS are 18× and 4× higher than LocoFS. This is because INFINIFS partitions the directory metadata to multiple servers, while LocoFS manages all the directory metadata in one single directory metadata server. When the number of clients increases and the directory tree expands, the single directory metadata server in LocoFS becomes the throughput bottleneck.

4) INFINIFS achieves higher metadata operation throughput than HopsFS and CephFS. For file `create` operations, the throughput of INFINIFS is 73× and 23× higher than that of HopsFS and CephFS, respectively. This is because INFINIFS reduces the latency of metadata operations by resolving paths speculatively in parallel, thus achieving a higher base throughput than HopsFS and CephFS.

5.2.2 Latency

In this section, we evaluate the latency of metadata operations in different distributed filesystems. In the evaluation, we use 32 metadata servers and measure the latency of each metadata operation issued by the client.

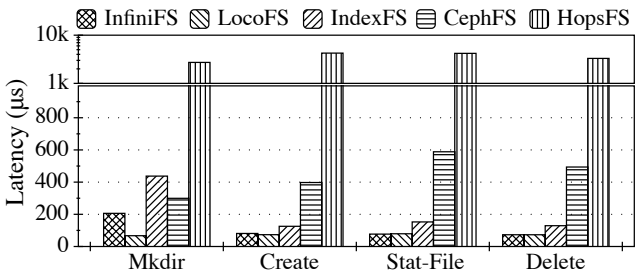


Figure 10: Latency of metadata operations.

Figure 10 shows the average latency of different metadata operations in the evaluated distributed filesystems. From the figure, we make the following observations:

1) For file `create`, file `stat`, and file `delete` operations, INFINIFS achieves comparable low latency with LocoFS, This is because the speculative path resolution and the optimistic access metadata cache in INFINIFS reduce the latency of path resolution, and metadata processing completes within a single server. INFINIFS has higher latency in `mkdir` than LocoFS. This is because INFINIFS partitions directory metadata across metadata servers, causing the directory creation operation to be a distributed transaction. On the contrary, LocoFS manages all directory metadata on one single directory metadata server. Thus, all directory metadata operations can complete in nearly one round-trip time (RTT).

2) INFINIFS achieves lower latency than IndexFS, CephFS, and HopsFS. This is because IndexFS and HopsFS require recursive RPCs to resolve pathnames in case of cache misses, which is slower than our parallelized approach. Besides, CephFS and HopsFS store metadata through external distributed object storage and MySQL NDB cluster, which increases the software stack and results in high latency.

5.3 Factor Analysis

In this section, we analyze how the design features contribute to the latency and throughput by breaking down the performance gap between the Baseline and INFINIFS. We accumulate design features into the Baseline, and measure the latency and throughput of file `create` and directory `mkdir` on 32 metadata servers. For the latency breakdown evaluation, we initiate one mdtest client to create empty files at a depth of 10, and measure the time consumption of path resolution and metadata processing separately. For the throughput breakdown evaluation, we initiate 1024 mdtest clients. Each client creates 0.1 million files or directories that are evenly distributed on a shared directory tree with a depth of 10.

Baseline. We implement the baseline upon the framework of INFINIFS, which partitions the directory tree at the per-directory granularity (like IndexFS, HopsFS, and Tectonic), but without the three design features. As shown in the bars in Figure 11, the path resolution takes 72% of the overall

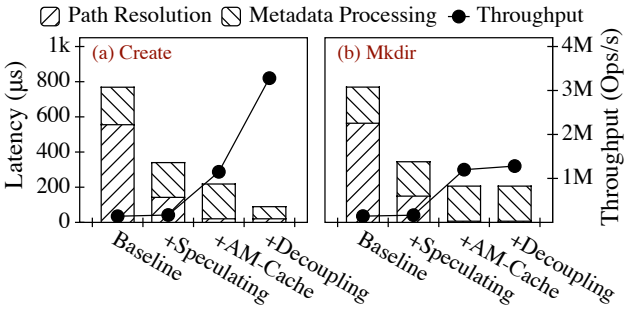


Figure 11: Contributions of design features to the latency (left Y-axis) and throughput (right Y-axis) of INFINIFS. Different segments inside the bar represent the decomposed latency. Design features are accumulated.

metadata operation latency, around 9 RTTs. This is because the path resolution needs to traverse and check permissions of all intermediate directories inside the path sequentially. The metadata processing takes 28% of the latency, around 3 RTTs. This is because both the file and directory creation need to create the target metadata and modify the parent's timestamps and entry list, thus requiring cross-server coordination. As shown in the lines in Figure 11, the metadata operation throughput is very low in the Baseline. This is because all metadata operations will access the near-root directories during path resolution, causing the overall throughput to be limited by the server with the near-root directories.

+Speculating. With the speculative path resolution, the latency of the path resolution is reduced down to 26% of the Baseline. The client leverages asynchronous RPCs to parallelize network requests, but the request processing overhead inside the Linux TCP/IP network stack keeps accumulating. This causes the latency of speculative path resolution to be more than one RTT. The speculative path resolution still faces the problem of near-root bottlenecks, so the metadata operation throughput remains nearly the same.

+AM-Cache. With the optimistic access metadata cache, the heavy read load of path resolution on near-root directories can be absorbed by the client-side cache. Thus, the near-root hotpot will not impair the filesystem throughput. This boosts the overall throughput of file `create` and directory `mkdir` operation to more than 1M ops/sec. Besides, cache hits will further speed up the path resolution, reducing the latency of path resolution to less than one RTT.

+Decoupling. The metadata processing of file `create` and directory `mkdir` involves three metadata objects, including the new file/directory metadata, the entry list, and the timestamps of the parent directory. Without the directory metadata decoupling, these metadata objects are typically located on different metadata servers after the directory tree partitioning. Therefore, metadata processing involves expensive cross-server coordination. With the decoupling, we group the entry list and the timestamps of each directory with the files underneath.

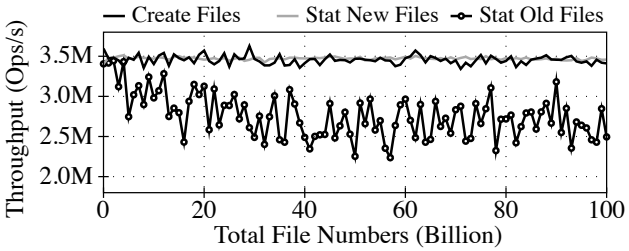


Figure 12: Throughput of INFINIFS with 100 billion files.

In this way, file `create` can process related metadata within a single server with no cross-server coordination. Decoupling boosts the file `create` throughput to 3.3M ops/sec, and reduces its latency to nearly one RTT. However, the throughput and latency of directory `mkdir` remain the same, as it still involves metadata objects across two servers.

5.4 Large-Scale Directory Tree

In this section, we demonstrate that INFINIFS can efficiently support large-scale directory trees. In this experiment, we deploy INFINIFS on 32 metadata servers and initiate 1024 clients. We increase the size of the directory tree up to 100 billion files. Each time we first insert one billion files into the directory tree, then generate a three-phase workload to measure the current performance. Specifically, we measure the throughput of file `create`, file `stat` at the new files, and file `stat` at the old files which are created at the beginning.

Evaluation results are shown in Figure 12. From the figure, we make the following observations:

1) INFINIFS can provide steady performance for file `create` and file `stat` operations (~ 3.5M ops/sec), even when the directory tree expands to a huge size (100 billion files). In real-world datacenters, the Tectonic of Facebook manages 10.7 billion files [28], and the datacenter of Alibaba Cloud maintains up to tens of billions of files. As a result, we believe INFINIFS matches real-world scenarios by supporting 100 billion files with stable performance.

2) The throughput of `stat` old files is lower than that of `stat` new files. This is because INFINIFS stores metadata in RocksDB, which holds key-value pairs in multiple levels of SSTables. Performance drops slightly as the old key-value pairs are mitigated to lower levels. The lower the level, the higher the capacity, and therefore the performance degradation slows down (only ~ 20% at 100 billion files.).

5.5 Cache Efficiency

In this section, we evaluate the efficiency of the lazy invalidation mechanism. To compare with the lease mechanism, we implement a lease version of INFINIFS that uses lease to maintain cache coherence. In the experiments, we deploy

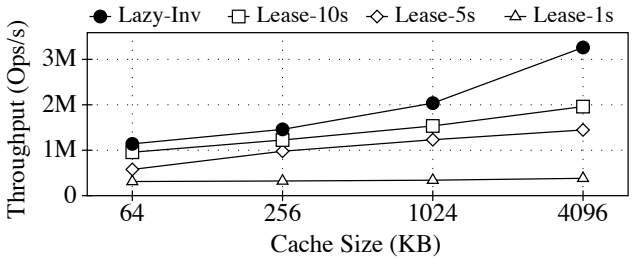


Figure 13: Comparisons of the lazy invalidation with the lease mechanism. The lease expiration time is set to 1s, 5s, and 10s.

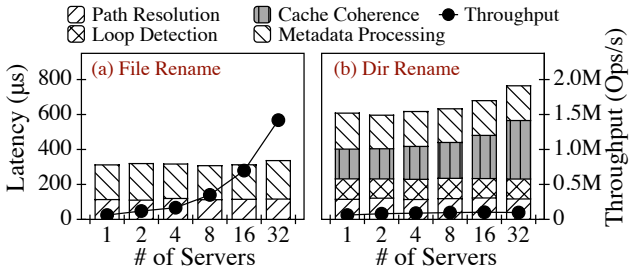


Figure 14: Latency (left Y-axis) and throughput (right Y-axis) of file `rename` and directory `rename`. Different segments inside the bar represent the decomposed latency.

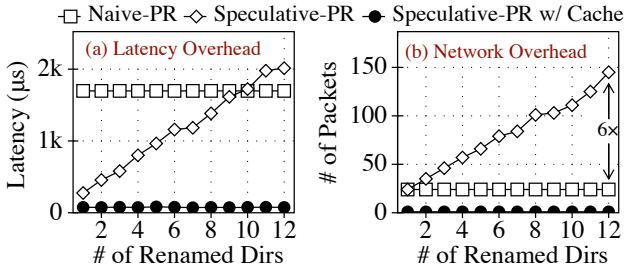


Figure 15: Latency and network packets of three different path resolution approaches. PR: path resolution.

INFINIFS on 32 metadata servers and initiate 2048 clients to `stat` files.

Evaluation results are shown in Figure 13. From the figure, we make the following observations:

1) The lazy invalidation mechanism outperforms the lease mechanism. This is because, in the lease mechanism, cache entries expire periodically regardless of the cache size. Increasing the lease expiration time can improve the throughput, but also increases the latency of all modification operations.

2) As the cache size increases, the throughput increment is more pronounced for the lazy invalidation mechanism than the lease mechanism. This is because the lease mechanism suffers from load imbalance caused by cache renewals at the near-root directories. All clients have to repeatedly renew their cache entries at the near-root directories for the path resolution procedure. As the number of clients is huge, such load imbalance eventually becomes the performance bottleneck, impairing the overall throughput.

5.6 Rename

In this section, we evaluate the latency and throughput of file `rename` and directory `rename` operations in INFINIFS. In the experiments, we scale the number of metadata servers and measure the peak throughput. We break down `rename` operations to measure the time consumption of each phase.

1) We find that the latency of file `rename` is much lower than directory `rename`. Moreover, as the number of servers increases, the latency of file `rename` remains steady, while the latency of directory `rename` increases slowly. This is because the directory `rename` operation is more complex than the others, involving the following four steps: (1) resolve the source and the destination path, (2) detect whether it leads to orphaned loops, (3) broadcast modification information to metadata servers to maintain cache coherence, and (4) process related metadata across two servers. The latency of directory `rename` grows slowly as the number of servers grows, because the broadcast messages are sent to servers in parallel.

Evaluation results are shown in Figure 14. From the figure, we make the following observations:

2) We find that the throughput of file `rename` scales with the number of servers, while the throughput of directory `rename` does not scale (stays near 10K ops/sec). This is because file `rename` only requires coordination with the source and destination metadata servers for atomicity, thus scales well. However, directory `rename` operations require the central coordination server for orphaned loop detection and broadcasting, thus do not scale. However, these operations rarely occur (accounting for ~ 0.0083%) based on the read-world workload studies in §2.3. Therefore, one single coordination server should be sufficient to handle these infrequent operations.

5.7 Overhead of Misprediction

In this section, we evaluate the overhead of misprediction, including latency overhead and network overhead. A misprediction means INFINIFS mispredicts the ID of a directory during the speculative path resolution. Only the directory `rename` and the hash collision will cause the misprediction. In this experiment, we generate a directory path with a depth of 24, and increase the counts of mispredictions by adding more renamed directories within the path. The renamed directories are evenly distributed inside the pathname. We create 10K files under the path with one client, and measure the average latency and network packets during path resolution.

Evaluation results are shown in Figure 15. From the figure, we make the following observations:

1) Mispredictions increase the latency of the speculative path resolution, but do not affect the naive approach. However,

speculative path resolution still outperforms naive path resolution unless nearly half of the intermediate directories have been renamed, which is rare. Besides, with the optimistic access metadata cache, clients can cache the true IDs of renamed directories, thus avoid mispredictions in the future.

2) Mispredictions increase the number of network packets required for the speculative path resolution. However, trading excess network bandwidth for lower metadata operation latency is worthwhile, as the network bandwidth is not the performance bottleneck. For example, the 25-Gbps CX4 NIC can process 12.3 million packets per second [19]. Thus, 32 metadata servers can process a total of 393.6 million packets per second, which is substantially higher than the peak throughput of metadata operations (~ 4M ops/sec), Besides, INFINIFS comes with the optimistic access metadata cache on the client side, so as to eliminates the extra packets by avoiding mispredictions in the future.

6 Related Work

Efficient distributed filesystems have always been important research topics. Due to the rapidly increasing file quantities, metadata service becomes the performance bottleneck for large-scale distributed filesystems [22, 32, 33, 36].

Directory tree partitioning. Early distributed filesystems, such as GFS [12], HDFS [35], Farsite [11], and QFS [27], distribute file data to multiple data servers while managing all metadata in a single dedicated metadata server. However, they fail in the extremely large-scale scenario with billions of files, because the amount of metadata exceeds the capacity of a single server, and the throughput of metadata operations will be bottlenecked due to the limited resources.

Some distributed filesystems partition the directory tree into subtrees, such as AFS Volumes [15], Sprite Domain [26], and HDFS Federation [9, 35]. Subtree-based metadata partitioning can achieve high metadata locality, but suffers from low scalability due to load imbalance and data migration. Some distributed filesystems partition the directory tree into user-visible partitions and disallow cross-partition renames. As the directory tree expands, organizing and maintaining the static partitioning scheme becomes impractical. CephFS [6, 39–41] partitions metadata into subtrees as well, but when a load imbalance is detected, it migrates hot subtrees across metadata servers. Mantle [34] provides a programmable interface to adjust CephFS's balancing policy for various metadata workloads. However, they suffer from the high overhead of frequent metadata migrations, when workloads are diverse and vary frequently.

Some distributed filesystems partition the directory tree at the per-directory granularity, such as IndexFS [31, 46], HopsFS [25], and Tectonic [28]. Due to the fine-grained partitioning, they can achieve load balancing and good scalability. However, they sacrifice the metadata locality, causing frequent distributed locks and distributed transactions. These distributed protocols impose expensive coordination overhead, resulting in high latency and low throughput [8, 20].

Path resolution. LocoFS [23] stores all directory metadata on a single metadata server, in order to reduce the latency of path resolution, However, it suffers from the single node bottleneck. Some distributed filesystems use the full pathname or the hashing on the full pathname to index files, such as BetrFS [17, 18, 44, 45], Giraffa [37], and CalvinFS [38]. BetrFS uses the full pathname to index files in the local filesystem. Giraffa uses the full pathname as the primary key to the file metadata. CalvinFS locates the file metadata by hashing the full pathname. However, they make the hierarchy semantic to be hard to implement. For example, the directory rename operation becomes prohibitively costly, as it changes the full pathname of all descendants, causing all descendants' metadata must be migrated to new locations.

Client-side metadata caching. HopsFS caches the metadata location information on the server side to parallel path resolution. However, it suffers from the near-root hotspot, as all metadata operations need to read the near-root directories for path traversing and permission checking. LocoFS [23], IndexFS [31], and NFS v4 [30] leverage the lease mechanism to cache both the directory entries and permissions on the client side. However, the lease mechanism suffers from load imbalance caused by cache renewals at the near-root directories. As the number of clients increases, such load imbalance at the near-root directories will become the performance bottleneck, impairing the overall throughput.

7 Conclusion

This paper presents INFINIFS, an efficient metadata service for extremely large-scale distributed filesystems. INFINIFS decouples the directory's access and content metadata, so that the directory tree can be partitioned with both high metadata locality and good load balancing; then parallelizes path resolution with speculation to substantially reduce the latency of metadata operations; and finally, cache access metadata on the client-side optimistically with lazy invalidation. The extensive evaluation shows that INFINIFS provides high-performance metadata operations for large-scale filesystem directory trees.

Acknowledgments

We sincerely thank our shepherd Brent Welch and the anonymous reviewers for their valuable feedback. We also thank Wenhui Yao from the Alibaba Group, Qing Wang, Xiaojian Liao, Youmin Chen, and Zhe Yang from the Tsinghua University for the help on this work. This work is supported by the National Natural Science Foundation of China (Grant No. 61832011, 62022051). This work was also supported by Alibaba Group through Alibaba Innovative Research Program.

References

[1] Mdtest HPC Benchmark. https://sourceforge.net/projects/mdtest/, 2014.

[2] Pangu: The High Performance Distributed File System by Alibaba Cloud. https://www.alibabacloud.com/blog/pangu-the-high-performance-distributed-file-system-by-alibaba-cloud_594059, 2018.

[3] Apache Thrift. https://thrift.apache.org/, 2021.

[4] HDFS Federation. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html, 2021.

[5] RocksDB. http://rocksdb.org/, 2021.

[6] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 353–369, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A Five-Year Study of File-System Metadata. *ACM Trans. Storage*, 3(3):9–es, oct 2007.

[8] James Cowling and Barbara Liskov. Granola: Low-Overhead Distributed Transaction Coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 223–235, Boston, MA, June 2012. USENIX Association.

[9] Dipayan Dev and Ripon Patgiri. Dr. Hadoop: an infinite scalable metadata management for Hadoop—How the baby elephant becomes immortal. *Frontiers of Information Technology & Electronic Engineering*, 17(1):15–31, 2016.

[10] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '99, page 59–70, New York, NY, USA, 1999. Association for Computing Machinery.

[11] John R. Douceur and Jon Howell. Distributed Directory Service in the Farsite File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 321–334, USA, 2006. USENIX Association.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 29–43, New York, NY, USA, 2003. Association for Computing Machinery.

[13] Ibrahim F. Haddad. PVFS: A Parallel Virtual File System for Linux Clusters. *Linux J.*, 2000(80es):5–es, nov 2000.

[14] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into Google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system, 2021.

[15] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, feb 1988.

[16] Mahmoud Ismail, Salman Niazi, Mikael Ronström, Seif Haridi, and Jim Dowling. Scaling HDFS to More than 1 Million Operations per Second with HopsFS. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, page 683–688. IEEE Press, 2017.

[17] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 301–315, USA, 2015. USENIX Association.

[18] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. BetrFS: Write-Optimization in a Kernel File System. *ACM Trans. Storage*, 11(4), nov 2015.

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter rpcs can be general and fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 1–16, USA, 2019. USENIX Association.

[20] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, aug 2008.

[21] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander Sandler. Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 15–31, USA, 2019. USENIX Association.

[22] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 213–226, USA, 2008. USENIX Association.

[23] 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*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[24] 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.

[25] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. HopsFS: Scaling hierarchical file system metadata using NewSQL databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 89–104, Santa Clara, CA, February 2017. USENIX Association.

[26] J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, and B.B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, 1988.

[27] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.

[28] 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, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231. USENIX Association, February 2021.

[29] Swapnil Patil and Garth Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 13, USA, 2011. USENIX Association.

[30] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*, 2000.

[31] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, page 237–248. IEEE Press, 2014.

[32] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, page 4, USA, 2000. USENIX Association.

[33] Margo Seltzer and Nicholas Murphy. Hierarchical File Systems Are Dead. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, page 1, USA, 2009. USENIX Association.

[34] Michael A. Sevilla, Noah Watkins, Carlos Maltzahn, Ike Nassi, Scott A. Brandt, Sage A. Weil, Greg Farnum, and Sam Fineberg. Mantle: A Programmable Metadata Load Balancer for the Ceph File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[35] 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)*, pages 1–10, 2010.

[36] Konstantin V Shvachko. HDFS Scalability: The limits to growth. *; login:: the magazine of USENIX & SAGE*, 35(2):6–16, 2010.

[37] Konstantin V Shvachko and Yuxiang Chen. Scaling Namespace Operations with Giraffa File System. *USENIX; login*, 2017.

[38] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 1–14, USA, 2015. USENIX Association.

[39] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 307–320, USA, 2006. USENIX Association.

[40] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-Scale Storage Clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, page 35–44, New York, NY, USA, 2007. Association for Computing Machinery.

[41] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, page 4, USA, 2004. IEEE Computer Society.

[42] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, USA, 2008. USENIX Association.

[43] Lin Xiao, Kai Ren, Qing Zheng, and Garth A. Gibson. Shardfs vs. indexfs: Replication vs. caching strategies for distributed metadata management in cloud storage systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 236–249, New York, NY, USA, 2015. Association for Computing Machinery.

[44] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Optimizing every operation in a write-optimized file system. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, page 1–14, USA, 2016. USENIX Association.

[45] Yang Zhan, Alex Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The Full Path to Full-Path Indexing. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 123–138, Oakland, CA, February 2018. USENIX Association.

[46] Qing Zheng, Kai Ren, and Garth Gibson. BatchFS: Scaling the File System Control Plane with Client-Funded Metadata Servers. In *Proceedings of the 9th Parallel Data Storage Workshop*, PDSW '14, page 1–6. IEEE Press, 2014.

ScaleXFS: Getting scalability of XFS back on the ring

Dohyun Kim Kwangwon Min Joontaek Oh Youjip Won
Department of Electrical Engineering, KAIST

Abstract

This paper addresses the scalability issue of XFS journaling in the manycore system. In this paper, we first identify two primary causes for XFS scalability failure: the contention between in-memory logging and on-disk logging and the contention among the multiple concurrent in-memory loggings. We then propose three key techniques to address the scalability issues of XFS; (i) Double Committed Item List, (ii) Per-core In-memory Logging, and (iii) Strided Space Counting. Contention between the in-memory logging and the on-disk logging is mitigated using the Double Committed Item List. Contention among the in-memory logging operations is addressed with Per-core In-memory Logging. Strided Space Counting addresses the contention on updating the global journaling state. We call the newly developed filesystem ScaleXFS. In modified `varmail` workload, the latency of metadata operation, `unlink()`, decreases to 1/6th from 0.59 ms to 0.09 ms under 112 threads against stock XFS. The throughput of ScaleXFS corresponds to 1.5×, 2.2×, and 2.1× of the throughput of the stock XFS under `varmail`, `dbench` and `mdtest`, respectively.

1 Introduction

Modern computing platforms are experiencing two technical advancements; storage devices are getting quicker and faster with sub-msec flush latency and the number of CPU cores in the commodity server is reaching the hundreds [16,29,38,39]. Considering these circumstances, modern filesystem designs face the new challenges, such as manycore scalability of the filesystem operation.

The filesystem is responsible for two types of operations; updating the state of the in-memory filesystem and synchronizing the in-memory filesystem state to the storage. A substantial effort has been dedicated to improving the throughput and the latency of synchronizing the filesystem state to the disk, also known as *filesystem journaling* [14, 35, 44, 47]. These works aim to hide flush latency: the latency of making the updated filesystem blocks durable in storage. A number of techniques have been proposed to make the flush latency invisible to the host, e.g. `no_barrier` mount operation [20], the adoption of the power loss protection feature at the flash storage [46], the kernel patch that omits the flush command [1], etc. Thanks to all these techniques, the latency to synchronize the filesystem to the storage has become less of a concern.

As the filesystem operation is relieved from the excessive flush latency, many works have focused on improving the throughput of filesystem journaling via increasing the concurrency in filesystem journaling. They include providing multiple running transactions [27, 40], providing multiple committing transaction [26,27,34,40,48], using the lock-free algorithm to manage the log block list in more concurrent manner [43].

In this work, we revisit the manycore scalability issue in the modern filesystem. In particular, we focus our effort on XFS. The XFS filesystem is one of the most widely used journaling filesystems in the enterprise server as well as in cloud platforms. XFS is the default filesystem for RHEL [2] which operates 33% of the enterprise servers [6] and handles more than 50% of stock transactions in the world [7]. Until recently, it has also been the default backend filesystem for distributed storage system, Ceph [13].

Despite the significance of XFS filesystems in modern computing platforms, few works have explored the performance and scalability aspect of the XFS and/or propose the solution in a rigorous manner. Existing articles on XFS are introductory article about the XFS data structures and algorithms [10], personal comment on the design of the XFS [19], comparison of the multiple filesystem performances without detailed analysis [12, 23, 37]. A few works coined performance and scalability issue of the XFS filesystem [13,37], whose solution has yet to be addressed.

Most works regarding filesystem scalability use EXT4 as their baseline filesystem [26, 27, 34, 40, 43, 48]. While both XFS [44] and EXT4 [35] use journaling to synchronize the filesystem state to the disk, few of the scalability techniques for EXT4 are readily applicable for XFS. This is because the details of their journaling subsystem designs lie at the other end of the extreme. XFS uses multiple granularity differential logging while EXT4 uses page granularity physical logging. XFS uses the copy of the update for journal commit while EXT4 uses the original page cache entry for journal commit. XFS allows multi-threaded concurrent journal commit while EXT4 has single threaded serial commit. As long as filesystem journaling is concerned, XFS adopts far more sophisticated data structure to make filesystem journaling more efficient and scalable. Most works that deal with EXT4 journaling scalability are about how to migrate the page cache entries among the running transactions (or committing transactions) when multiple threads update the same page cache

entry [24,26,27,34,40,48]. XFS is free from the page conflict since the filesystem operation creates its own copy of the update.

In this work, we examine the scalability of the XFS filesystem under three different workloads and perform in-depth analysis to find the prime cause for the observed scalability failure. Through this analysis, we find that the contention on the two locks are the root cause for its scalability failure and is caused by the contention among in-memory logging activities and the on-disk logging activities to access the global log data list which is called the *committed item list* in XFS. Furthermore, we find that the lock contentions observed in XFS are caused by two entirely different system behaviors.

The contribution of this work can be summarized as follows. First, we identify the root cause of the scalability failure in XFS; the contention on `cil_lock` and `ctx_lock-R` (shared lock of `ctx_lock`) that protect the committed item list. Second, we identify the essential filesystem activity that causes the contention on these locks: (i) the contention between in-memory logging and on-disk logging and (ii) the contention among the multiple concurrent in-memory loggings. Third, we propose ScaleXFS to address the scalability issue in XFS. We extend XFS with the techniques proposed below (Linux kernel v5.8.5). ScaleXFS consists of three key technical ingredients as follows.

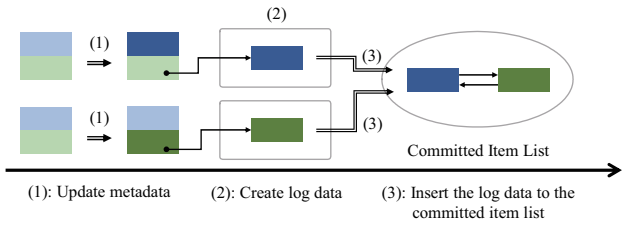
- **Double committed item list** XFS provides a single global committed item list. Unlike stock XFS, ScaleXFS provides two committed item lists so that in-memory logging and on-disk logging can work on its own committed item list. Double Committed item list prohibits the on-disk logging from blocking the in-memory logging activity.
- **Per-core in-memory logging** ScaleXFS forms each committed item list with a set of per-core committed item lists to mitigate the lock contention on the in-memory loggings. The application threads must acquire an exclusive lock on the committed item list when they insert the log data to the committed item list. Under the manycore system, multiple applications can perform in-memory logging concurrently and may compete for acquiring the exclusive lock on the committed item list. To mitigate the contention among the multiple concurrent in-memory loggings, ScaleXFS adopts a per-core committed-item list.
- **Strided Space Counting** XFS maintains a global state of the committed item list, which must be updated each time the log data is added to the committed item list. The application threads must acquire an exclusive lock on the global state of the committed item list when it updates the state. To mitigate the contention on accessing the global state of the list, we adopt sloppy counter [17] style per-core distributed counting scheme for maintaining the state of the committed item list. We call this, *Strided Space Counting*.

We believe that the beauty of ScaleXFS is its elegant simplicity. We overhaul the XFS journaling behavior, precisely identify the scalability bottleneck and propose a simple yet elegant mechanism that addresses the scalability issues in the XFS filesystem with minimal modifications. Unlike the other approaches that modify the on-disk layout of the existing filesystem partition [26, 27, 34, 40], the filesystem layout remains unchanged. ScaleXFS can mount the existing XFS partition. In ScaleXFS, the latency of `unlink()` decreases to 1/6th under modified `varmail` workload [1] [45], compared to stock XFS. In modified `varmail` and `dbench` [11] workloads, ScaleXFS renders as much as 1.5× and 2.2× performance against stock XFS, respectively. In ScaleXFS, the performance scales well till the number of cores reaches sixty while in stock XFS, the performance saturates beyond twelve cores under `mdtest` [33].

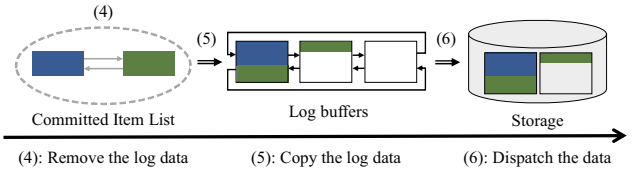
2 Background

2.1 XFS

XFS has been introduced to address the scalability issue of then Unix filesystem [44] (e.g. EFS [42] and FFS [36]). It supports a full 64-bit filesystem. It partitions a filesystem into a number of fixed-size partitions, the *allocation groups*, for scalable access to the filesystem metadata [44]. It uses B+ tree to manage the free blocks and the free inodes, respectively, and it uses hashing and B+ tree to manage a large number of directory entries within the directory.



(a) Filesystem Update



(b) Filesystem Journaling (journal thread)

Figure 1: Updating the filesystem object and logging the associated updates to the disk.

For journaling, XFS adopts multi-granularity differential logging [28, 30]. XFS applies different logging granularity subject to the size of the metadata that is updated. For small metadata such as inode, it creates a copy of the updated metadata for logging. For large metadata, e.g. superblock (4KByte)

[1] Each thread works on its own separate directory.

or B+ tree node (4KByte), it partitions the 4 KByte block into 128 Byte units and creates a copy of the updated region in 128 Byte granularity [9]. Each metadata object of XFS maintains the logs for the associated updates. The updates are synchronized to the disk either through periodic flush (30 sec by default) or by `fsync()` call.

For each metadata update, XFS creates the log data for journaling, which is a copy of the update. XFS maintains three types of log data lists: (i) log data that needs to be committed to the disk (*committed item list*), (ii) log data that is being committed (*committing list*) and (iii) log data that needs to be checkpointed (*active item list*). The filesystem thread creates and migrates the log data among these lists. The application threads and the journaling thread lock these lists and the associated data structure to avoid race condition. The contention among these threads leaves the XFS under potential scalability failure [44].

We can categorize the filesystem operations into two; metadata operation, e.g. `creat()` and `unlink()` and journaling operation `fsync()`. We explain the details of the metadata operation and the journaling operation in the following sections from the aspect of filesystem journaling.

2.2 Metadata Operation in XFS

For metadata operation, the filesystem acquires the exclusive lock on the metadata it needs to update. Then, it updates the metadata and performs *in-memory logging*. In-memory logging consists of two phases: (i) creating the log data and (ii) inserting the newly created log data into the committed item list. In the first phase, the application establishes the shared lock on the committed item list and creates the log data. When the application modifies the metadata for which log data is already in the list, it replaces the existing log data with a newly created one. In practice, a committed item list is a list of the keys (`Log Item`) each of which refers to the associated log data. Separating the key from the value, the log data can be replaced with a new one in the committed item list without deleting and inserting the entry in the committed item list. When the application updates the metadata whose log data is already in the committed item list, the associated entry in the committed item list is updated to refer to the newly created log. In the first phase of in-memory logging, the application establishes a shared lock on the committed item list. By using the shared lock, multiple applications (or threads) can concurrently update the log entries in the committed item list. The shared lock prohibits the journal thread from committing the committed item list if in-memory logging is in-progress. For the second phase, the filesystem establishes the exclusive lock (spinlock) on the committed item list and inserts the newly created log data. XFS inserts the new log data at the end of the committed item list. When it updates the existing log data, the updated log data is moved to the end of the committed item list.

Fig. 2 illustrates the case when the same inode is updated twice before the updates are committed to disk. First, the file attribute of the inode was updated to A'. The associated log is inserted to the committed item list. Second, the inline data of the inode is updated to B'. In the second update, the application creates the new log data which harbors both updates, [A',B'], and replaces the existing log [A'] with the new log data of [A',B'].

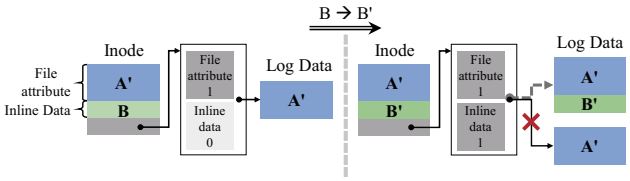


Figure 2: Differential Logging and the log management: An inode is updated twice. The first update is for the file attribute. The second update is for the inline data of the inode.

The filesystem maintains the state of the committed item list, e.g. the total size of the log data in the list. The filesystem updates this state when the log data is added to (or deleted from) the list. After the log data is inserted (or deleted) in the committed item list and the state is updated, the application releases the exclusive lock (spinlock) and the shared lock in the reverse order in which they are acquired.

2.3 Journaling Operation in XFS

Filesystem journaling is triggered through periodic journal flush (30 sec by default), by `fsync()` call, or when the size of the outstanding logs in memory exceeds a certain threshold. We define the filesystem journaling operation as on-disk logging for short. XFS defines *log buffer* as a unit of IO in on-disk logging. The default size of the log buffer is 32 KByte. XFS organizes the on-disk logging with two separate tasks and dedicates different threads for each. In this work, we call these two types of threads as the *journal thread* and the *commit thread*, respectively.

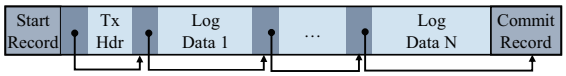


Figure 3: Structure of Transaction in Log Buffer.

XFS creates a new journal thread each time when the on-disk logging is triggered and can start new on-disk logging before the preceding one completes. The newly created journal thread establishes an exclusive lock on the committed item list and migrates the log data from the committed item list to the committing list. Then, it populates the log buffer with the log data in the committing list. If the log buffer becomes full, the journal thread flushes the log buffer to the disk (with `PREFLUSH` and `FUA` flags [22]). If all log data are copied to

the log buffer, the journal thread puts the commit record at the end of the sequence of the log data in the log buffer.

Fig. 3 illustrates the structure of the transaction in the log buffer. When the journal thread writes the commit record at the log buffer, the journal thread returns.

In copying the log data to the log buffer, XFS first moves the log data in the committed item list to the committing list and then releases the exclusive lock on the committed item list. After the journal thread migrates the log data from the committed item list to the committing list, the journal thread continues on-disk logging such as populating the log buffer with the log data from the committing list and flushing the log buffer when it is full. After the journal thread releases the exclusive lock on the committed item list, the committed item list becomes available for subsequent in-memory logging or on-disk logging. XFS allocates the non-overlapping partition of the log buffer to each journal thread. A number of journal threads can concurrently update the non-overlapping partitions of the same log buffer.

The committed item list has a unique transaction ID. It is allocated each time when new on-disk logging starts. When the journal thread holds the exclusive lock on the committed item list, it assigns the transaction ID to the committed item list. The filesystem monotonically increases the transaction id each time when the new journal thread is created, i.e. when the new on-disk logging starts. XFS maintains a set of transaction IDs that are being committed. When the journal thread finishes migrating the log data in the committed item list to the log buffer, it inserts the associated transaction id to the set of committing transactions. When the transaction is made durable, the transaction ID is removed from this set. All log data in the same committed item list are committed to storage as a single transaction.

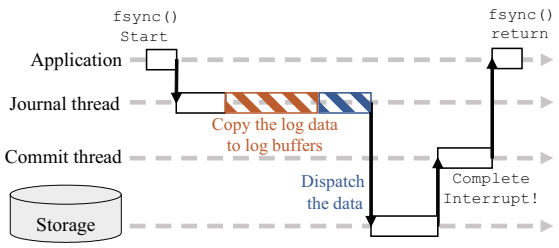


Figure 4: `fsync()` in XFS.

The commit thread is responsible for finishing the journal commit. When the host receives an interrupt informing that the log buffer has been made durable, the host creates the commit thread to handle the rest of the journal commit procedure. The commit thread marks the status of the log buffer as durable (`done_sync`). It then checks if all preceding log buffers have been made durable. If any of the preceding log buffers have not been made durable yet, the commit thread returns without finishing the on-disk logging. If it ensures that all preceding log buffers have been made durable, it releases the log buffer, removes the transaction ID from the set of IDs of outstanding transactions, and migrates the log data in the transaction to the checkpoint data list (`Active Item List`). If there exist the following log buffers that have been made durable and which have been waiting for its preceding log buffer to become durable, the commit thread also performs the same task for the associated log buffer, e.g. releases the log buffer and migrates the committed log data to the active item list. Then, the commit thread returns, and the caller of `fsync()` unblocks. Fig. 4 illustrates the execution of `fsync()` in XFS.

2.4 Concurrency in XFS journaling

XFS adopts a sophisticated mechanism for the filesystem scalability. The first is differential logging. Differential logging not only saves log space but also eliminates the conflict between the metadata operation and the journaling operation. Each metadata operation creates its own version of the update and is free from conflict with the other in-memory logging operations [9]. Also, the metadata operation and the journaling operation can proceed in parallel without interfering with each other. In EXT4, the application is blocked when it attempts to modify the page cache entry that is being committed to the journal region. The second is concurrent journal commit; XFS allows multiple transaction commits in flight. The third mechanism is out-of-order on-disk logging; XFS allows multiple threads to commit the journal transaction concurrently and independently. Journal thread places `cycle number` at each disk block of the journal region. The cycle number represents the number of times a given disk block is overwritten. By placing the cycle number at the beginning of each disk block, the journal threads can write the log blocks to the storage in an out-of-order manner and yet the recovery module can recover the filesystem to the correct state in case of the system failure.

3 XFS Scalability

3.1 Scalability Analysis

We conducted a experiment to examine the scaling behavior of the XFS filesystem. In our experiment, two different storage devices and three different workloads were used. Two storage devices, one with and one without the Power-Loss Protection feature, were used: Intel Optane 905P (I905) and the Samsung 970Pro (S970). The three workloads correspond to varmail per-thread DIR (W_{vptd}) which is a variant of the varmail workload of filebench (`varmail-ptd`) [45], client workload of `dbench` (W_d) [11], and `mdtest` (W_m) [33]. Each of the three workloads exhibit different frequency of metadata operation (`creat()` and `unlink()`) and the different intensity of the filesystem journaling (`fsync()`). We develop

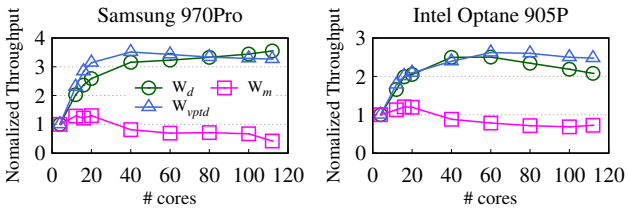


Figure 5: Performance Scalability of XFS. Throughput is normalized against the throughput with the four cores. Server: HPE ProLiant DL580 Gen10 with 112 cores, 4 socket (28 cores per socket), 512 GB DRAM.

the `varmail-ptd` to eliminate the contention on the shared directory in the original `varmail` workload. In the original `varmail`, multiple threads work on a shared directory for creating and deleting the files. The contention on the shared directory occurs at the VFS layer and may make the scaling behavior of the underlying filesystem invisible from the outside. The `varmail-ptd` allocates the separate directory for each thread. The `varmail-ptd` is the most journaling-intensive workload among the three workloads used here. In `varmail-ptd`, `fsync()` accounts for 15.4% of its system calls. In `dbench`, `fsync()` accounts for only 1% of the system calls. There is no `fsync()` in `mdtest`. Table 1 summarizes the characteristics of the three workloads.

Workload	`creat()`	`unlink()`	`rename()`	`read()`	`write()`	`fsync()`
`varmail-ptd`	7.7 %	7.7 %	0 %	15.4 %	15.4 %	15.4 %
`dbench`	17.3 %	3.5 %	0.7 %	27.2 %	8.6 %	1.2 %
`mdtest`	50 %	0 %	0 %	0 %	50 %	0 %

Table 1: Characteristics of workloads: Ratio of individual file system operations.

We use a 112 core machine (HPE ProLiant DL580 Gen10, 28 cores per socket, four sockets) to run all three workloads. We vary the number of active cores from four to 112. The number of threads were set to be equal to the number of cores, and the workload throughput was normalized against the throughput of four active cores. The results are illustrated in Fig. 5. All these workloads fail to scale. The performance of `mdtest` saturates with sixteen cores, and the performance of `dbench` and `varmail-ptd` saturates beyond thirty cores.

To identify the source of scalability failure, we measure the latency of the individual system calls in these workloads, `creat()`, `unlink()` and `fsync()`. Fig. 6 illustrates the results. The `mdtest` is omitted since it does not call `fsync()`.

When `fsync()` becomes quicker, e.g. due to the adoption of the high performance storage device, SSD with Power-Loss-Protection, or the filesystem mounted with `no-barrier` option, the latency of metadata operation, e.g. `unlink` becomes more dominant compared against `fsync()` latency. In `varmail-ptd` (Fig. 6a), the latency of metadata operation increases as `fsync()` gets quicker. This is due to the increased

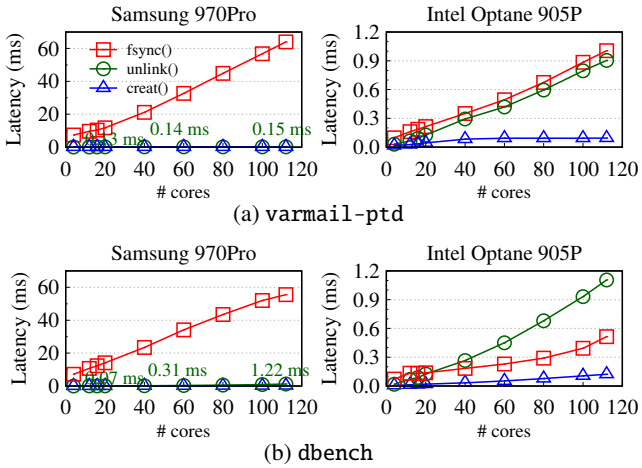


Figure 6: Average latencies of `creat()`, `unlink()` and `fsync()`.

contention on the shared object in the metadata operation. For `varmail-ptd` (Fig. 6a), the `fsync()` latency decreases from 64 ms to 1 ms and the `unlink()` latency increases by 6× from 0.15 ms to 0.904 ms when we compare the performance of S970 and I905. For `dbench` (Fig. 6b), the `fsync()` latency decreases from 56 ms (S970) to 0.51 ms (I905). In I905, due to shorter `fsync()` latency, the `unlink()` latency is 2.1× of the `fsync()` latency.

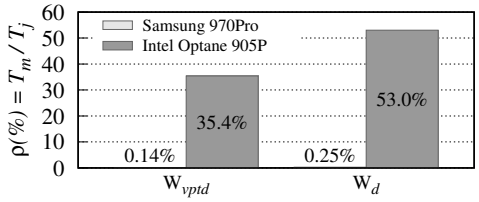


Figure 7: Percentage of lock wait time to T_j. (T_m = Wait time for acquiring lock in in-memory logging, T_j = Wait time for journaling IO in `fsync()`.

For finer analysis, we examine the wait time for acquiring the lock (lock wait time) in in-memory logging and the wait time for journaling IO in `fsync()`. XFS uses two locks; read-write semaphore (`ctx_lock`) and spinlock (`cil_lock`) to protect the committed item list and the log data from the race condition. The lock wait time for in-memory logging denotes the wait time for acquiring these two locks (`ctx_lock` and `cil_lock`). The wait time for journaling I/O in `fsync()` denotes the latency to writing the log blocks to the storage. This corresponds to the length of time interval from when the journal thread puts the commit record at the log buffer till when `fsync()` returns. Fig. 7 illustrates the ratio between the lock wait time and the wait time for journaling IO in S970 and I905, respectively. In S970, the lock wait time is less than 1% of the latency of journal I/O under both workloads. In I905,

the lock wait time becomes more significant. The ratio of lock wait time to wait time for journaling IO is 35.4% and 53.0% under `varmail-ptd` and `dbench`, respectively, in I905.

3.2 Component Analysis

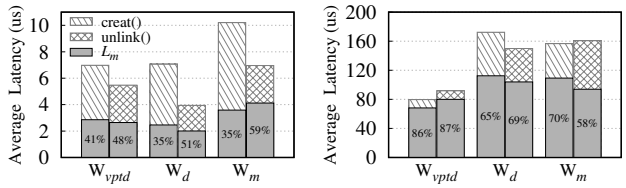


Figure 8: Average latency of `creat()` and `unlink()` and the ratio of the in-memory logging, L_m: the latency of in-memory logging. Storage Device: I905.

Based on the aforementioned experimental results, we found that the metadata operations, `creat()` and `unlink()`, are the prime suspect for the scalability failures in all three workloads. We examine the details of these filesystem operations and identify the main cause of scalability failure. The metadata operation consists of the metadata update and in-memory logging. We run three different metadata intensive workloads; `varmail-ptd`, `dbench` and `mdtest`. In each workload, we measure the latency of the metadata update and the latency of in-memory logging operations for `creat()` and `unlink()`, respectively. Fig. 8 illustrates the overhead of in-memory logging in `creat()` and `unlink()`, respectively. To examine performance behavior while varying the number of cores, we compared the latencies of metadata operations under four cores and 112 cores. In both workloads, the latency of the in-memory logging increases substantially with the increase in the number of cores. When there are four cores, the time for in-memory logging accounts for less than 50% of the latency of metadata operation in most cases (Fig. 8a). When there are 112 cores, in-memory logging accounts for as much as 90% of the latency of metadata operation in W_{vptd} workload (Fig. 8b).

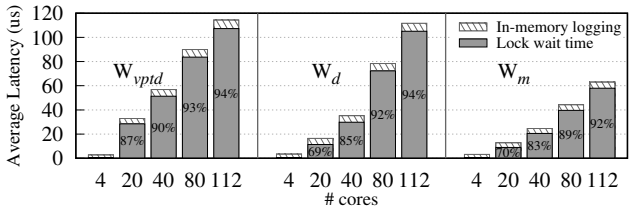


Figure 9: Lock wait time in in-memory logging, Storage Device: I905.

We examine the details of the in-memory logging overhead with the varying number of cores. We measure the wait time for acquiring the lock (lock wait time) and the actual time to update the committed item list and the log data. As in Fig. 9, the lock wait time increases with the number of cores while the actual time for inserting the log data to the committed item list remains unchanged regardless of the number of cores. When there are 112 cores, the lock wait time accounts for more than 90% of the total in-memory logging time.

3.3 Lock Contention Analysis

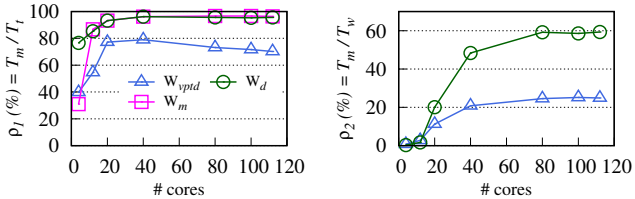


Figure 10: Percentages of lock overhead for in-memory logging to (a) or (b). T_m = Wait time for in-memory logging, T_t = Wait time for all locks, T_w = Workload Execution time. Storage Device: I905.

To precisely identify the scalability failure cause in in-memory logging, we analyze the lock contention in the filesystem. We use `Lockstat` [5] to obtain the lock wait times which are shown in Fig. 10.

Fig. 10a shows the ratio between the lock wait time associated with the in-memory logging (`ctx_lock` and `cil_lock`) against the total lock wait time in the filesystem under three workloads. The total wait time in the file system includes the wait time for all locks. The lock wait time associated with the in-memory logging accounts for a dominant fraction of the total wait time in `dbench` and `mdtest`, more than 90% beyond 20 cores. The lock contention overhead in the in-memory logging is far more severe than the lock contention on the filesystem metadata.

Fig. 10b shows the result of the ratio of the lock wait time for in-memory logging examined against the total workload execution time. When there is a small number of cores, e.g. ten cores, the lock wait time for in-memory logging accounts for less than 5 % of the total execution time. When there are 80 cores, the lock wait time accounts for 60% of the total execution time in `dbench`. In `varmail-ptd` with 80 cores, 25% of the total execution time is spent on the lock wait time for in-memory logging.

Through this in-depth analysis, we found that the lock contention associated with `ctx_lock` and `cil_lock` is the main cause of scalability failure in the three workloads.

4 ScaleXFS

4.1 Design Overview

XFS employs sophisticated techniques to reduce the time for committing a journal transaction to the storage. This is to hide the slow flush latency of the storage device. The techniques include differential logging, concurrent journal commit, out-of-order on-disk logging, re-logging [9], etc. These works are for reducing the amount of logs written to the journal region or for eliminating the ordering dependency within and between the journal transactions. On the other hand, the in-memory logging aspect of the XFS has not received proper attention. The advancement of the low latency storage devices [4] combined with the introduction of the manycore machine that has hundreds of CPU cores introduces another dimension of the complexity in the XFS journaling design. Due to the introduction of low latency storage devices loaded with power loss protection, the importance of efficiently handling the journal commit is less emphasized. Contrarily, as we observed so far, in-memory logging has become the major bottleneck in the performance and scalability of the XFS filesystem. In this work, we focus on resolving the scalability issue in the in-memory logging of XFS.

We observe that the lock that are used to protect the global objects in in-memory logging are the main cause of the scalability failure. There are two main objects in filesystem journaling in XFS: the committed item list and log data. These are used by both in-memory logging as well as on-disk logging.

The objective of this work is to mitigate the contention on the committed item list. In XFS journaling, there can be three types of contention on the committed item list; between in-memory logging and on-disk logging, among the concurrent in-memory loggings, and among the concurrent on-disk loggings. The first is between the application thread and the journal thread. The second is among the application threads that perform the metadata operation. The third is among the journal threads, which we find to be almost non-existent. We propose three key techniques to make the in-memory logging of XFS scalable; (i) To mitigate the contention between the in-memory logging and on-disk logging, we develop *Double Committed Item List*, (ii) To mitigate the contention among the in-memory loggings, we develop *Per-core In-memory Logging* and (iii) To mitigate the contention on the global state of the committed item list, we propose *Strided Space Counting*.

4.2 Double Committed Item List

We propose *Double Committed Item List* to eliminate the contention between the on-disk logging of the journal thread and the in-memory logging of the application thread. In the Double Committed Item List, we define two committed item lists. When the application thread finds that one of the committed item lists is being exclusively used by the journal thread, the application thread inserts the log item to the other committed item list. In ScaleXFS, a metadata object maintains up to two versions of the updates, one for each committed item list. The application may attempt to modify the metadata whose log data is being committed to the disk. If this happens, ScaleXFS inserts the newly created log data to the other committed item list than the one that is subject to (and locked by) on-disk logging. With a dual committed item list, in-memory logging and on-disk logging can proceed in parallel each of which works on its own committed item list. Fig. 11 illustrates an example of this. Application threads, T_1, T_2 and T_3 are accessing the committed item list for in-memory logging. These threads hold the shared lock on this committed item list. Journal thread T_{jrnl} is accessing the other committed item list for on-disk logging. T_{jrnl} holds the exclusive lock on this committed item list.

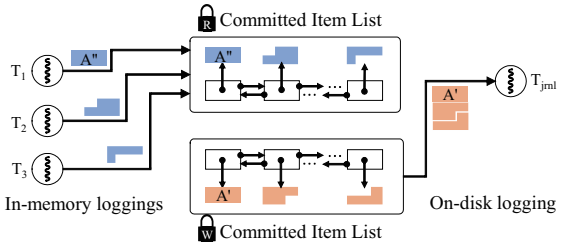


Figure 11: Double Committed Item List.

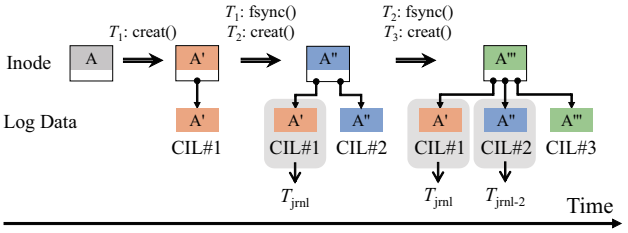


Figure 12: Multiple Committed Item Lists for in-memory logging: Triple Committed Item Lists.

Fig. 12 illustrates how the update log is inserted to the committed item list when we use three committed item lists. In practice, ScaleXFS is deliberately designed to use only *two* committed item lists in its design. In Fig. 12, there are three threads, T_1, T_2 and T_3. The three threads share a directory and update it, e.g. `creat()`. After they update the shared directory, each of them invokes `fsync()`. The first thread, T_1, updates the inode from A to A'. The log data is inserted to the committed item list 1. Then, T_1 calls `fsync()` to commit A' to the disk. While the log data A' is being committed, The second thread, T_2, updates the inode to A". Since committed item list 1 is being locked for on-disk logging, the application inserts the log data A" to the committed item list 2. Assume that T_2 calls `fsync()`. The newly arriving `fsync()` locks

the committed item list 2 that has log data A" and starts on-disk logging for the committed item list 2. Assume that third thread, T_3, attempts to update the same inode to A"'. Then, T_3 will select one of the committed item lists that are available for logging, either committed item list 1 or committed item list 3, and inserts the log for A"' to the selected committed item list. In Fig. 12, T_3 selects committed item list 3 and inserts A"' to selected one.

# cores	4	20	40	60	80	112
T_{create} (us)	42.5	42.5	68.5	91.5	122.4	192.7
T_{access} (us)	4.8	9.2	16.9	21.3	28.4	43.3
T_{create} / T_{access}	8.9×	4.6×	4.1×	4.3×	4.3×	4.4×

Table 2: Time to create journal thread (T_{create}) and for accessing the committed item list (T_{access}) under the `varmail-ptd`. HPE ProLiant DL580 Gen10, 4 socket (28 cores per socket) with 512 GB DRAM, Intel Optane 905P.

As we define the larger number of the committed item lists, the contention between the in-memory logging and the on-disk logging may be further reduced, but the overhead of maintaining the multiple lists increases. With detailed physical experiment and analysis, we find that the optimal number of the committed item lists in ScaleXFS is *two*. In Linux OS, creating a thread is a serial activity. Through physical experiment, we measure the time to create a thread (T_{create}) and the length of time interval during which the journal thread holds an exclusive lock on the committed item list (T_{access}). During T_{access}, the log data is migrated from committed item list to committing list. In on-disk logging, the time to create a journal thread is at least 4× longer than the time during which the journal thread holds the exclusive lock on the committed item list (Table 2). Technically, there can be multiple on-disk logging activities in flight in XFS. In reality, it is unlikely that two ore more journal threads compete for the exclusive lock on the committed item list. It is unnecessary to allocate more than one committed item list to mitigate the contention among the journal threads.

To prohibit the journal threads from locking both committed item lists (though it is unlikely to happen) and blocking of in-memory logging due to the lack of available committed item list, we allow at most one journal thread to lock the committed item list. In our physical experiment, we confirm that the newly arriving journal thread rarely observes any of the committed item lists being locked by on-disk logging activity.

The committed item list can be in one of the three states: Standby, Active and Blocked. In the Standby state, the committed item list is empty. In the Active state, the committed item list has the log data in it. The committed item list in the Active state can be free or locked by the shared lock for in-memory logging. The committed item list changes to the Blocked state if the journal thread requests for the exclusive lock on the committed item list. If the committed item list is available, the exclusive lock request is granted immediately. If the committed item list has been locked by the shared lock, the journaling thread waits until the shared lock is released. In both cases, the state of the committed item list changes to the Blocked state and the subsequent request for the shared lock is blocked.

ScaleXFS selects the committed item list for in-memory logging using a simple flip-flop based algorithm. The journal thread is assigned a transaction ID. The transaction ID monotonically increases with an increment of one each time the journal thread is created. The journal threads alternate between the two committed item lists for on-disk logging. The journal threads with an odd transaction ID will be using the same committed item list, e.g. CIL 1, while the journal threads with an even transaction ID use the opposite, e.g. CIL 2, for on-disk logging. Based upon the transaction ID of the current journaling thread, XFS selects the committed item list for in-memory logging.

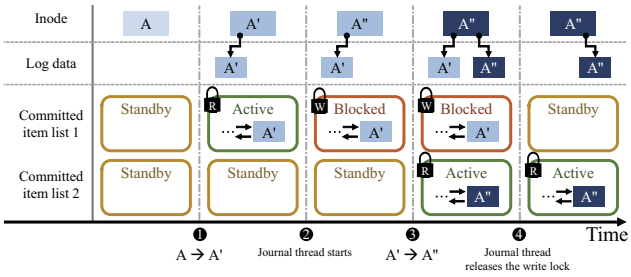


Figure 13: State of Committed item list.

Fig. 13 shows an example of the process of changing the state of the committed item list in double committed item list.

❶ **Standby** → **Active**: At the beginning, XFS creates the two committed item lists. Both of them are in the Standby state. Based upon the current transaction ID, we select the committed item list for in-memory logging and establish the shared lock. Once the shared lock is established, the state of the committed item list changes from Standby to Active state and the application thread performs in-memory logging on the selected committed item list. After the application thread finishes in-memory logging and releases the shared lock, the committed item list is still in the Active state. The subsequent application threads use the same committed item list since the current transaction ID has not changed yet.

❷ **Active** → **Blocked**: When the journal commit request arrives, the filesystem creates the journal thread increasing the transaction ID by one. Then, the journal thread establishes the exclusive lock on the committed item list and performs on-disk logging. The state of the committed item list becomes *Blocked*.

❸ **Standby** → **Active** If the application thread needs to perform in-memory logging while one of the committed item lists is in the Blocked state, the application thread chooses the other committed item list for in-memory logging. This committed item list goes into the Active state.

❹ **Blocked → Standby** The journal thread migrates all log data in the committed item list into the committing list. Once this is done, the committed item list becomes empty and the journal thread releases the lock on the committed item list. The committed item list goes into the Standby state.

We use `trylock()` [8] in establishing the shared lock on the committed item list. This is to avoid the situation where in-memory logging is blocked by the on-disk logging. If `trylock()` fails, ScaleXFS establishes the shared lock on another committed item list.

4.3 Per-core In-memory Logging

To mitigate the contention among the in-memory logging activities of the application threads, we propose *Per-core In-memory Logging*. The committed item list is organized with multiple lists on a per-core basis. With Per-core In-memory Logging, the application thread inserts the log data into the list allocated for the current CPU core on which it is being executed. Per-core in-memory logging mitigates the contention among the application threads that perform in-memory logging. In on-disk logging, the journal thread scans the per-core lists and merges them into a single list for on-disk logging. The rest of the on-disk logging procedure remains the same as when the committed item list is a single list of log data. Combined with the Double Committed Item List, each committed item list consists of a set of per-core lists (Fig. 14).

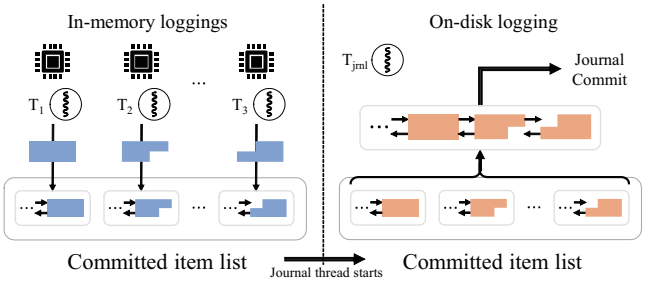


Figure 14: Per-core In-memory Logging.

There are two issues in organizing the committed item list with a set of per-core lists. First, we can preserve the order among the updates *only* within the per-core list. We cannot specify the ordering dependency among the logs in different per-core lists. Second, there can be a conflict among the per-core lists. Applications running on a different core may update the same metadata. There can be only one log data for each metadata object. The log data may need to be migrated between the per-core lists of different cores if the log data is updated by the threads running on the different cores.

We introduce *timestamp* for each log data to maintain the global order among the log data across the per-list lists. When the log data is newly created or updated, we put the timestamp

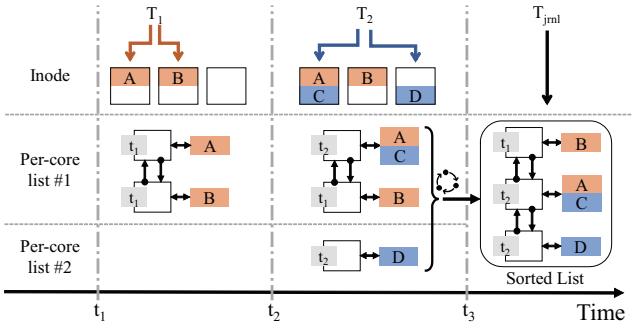


Figure 15: Ordering mechanism with timestamp: T_1 and T_2 modify the inodes at times t_1 and t_2, respectively.

at the associated log data, as illustrated by the example in Fig. 15. For the conflict, we insert the log data at the original per-core list where the preceding update exists.

At the beginning, there are two logs; A and B at per-core list #1. They are updated by thread T_1 at t_1. At time t_2, thread T_2 updates $inode_1$ to C and $inode_3$ to D. The per-core list #1 contains a log A for $inode_1$. When thread T_2 updates $inode_1$ to C, the log data in per-core list #1 is updated from A to A,C and the timestamp is also updated from t_1 to t_2. Thread T_2 creates the new log data for $inode_3$ and insert it to per-core list #2 with t_2. The updated log data remains at the original per-core list even though the thread at the different core updated it, e.g. at t_2 on per-core list #1 (Fig. 15). By pinning the updated log data to its original per-core list, we avoid modifying the per-core list across the core. In on-disk logging, the journal thread coalesces multiple per-core lists into a single list which maintains the global order of update with respect to the timestamp, e.g. at t_3 (Fig. 15).

4.4 Strided Space Counting

XFS maintains the total log data size in the committed item list. We call it a space counter. The space counter is used to estimate the size of disk space that is used to accommodate the logs in the committed item list. If the free space in the journal region is less than the space counter, XFS checkpoints the journal logs in the disk to make a room for the incoming logs. Each in-memory logging updates the space counter. The space counter is protected by spinlock (`cil_lock`).

To mitigate the contention on the space counter, we develop *Strided Space Counter*. Strided Space Counter is a variant of the sloppy counter [17] that is tailored to estimate the available space in the journal region of the disk. Strided space counter consists of the per-core space counters, the per-core strided space counter, the global space counter, and the stride length. The per-core space counter and the per-core strided space counter are initially set to 0. When the application performs in-memory logging, it increases the space counter by the size of the log data. If the local (per-core) space counter exceeds

the local strided space counter, the local strided space counter is folded to the global strided space counter. After it is folded to the global space counter, the local strided space counter increases by the length of stride. The thread increases the local counter when it needs to increase the space requirement. When the thread needs to check the space availability, it reads a global counter.

Strided space counter shares much of its behavior with sloppy counter but is not entirely the same. To avoid confusion, we call our counter Strided Space Counter. Sloppy counter estimates the *minimum* value of the sum of the local counters and is used to check if the reference counter value is non-zero [17]. Strided Space Counter estimates the *maximum* value of the sum of the local counters and is used to check if sufficient amount of space is reserved at the journal region. In sloppy counter, if the local counter exceeds the threshold value, the local counter is folded to the global counter. In Strided Space Counter, if updating the local counter makes the sum of the local counters greater than the global counter, we increase the value of the global counter by the stride length before we increase the local counter.

Time	Counter (L) / Strided Counter (S)								G
	L_1	S_1	L_2	S_2	L_3	S_3	L_4	S_4	
t_0	0	0	0	0	0	0	0	0	0
t_1	3	5	0	0	0	0	1	5	10
t_2	4	5	0	0	6	10	2	5	20
t_3	5	5	0	0	7	10	3	5	20
t_4	6	10	4	5	8	10	3	5	30

Figure 16: Stride Space Counting, `stride length`: 5.

Fig. 16 illustrates how strided space counting works. At t_1, the local counter at core 1 is changed to 3. The local strided counter of core 1 was 0. Now, it is updated to 5 by the stride length and it is folded to the global strided space counter. Simultaneously at t_1, the local counter at core 4 is change to 1. The local strided counter of core 4 is updated from 0 to 5 and it is folded to the global strided space counter. The global space counter G becomes 10 at time t_1. At time t_2, the local counter L_3 changes from 0 to 6. The local strided space counter becomes 10 and the global strided space counter becomes 20.

The stride length should be large enough to reduce the contention on the global counter. But, if the stride length is too large, the global space counter exceeds the required disk space for journaling too far and causes unnecessary checkpoint.

5 Evaluation

We examine the manycore scalability of ScaleXFS using a 112 core server (HPE ProLiant DL580, 28 core/socket, 4 sockets, Intel Xeon Platinum 8276) with 512 GByte DRAM. The CentOS 7.4 (kernel is 5.8.5) and Intel Optane 905p NVMe SSD were used. In the experiment, the number of active cores were varied and evenly distributed among the sockets. We compare three filesystems; EXT4, original XFS, and ScaleXFS. For ScaleXFS, we use three different settings; ScaleXFS$_{\mathrm{D}}$, ScaleXFS$_{\mathrm{DP}}$ and ScaleXFS$_{\mathrm{DPS}}$. Each subscript in ScaleXFS refers to a different feature. D, P and S refers to Double committed item list, Per-core in-memory logging, and Strided space counting, respectively.

5.1 Lock Contention

The goal of ScaleXFS is to eliminate the lock contention of shared journaling structures. We first show that ScaleXFS significantly reduces the lock contention of the overall system. `Lockstat` [5] was used to measure the lock contention to examine how it is mitigated in our work. For three workloads, the total lock wait time, the lock wait time for `cil_lock` and `ctx_lock-R` (shared lock of `ctx_lock`). were examined, and the results are shown in Fig. 17.

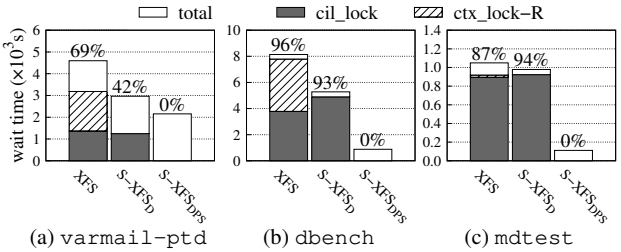


Figure 17: Total lock wait time of `cil_lock`, `ctx_lock-R` and all of the locks. Each number above each bar is the proportion of the sum of `cil_lock` and `ctx_lock-R` out of the total wait time of all locks, 112 cores.

Recall that `ctx_lock` protects the committed item list and the associated log data from the conflict between in-memory logging and on-disk logging. `cil_lock` protects them from conflicts among the multiple in-memory loggings. In XFS, `cil_lock` and `ctx_lock-R` combined account for as low as 69% (Fig. 17a), as high as 96% (Fig. 17b) of the total lock wait time. We do not observe any lock wait time for `ctx_lock-R` in `mdtest`. This is because `mdtest` does not issue any `fsync()` and therefore in-memory logging does not wait for the shared lock (`ctx_lock-R`).

It is shown that the double committed item list successfully eliminates the contention between in-memory logging and on-disk logging. ScaleXFS$_{\mathrm{D}}$ does not have any contention on `ctx_lock-R`, since in-memory logging and on-disk logging do not need to compete with each other anymore. By eliminating the overhead on `ctx_lock-R`, the total lock wait time also decreases by 36% and 35% on `varmail-ptd` and `dbench`, respectively. In ScaleXFS$_{\mathrm{DPS}}$, Per-core basis committed item list enables parallel in-memory logging, leading to the elimi-

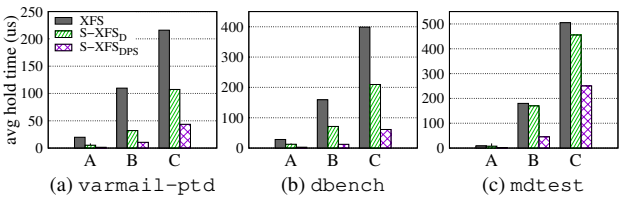


Figure 18: Average hold time of locks protecting metadata objects. A, B and C refer to `xfs_nondir_ilock_class-W`, `xfs_dir_ilock_class-W`, `i_mutex_dir_key` each. 112 cores.

nation of the contention on `cil_lock`. The total wait time of all locks decreases by 53%, 90%, and 90% for `varmail-ptd`, `dbench` and `mdtest`, respectively, in ScaleXFS$_{DPS}$ (Fig. 17).

As a result of reducing the in-memory logging latency, the ScaleXFS reduces the total time to hold the exclusive lock on the metadata object. This is because the filesystem needs to hold the exclusive lock on the metadata object until the metadata operation completes, i.e. until the in-memory logging completes. The lock wait times of three widely used locks, `xfs_nondir_ilock_class-W`, `xfs_dir_ilock_class-W` and `i_mutex_dir_key`, were examined under XFS, ScaleXFS$_D$ and ScaleXFS$_{DPS}$ filesystems. Three benchmark workloads, `varmail-ptd`, `dbench` and `mdtest`, were examined and their average hold times were measured at 112 cores. Fig. 18 illustrates the results. In `varmail-ptd`, ScaleXFS$_{DPS}$ reduces the hold time of each lock by 92%, 90%, and 80% against the original XFS. A similar improvement in the other workloads was also observed. Eliminating the lock contention associated with the in-memory logging and the on-disk logging, i.e. contention on `cil_lock, ctx_lock-R`, the entire lock hold time on the metadata update decreases to as much as 1/10th.

5.2 Latency of creat(), unlink() and fsync()

To show how the operation latencies are effectively reduced by eliminating lock contentions, we evaluated the average latencies of three filesystem operations; `creat()`, `unlink()`, and `fsync()` on `varmail-ptd` and `dbench` under a varying number of cores. In this experiment, we also include the performance result from EXT4.

ScaleXFS reduces the latency of metadata operations. The reduction becomes more pronounced as the number of cores increases. In `varmail-ptd`, `creat()` and `unlink()` latencies in ScaleXFS$_{DPS}$ are 1/5 and 1/6 of those in XFS at 112 cores, respectively (Fig. 19a and Fig. 19b). In `dbench`, `creat()` and `unlink()` latencies in ScaleXFS$_{DPS}$ are 1/4 and 1/6 of those in XFS, respectively (Fig. 19d and Fig. 19e). ScaleXFS$_{DPS}$ renders more scalable behavior than the XFS. Under `varmail-ptd` workload, when the number of cores increases from 4 to 112, in XFS and ScaleXFS$_{DPS}$, the latency of `unlink()` increases by 20× and by 3.5×, respectively.

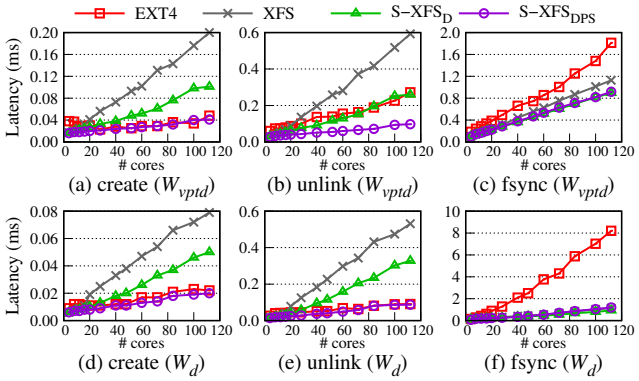


Figure 19: Average latencies of `creat()`, `unlink()`, `fsync()` at `varmail-ptd` (W_{vptd}) and `dbench` (W_d).

Fig. 19 shows that XFS and EXT4 have their own performance edges. XFS exhibits better performance in filesystem journaling than EXT4. EXT4 renders better performance in metadata operation than XFS. ScaleXFS$_{DPS}$ makes a significant improvement in the metadata operation against stock XFS. ScaleXFS$_{DPS}$ exhibits better filesystem journaling performance than XFS (Fig. 19c) and better performance in metadata operation than EXT4 (Fig. 19a and Fig. 19b).

5.3 Benchmark performance

`varmail-ptd`. Fig. 20a illustrates the result of `varmail-ptd`. ScaleXFS$_{DPS}$ outperforms XFS and EXT4 by 1.5×, 1.9× at 112 cores, respectively. ScaleXFS$_{DPS}$ scales well while XFS performance saturates beyond 20 cores. The performance difference between ScaleXFS$_D$ and ScaleXFS$_{DPS}$ is not significant in `varmail-ptd` because the on-disk logging accounts for a substantial fraction of the entire workloads, and subsequently contention among the in-memory loggings is not as severe as in the other workloads. Therefore, the benefit of using per-core in-memory logging is not significant.

`dbench`. Fig. 20b illustrates the result of `dbench`. ScaleXFS$_{DPS}$ shows 1.3× and 2.2× performance against ScaleXFS$_D$ and XFS, at 112 cores, respectively. ScaleXFS$_{DPS}$ also outperforms EXT4 by 4.5× at 112 cores. In `dbench`, per-core in-memory logging and Strided Space Counting becomes more effective than in `varmail-ptd`. In `dbench`, in-memory logging operation accounts for larger fraction of operation than in `varmail-ptd`. In `varmail-ptd`, on-disk logging (`fsync()`) accounts for 15.4% of the total number of system calls. In `dbench`, on-disk logging (`fsync()`) accounts for only 1.2% of the total number of system calls. (Table 1). In `dbench`, the contention among the in-memory logging operations becomes more intense than the contention between the in-memory logging and on-disk logging. The techniques to mitigate the contention among the in-memory loggings becomes far more effective in `dbench` than in

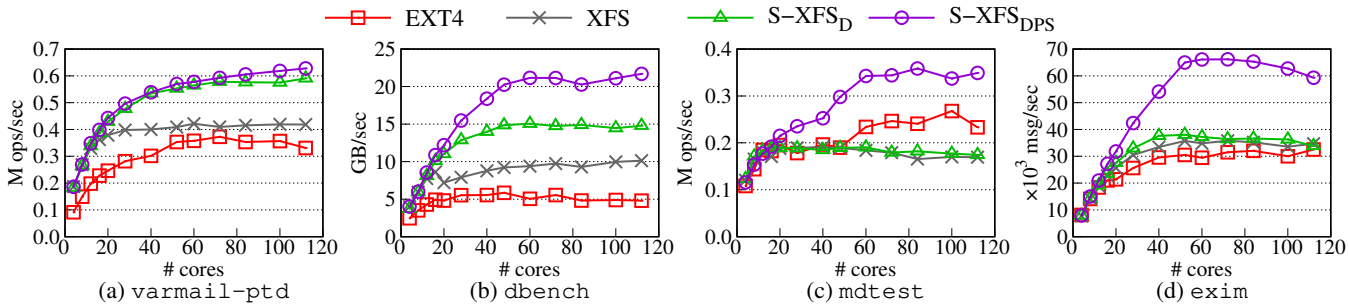


Figure 20: Throughput of `varmail-ptd`, `dbench`, `mdtest` and `exim`.

`varmail-ptd`. The performance gain of ScaleXFS$_{DPS}$ against ScaleXFS$_D$ becomes more substantial under `dbench` than under `varmail-ptd`.

The benefit of double committed item list is substantial in both `varmail-ptd` and `dbench` workload. ScaleXFS$_D$ renders 1.4× performance in `varmail-ptd` and 1.7× performance in `dbench` than XFS in 112 core server. In original XFS, on-disk logging (`fsync()`) blocks the in-memory logging. With double committed item list, in-memory logging and on-disk logging can proceed in parallel since each of them can work on its own committed item list. As a result, ScaleXFS$_D$ yields significant performance advantage against stock XFS in `dbench` and `varmail-ptd`.

`mdtest`. Fig. 20c illustrates the result of `mdtest`. ScaleXFS$_{DPS}$ outperforms the original XFS by 2.1× ScaleXFS$_{DPS}$ scales well till 60 cores while XFS saturates beyond 10 cores. ScaleXFS$_{DPS}$ adopts scalable in-memory logging and outperforms EXT4 by 1.5× `mdtest` is metadata intensive workload without `fsync()`. The performance benefit of adopting per-core in-memory logging is substantial in `mdtest`.

`exim`. Fig. 20c illustrates the result of `exim`. `Exim` [3] is a mail server, creating, renaming, and deleting small files repeatedly. We disable per-message `fsync()` in `exim` as in FxMark [37]. `Exim` does not issue `fsync()` similar to `mdtest` and the on-disk logging does not occur frequently. As a result, the effect of a double committed item list that is for removing the contention between the in-memory logging and the on-disk logging becomes less significant. The performance gap between ScaleXFS$_D$ and XFS is not remarkable. Contrarily, the contention among the in-memory loggings becomes more intense and the effect of per-core in-memory logging becomes significant. ScaleXFS$_{DPS}$ outperforms ScaleXFS$_D$ by 1.5×. ScaleXFS$_{DPS}$ outperforms XFS and EXT4 by as much as 1.9× and 2.2×, respectively. We observe that the throughput of ScaleXFS$_{DPS}$ slightly drops after 60 cores. This is caused by the overhead of spinlock in the VFS layer. The number of spool directories in `exim` is configured to 62. As ScaleXFS significantly increases the throughput, the overhead on spool directories becomes more substantial [31].

`FxMark`. Fig. 21 shows the result of `FxMark`. To analyze the performance improvement of the metadata operation, we examine the performance of `unlink()` under different sharing levels of FxMark [37].

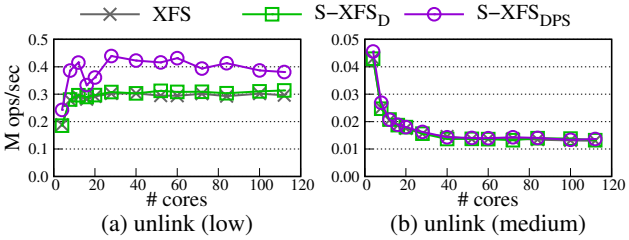


Figure 21: Throughput of FxMark's `unlink()` operation on low and medium sharing levels.

In the low sharing level, the processes perform the metadata operations in their private directory. ScaleXFS$_{DPS}$ exhibits superior performance against ScaleXFS$_D$ and XFS (Fig. 21a). Per-core in-memory logging is effective in eliminating the contention among the in-memory loggings.

In the medium sharing level, the processes perform the metadata operations in a shared directory. The contention to hold the exclusive lock on the shared metadata neutralizes the effectiveness of per-core in-memory logging, ScaleXFS$_{DPS}$ does not render any performance improvement against XFS (Fig. 21b).

5.4 Strided Space Counting

We measure the average latency and tail latency (@95%) of in-memory logging, and the throughput of `mdtest` under varying the stride lengths. We vary the stride length from 0 Byte to 8 KByte. When the stride length is 0 Byte, it represents the performance of ScaleXFS$_{DP}$. Stride space counting reduces the average latency as much as by 7.3% (Fig. 22a). The tail latency of in-memory logging is unaffected by strided space counting (Fig. 22b). Strided space counting improves the throughput by up to 13% compared against ScaleXFS$_{DP}$ (Fig. 22c). Dominant operations of `mdtest` workload is in-

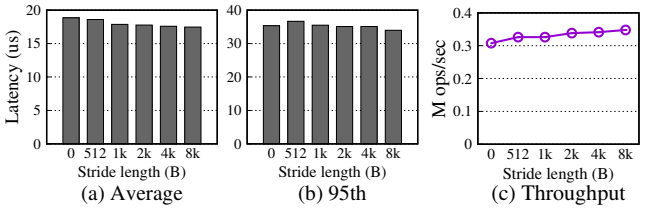


Figure 22: Effect of stride length: In-memory logging latency and throughput in `mdtest`, `creat()`, 112 cores.

memory logging. In manycore environment, large number of in-memory logging operations can proceed in parallel rendering intense contention on the shared variable such as locks and space counter. Via introducing the per-core space counter, ScaleXFS effectively mitigates the contention on the global variable, space counter substantially.

Larger stride length may render overestimating the size of the required journal region on the disk. We found that 8 KByte stride length yields 14 KByte of space waste for a 2 GByte journal region. We believe that space waste of 14 Kbyte for 8 KByte stride length is negligible. Given this, we set the default stride length to 8 KByte.

6 Related Work

A few works have examined the performance and scalability behavior of the XFS [13, 37]; however, they do not provide a detailed analysis on the observed behavior or offer a solution to address it. Other works explain the internals of XFS [10] and compare the performance of multiple filesystems [12, 23]. Few of these works perform an in-depth study on the performance bottleneck of the XFS filesystem and propose a solution to address the observed issues.

EXT4 suffers from sub-optimal performance primarily due to its page granularity physical logging that uses the original page cache entry for journal commit [48] and due to the serial journal commit [37]. A number of works proposed using multiple running transactions [27, 40] and/or multiple committing transactions [26, 27, 34, 40, 48] to address the performance and the scalability of the filesystem journaling. Son et al. [43] allows multiple processes to insert the log data to the running transaction concurrently via a lock-free mechanism. ScaleFS [15] decouples the in-memory filesystem from the on-disk filesystem. The in-memory filesystem adopts highly concurrent data structures with the per-core operation log and the on-disk filesystem merges the operation log and synchronize it to the disk.

Min et al. [37] found that the cache line bounce of the global lock causes a performance collapse in the manycore system. A number of works propose a scalable lock primitive. PRW lock [32] and RPS [31] adopt the per-core indicators to reduce the contention among readers. To improve the write lock latency, these works update the global value when the IPIs are transmitted [32] or when the per-core flags are set [31] by the writer. RPS efficiently checks the per-core indicators and flags, by leveraging the CPU scheduler [31]. BRAVO [21] uses the global hash table to reduce the memory footprint.

A number of works improve the scalability of the read operation. Refcache [18] adopts per-core reference delta caches and merges the changes between epoch into a single operation. PayGo [25] adopts the per-core hash technique and anchor counter for scalable counter. Lodic [41] achieves file block scalability by implementing a local counter on the page table entry considering the popularity of a file.

7 Conclusion and Future Work

In this work, we address the XFS filesystem scalability. We identify the prime cause for scalability failure in XFS journaling: the contention among the application threads and the journal thread to access the global list of log data. To address this issue, we propose Dual Committed Item List, Per-Core In-memory Logging and Strided Space Counting. Our experimental results confirm that ScaleXFS resolves the scalability bottleneck of XFS under various workloads. ScaleXFS reveals two new bottlenecks that were not visible before. The first one is the host-side overhead of handling on-disk logging. XFS allows that multiple committing transactions are made durable at the disk in out-of-order manner. However, to ensure the filesystem integrity, the XFS filesystem at the host-side finishes the journal commit in the order in which the transaction commit requests are made; XFS finishes the journal commit of a transaction only when all its preceding journal commit finishes. We find that this in-order completion for journal commit mechanism becomes a scalability bottleneck when we resolve the lock contention in in-memory logging. The second one is the memory copy overhead of differential logging. As the journaling becomes efficient, the memory copy overhead associated with creating the differential copy of the updated metadata accounts for a relatively larger fraction of journaling overhead. Currently, XFS adopts a crude coarse grain differential logging and it leaves substantial room for improvement. We like to address these two issues in our future effort.

8 Acknowledgements

We would like to thank our shepherd, George Amvrosiadis, for helping shape the final version of this paper. We are also grateful to the anonymous reviewers for their valuable feedback that have improved this paper. This work was in part supported by IITP of Korea (No. IITP-2018-0-00549), NRF of Korea (No. NRF-2020R1A2C3008525), and SNU-SK Hynix Solution Research Center (S3RC) (No. MOUS002S).

References

[1] http://lkml.iu.edu/hypermail/linux/kernel/1603.2/04523.html.

[2] Chapter 3. the xfs file system. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/ch-xfs.

[3] Exim. https://www.exim.org.

[4] Intel. breakthrough performance for demanding storage workloads. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf.

[5] lockstat. https://www.kernel.org/doc/Documentation/locking/lockstat.txt.

[6] Red hat continues to lead the linux server market. https://www.redhat.com/en/blog/red-hat-continues-lead-linux-server-market.

[7] Red hat enterprise linux powers the globe's stock exchanges. https://www.redhat.com/en/blog/red-hat-enterprise-linux-powers-the-globes-stock-exchanges.

[8] trylock function. https://www.kernel.org/doc/htmldocs/kernel-locking/trylock-functions.html.

[9] Xfs delayed logging design. In *Filesystems in the Linux kernel*. https://www.kernel.org/doc/html/latest/filesystems/xfs-delayed-logging-design.html.

[10] SGI XFS Algorithms & Data Structures, 3rd Edition. 2006. http://ftp.ntu.edu.tw/linux/utils/fs/xfs/docs/xfs_filesystem_structure.pdf.

[11] Dbench. 2008. https://dbench.samba.org/.

[12] Mohd Bazli Ab Karim, Jing-Yuan Luke, Ming-Tat Wong, Pek-Yin Sian, and Ong Hong. Ext4, xfs, btrfs and zfs linux file systems on rados block devices (rbd): I/o performance, flexibility and ease of use comparisons. In *Proc. of ICOS*, pages 18–23. IEEE, 2016.

[13] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proc. of ACM SOSP*, pages 353–369, 2019.

[14] Steve Best. Jfs overview. how the journaled file system cuts system restart times to the quik. 2000. http://jfs.sourceforge.net/project/pub/jfs.pdf.

[15] Srivatsa S Bhat, Rasha Eqbal, Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proc. of ACM SOSP*, pages 69–86, 2017.

[16] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proc. of the 44th annual design automation conference*, pages 746–749, 2007.

[17] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Tappan Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *Proc. of USENIX OSDI*, volume 10, pages 86–93, 2010.

[18] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proc. of ACM EUROSYS*, pages 211–224, 2013.

[19] J Corbet. Xfs: the filesystem of the future?, 2012.

[20] Jonathan Corbet. Barriers and journaling filesystems. https://lwn.net/Articles/283161/.

[21] Dave Dice and Alex Kogan. Bravo—biased locking for reader-writer locks. In *Proc. of USENIX ATC*, pages 315–328, 2019.

[22] C. HELLWIG. Patchwork block: update documentation for req_flush / req_fua. https://patchwork.kernel.org/patch/134161/.

[23] Christoph Hellwig. Xfs: the big storage file system for linux. *; login:: the magazine of USENIX & SAGE*, 34(5):10–18, 2009.

[24] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. Txfs: Leveraging file-system crash consistency to provide acid transactions. *ACM TOS*, 15(2):1–20, 2019.

[25] Seokyong Jung, Jongbin Kim, Minsoo Ryu, Sooyong Kang, and Hyungsoo Jung. Pay migration tax to homeland: anchor-based scalable reference counting for multicores. In *Proc. of USENIX FAST*, pages 79–91, 2019.

[26] 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 USENIX ATC*, pages 249–261, 2015.

[27] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. Z-journal: Scalable per-core journaling. In *Proc. of USENIX ATC*, pages 893–906, 2021.

[28] Yi-Reun Kim, Kyu-Young Whang, and Il-Yeol Song. Page-differential logging: an efficient and dbms-independent approach for storing data into flash memory. In *Proc. of ACM SIGMOD*, pages 363–374, 2010.

[29] George Kurian, Jason E Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C Kimerling, and Anant Agarwal. Atac: A 1000-core cache-coherent processor with on-chip optical network. In *Proc. of 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 477–488. IEEE, 2010.

[30] Juchang Lee, Kihong Kim, and Sang Kyun Cha. Differential logging: A commutative and associative logging scheme for highly parallel main memory database. In *Proc. of ICDE*, pages 173–182. IEEE, 2001.

[31] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A multicore-accelerated file system for flash storage. In *Proc. of USENIX ATC*, pages 877–891, 2021.

[32] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proc. of USENIX ATC*, pages 219–230, 2014.

[33] LLNL. mdtest. `https://sourceforge.net/projects/mdtest/`.

[34] 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 USENIX OSDI*, pages 81–96, 2014.

[35] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proc. of Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

[36] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. A fast file system for unix. *Proc. of ACM TOCS*, 2(3):181–197, 1984.

[37] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proc. of USENIX ATC*, pages 71–85, 2016.

[38] Gibson Ming-Dar. Introduction of power loss protection function on ssd. 2019. `https://www.embeddedcomputing.com/technology/storage/introduction-of-power-loss-protection-function-on-ssd`.

[39] Timothy Prickett Morgan. Flashtec nvram does 15 million iops at sub-microsecond latency. 2014. `https://www.enterpriseai.news/2014/08/06/flashtec-nvram-15-million-iops-sub-microsecond-latency/`.

[40] Daejun Park and Dongkun Shin. ijournaling: Fine-grained journaling for improving the latency of fsync system call. In *Proc. of USENIX ATC)*, pages 787–798, 2017.

[41] Jeoungahn Park, Taeho Hwang, Jongmoo Choi, Changwoo Min, and Youjip Won. Lodic: Logical distributed counting for scalable file access. In *Proc. of USENIX ATC*, pages 907–921, 2021.

[42] Silicon Graphics, Inc. IRIX Advanced Site and Server Administration Guide, Chapter 8. `https://irix7.com/techpubs/007-0603-100.pdf`.

[43] Yongseok Son, Sunggon Kim, Heon Y Yeom, and Hyuck Han. High-performance transaction processing in journaling file systems. In *Proc. of USENIX FAST*, pages 227–240, 2018.

[44] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *Proc. of USENIX ATC*, volume 15, 1996.

[45] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[46] Kingston Technology. A closer look at ssd power loss protection. 2019. `https://www.kingston.com/en/solutions/servers-data-centers/ssd-power-loss-protection`.

[47] Stephen C Tweedie et al. Journaling the linux ext2fs filesystem. In *Proc. of Annual Linux Expo*. Durham, North Carolina, 1998.

[48] 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 USENIX FAST*, pages 211–226, 2018.

exF2FS: Transaction Support in Log-Structured Filesystem

Joontaek Oh Sion Ji Yongjin Kim Youjip Won
Department of Electrical Engineering, KAIST

Abstract

In this work, we present exF2FS, a transactional log-structured filesystem. The proposed filesystem consists of three key components: Membership-Oriented Transaction, Stealing-Enabled Transaction, and Shadow Garbage Collection. Membership-Oriented Transaction allows the transaction to span multiple files where the application can explicitly specify the files associated with a transaction. Stealing-Enabled Transaction allows the application to execute the transaction with a small amount of memory and to encapsulate many updates, e.g., hundreds of files with tens of GBs in total size, with a single transaction. Shadow Garbage Collection allows the log-structured filesystem to perform garbage collection without affecting the failure-atomicity of ongoing transactions. The transaction support in exF2FS is carefully trimmed to meet the critical needs of the application while minimizing the code complexity and avoiding any performance side effects. With exF2FS, SQLite multi-file transaction throughput increases by 24× against the multi-file transaction of stock SQLite. RocksDB throughput increases by 87% when it implements the compaction as a filesystem transaction.

1 Introduction

Modern applications strive to protect their data in a crash-consistent manner which is often split over multiple file abstractions. In the absence of proper transaction support from the underlying filesystem, the application employs complicated protocols to ensure the transactional updates that span multiple files, yielding long sequence of writes and `fsync()`'s. Text editors, such as `vim` and `emacs`, use atomic `rename()` to save the updated file atomically [1]. For a transaction that updates the multiple database files, the library-based embedded DBMS, SQLite, maintains the separate journal file for each database file [69], yielding excessive `fdatasync()` calls and a large write amplification [31]. Compaction operation of the modern LSM-based key-value store, such as RocksDB [22], maintains the state of the merge-sort at a separate journal file known as the *manifest file*. For the failure-atomicity of the compaction operation, the key-value storage engine flushes the output files separately and also flushes the global state of the compaction to the manifest file. With the transaction support from the filesystem, the application can replace the multiple `fsync()`'s for each output files and the manifest file with a single filesystem transaction, rendering higher performance by eliminating redundant IO's.

Despite the clear benefits of supporting transactions, it remains a challenge for the operating system and filesystem. To successfully deploy the transaction enabled system, the right balance must be found among the four requirements: easy to use, code complexity, degree of ACID support and performance. Unfortunately, achieving one of these is often at the cost of another. The system level supports for transaction can largely be categorized into four: native operating system support [57, 61, 61, 75], kernel level filesystem [14, 26, 27, 46, 55, 66, 72, 78], user level filesystem [24, 48, 54] and transactional block device [12, 28, 32, 52, 58, 65]. Supporting transaction as the first-class citizen of the operating system is ideal; however, it requires substantial change in the operating system. Transaction support from the user level filesystem exploits the user level DBMS to provide full ACID transaction [24, 48, 54]. ACID support comes at the cost of the performance. The transaction support from the kernel level filesystem can further be categorized with respect to the degree of ACID support: full ACID semantics [27, 66], ACD without isolation support [55, 72] or even AC without isolation and durability support [34]. An F2FS transaction [34] supports only the atomicity, neither isolation nor durability. The transaction in F2FS cannot span multiple files. Ironically, despite its barest minimum support for the transaction, F2FS is the only filesystem that successfully deploys its transaction support to the public. F2FS's transaction support has a specific target application: SQLite. With atomic write of F2FS, SQLite can implement the transaction without the rollback journal file and can eliminate the excessive flush overhead [31, 64].

In this work, we revisit the issue of providing the filesystem-level transaction support. In particular, we focus the domain of interests to the log-structured filesystem. Most of the preceding works on the transactional filesystems use the journaling filesystem as a baseline filesystem [27, 66, 72]. These works exploit the journaling layer of the filesystem to provide the transaction capability. F2FS, the log-structured filesystem designed for flash storage, recently gained wide popularity on smartphone platforms [56] and is beginning to expand into cloud platforms [6]. Few works have dealt with the transaction support in the log-structured filesystem. Seltzer et al. [62] is the nearest effort; however, their work is limited in terms of the transaction support. Their study does not support multi-

file transaction, stealing in the transaction, nor the conflict handling between a transaction and the garbage collection.

In this study, we present transaction support in the log-structured filesystem with three design objectives; (i) the transaction should be able to span multiple files, including the directory, (ii) the transaction should be able to handle large amounts of updates and (iii) the transaction should not be affected by the execution of garbage collection. Each of these requirements looks plain and essential from the application's point of view. Unfortunately, developing the transactional log-structured filesystem which satisfies these simple and plain requirements is a non-trivial exercise which calls for substantial changes in the underlying filesystem from the aspect of design as well as implementation; developing a new transaction model, redesigning the filesystem's page reclamation procedure and redesigning the garbage collection procedure. We find that few modern transactional filesystems address any of these essential requirements in its transaction management with sufficient maturity. To allow the transaction to span multiple files, we develop *Membership-Oriented Transaction Model*. To allow the transaction to handle large size transactions which may consist of hundreds of files with tens of GBs of data, we develop *Stealing* for the filesystem transaction. To prohibit the garbage collection from interfering with the ongoing transaction, we develop *Shadow Garbage Collection*. The main contributions of this work are as follows.

- **Membership-Oriented Transaction**. In Membership-Oriented Transaction, the filesystem maintains a kernel object, *Transaction File Group* that specifies the set of files, including directories, associated with the transaction. With Membership-Oriented Transaction, the application can explicitly specify the files that are subject to the transaction.
- **Stealing**. We allow dirty pages of uncommitted transactions to be evicted and yet guarantee the atomicity of the transaction. We develop *Delayed Invalidation* and *Relocation Record* to realize Stealing in the filesystem transaction. Delayed Invalidation prohibits the old disk locations of evicted pages from being garbage-collected until the transaction commits. Relocation Record maintains undo and redo information to abort and commit evicted pages, respectively.
- **Shadow Garbage Collection**. We develop *Shadow Garbage Collection* to prohibit the garbage collection module from making the dirty page of the uncommitted transaction prematurely durable and recoverable. Shadow Garbage Collection allows the filesystem to perform garbage collection transparently to the ongoing transactions.

We implement these features in F2FS. We call the newly developed filesystem extended F2FS (*exF2FS*). exF2FS improves the SQLite performance by 24× against stock SQLite and reduces the write volume to 1/6 compared to the PERSIST journal mode of SQLite. It improves RocksDB performance by 87% in the YCSB workload-A [13]. Special care has been taken not to change any on-disk structure of the existing F2FS so that exF2FS can mount the existing F2FS partition.

2 Background and Motivation

2.1 Multi-file Transaction

Multi-file transaction is an essential part of the modern software. The followings are a few examples of multi-file transaction method currently being used.

Maintaining the browsing history in the web browser. The Chrome browser maintains user browsing activity, such as visited `URL`'s, the list of downloaded files, an access history for each `URL` and the list of the most frequently visited `URL`'s. Chrome maintains each of these in a separate file and updates these files in failure-atomic fashion. For failure-atomicity, Chrome uses SQLite in updating these files [50] which renders excessive IO. The inefficiency of SQLite transactions will be explained later in this study.

Compaction in LSM-based key-value Store. Compaction is a process of merge-sorting several SSTables with overlapping intervals into a sequence of the output files with non-overlapping intervals [21]. The failure-atomicity of the compaction operation invokes `fsync()`'s for each output file and the parent directory and flushes the global state of the transaction to a special file called the *manifest file* [5, 23, 33]. In "load" workload of YCSB [13], a single compaction of RocksDB can create as many as 198 output files (over 200 `fsync()`'s) for a total of 13.3 GB.

Software Installation. Updating or installing a new software involves downloading and modifying hundreds of files and updating the associated directory in a failure-atomic manner. The partial completion of installation or update often leads to an unstable system [15, 42, 45, 73].

Mail client. `MAILDIR` IMAP format maintains the mailbox and the message as a directory and a file in the directory, respectively [2, 16, 20]. The email client updates the message files and the associated directory in transactional fashion. In the absence of transaction support from the underlying filesystem, mail clients use the expensive atomic rename to manage the mailbox and the message in transactional fashion [9, 70].

2.2 Multi-file Transaction and SQLite

SQLite is serverless embedded DBMS widely used in various applications: mobile applications such as Android Mail and Facebook App, desktop applications such as Gmail and Apple iWork [25, 27] and distributed filesystems such as Lustre [8] and Ceph [74]. These applications use SQLite to persistently manage the updates on the multiple files in failure-atomic fashion. To understand how the SQLite can benefit from the transaction support of the underlying filesystem, we instrument the IO behavior of the SQLite's multi-file transaction.

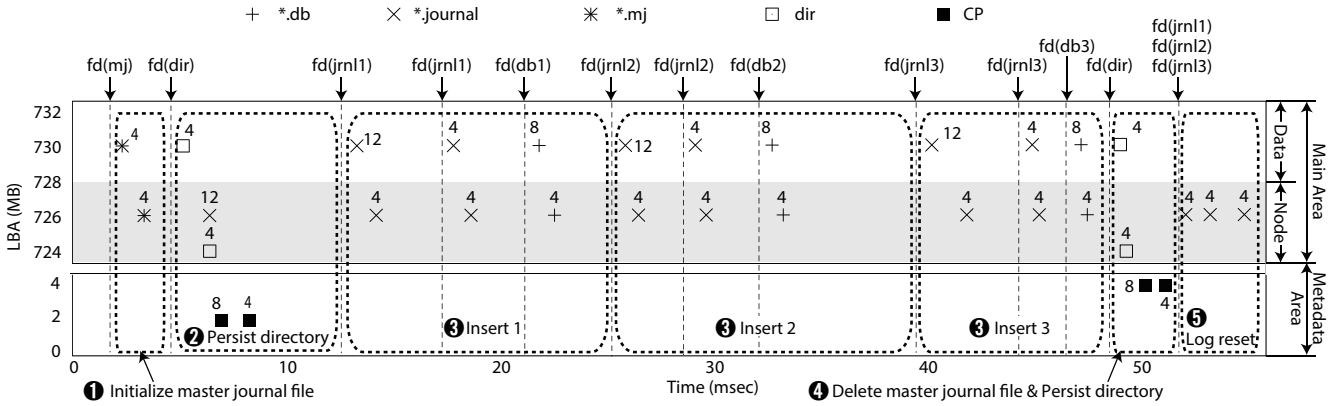


Figure 1: A multi-file transaction with three `insert()`'s in F2FS. Record size = 100 Byte, PERSIST mode. The number in each mark represents the number of KB written, Device: Samsung 850 PRO, `fd`: `fdatasync()`, `mj`: master journal file, `dir`: parent directory, `jrnl`: journal file, `db`: database file, `cp`: checkpoint

While the application can become simpler when using SQLite to persistently manage the data, it suffers from significant write amplification and excessive flush due to the page granularity physical logging and the file-backed journaling of SQLite [31, 64]. A single `insert()` of SQLite incurs five `fdatasync()`'s with 40 KB of `write()` [76]. A few studies have been dedicated to improving the extreme IO inefficiency of SQLite transaction [27, 31, 34, 36, 40, 53]. All these efforts are limited to improve the IO overhead in the transaction with a single database file.

SQLite constructs the multi-file transaction as a collection of the single file transactions and a few flushes to record the global state of the multi-file transaction at the *master journal file*. SQLite implements the multi-file transaction in the four steps listed below. Step 1 and Step 3 are for updating the master journal file. Step 2 and Step 4 are for executing the series of the single file transactions. Fig 1 shows how each of these steps is associated with the IO behavior through the physical experiment. Here, a transaction consists of three inserts to three different database files.

1. **Initializing the master journal file.** SQLite records the name of the journal files in the master journal file. Then, it flushes the master journal file (❶ in Fig. 1) and the updated directory to the disk (❷ in Fig. 1).

2. **Logging and Database Updates.** SQLite logs the undo records at the journal files and updates the database files. Each file is updated in the same way as in the single database transaction (❸ in Fig. 1). There are three ❸'s in Fig. 1 each of which corresponds to a single `insert()`.

3. **Deleting the master journal file.** As a mark of successful commit, SQLite deletes the master journal file and makes the associated directory durable (❹ in Fig. 1).

4. **Reset Logs.** SQLite resets the journal files and flushes them (❺ in Fig. 1).

In the Fig. 1, the X-axis and Y-axis denote the time and LBA, respectively. Here, we explicitly specify three regions of F2FS: the metadata area, data region of the main area, and node region of the main area. When SQLite flushes the dirty file block through `fdatasync()`, the underlying F2FS flushes not only data blocks but also the associated node block to the data region and the node region, respectively. In (❶), flushing the master journal file (`fd(mj)`) renders two separate 4 KB IO's to the disk: one for flushing the data block and the other for flushing the node block. The data block and the associated node block need to be made durable in order for guaranteeing the integrity of the filesystem. Each `insert()` has three `fdatasync()`'s (❸); the first and the second `fdatasync()` are for flushing the rollback journal file. The third one is for flushing the database file. In (❹), SQLite deletes the master journal file and persists the parent directory. When unlinking the master journal file becomes durable, the transaction is committed. In (❺), SQLite resets the rollback journal files of the transaction.

As in Fig. 1, the IO overhead of SQLite multi-file transaction is somewhat disastrous; inserting three 100 Byte records renders fifteen `fdatasync()`'s and 180 KBs write to the disk.

2.3 Log-structured Filesystem, F2FS and Atomic Write

We use F2FS [39] as a baseline log-structured filesystem. F2FS has a number of key design features that differentiate itself from the original log-structured filesystem designs [38, 60, 63]. Among them, the two features that we focus on in this work are *block allocation bitmap* and *dual log partition layout*. To realize *Stealing* and *Shadow Garbage Collection*, the way in which F2FS manipulates and updates the block allocation bitmap and the two logs must be overhauled.

The first is block allocation bitmap. In the original log-structured filesystem design [60, 63], there is no explicit data

structure that specifies whether a given block in the filesystem partition is allocated or not. The filesystem determines that a block in the filesystem partition is allocated if it is reachable through the file mapping. F2FS maintains the block allocation bitmap to denote whether a given block in the filesystem is valid or not. The second is dual log partition layout. Legacy log-structured filesystems treat the filesystem partition as a single log. They cluster the data block and the associated filemap[1] together and flush them in a single unit. F2FS organizes the filesystem partition with two separate logs: the data region and the node region. F2FS places the data block and the node block at the associated regions, respectively. Unlike the legacy log-structured filesystems, F2FS writes the data blocks and node blocks separately. To preserve the filesystem integrity against a system crash, F2FS ensures that the data blocks are made durable before the associated node blocks. Due to this ordering mechanism in F2FS, the block trace for writing the data block appears before the block trace for writing the node block in each pair of writes for the data block and the node block, as shown in Fig. 1.

F2FS provides the atomic write feature [34]; an application can write multiple blocks for a single file in a failure-atomic manner. This feature is primarily for addressing the excessive IO overhead of the SQLite's single file transaction.

```
start_atomic_write(fd) ;
write(fd, block1) ;
write(fd, block2) ;
commit_atomic_write(fd) ;
```

For atomic write, F2FS maintains the list of the dirty pages in the inode. When the transaction updates a file block, it inserts the dirty page to the per-inode dirty page list and pins the dirty page in memory. When the transaction commits, the filesystem unpins the dirty pages in the per-inode dirty page list and flushes the dirty pages and the associated node blocks that hold the updated file mapping to the disk. Since the atomic write pins the dirty pages in memory until it commits, F2FS, by design, cannot support Stealing in its atomic write transaction. When the transaction commits, F2FS sets the `FSYNC_BIT` flag at the node block. If more than one node blocks are flushed, atomic write places `FSYNC_BIT` flag at the last node block. F2FS sets the `FSYNC_BIT` flag at the node block to mark itself subject to the roll-forward recovery.

The log-structured filesystem periodically checkpoints its state, e.g. the updated file mapping, the updated bitmap (only for F2FS), and the disk location of the last block of each log. When the filesystem crashes, the recovery module recovers the state of the filesystem with respect to the most recent checkpoint information. After rollback recovery, the recovery module scans the logs from the last location, finds the node block with `FSYNC_BIT`, i.e. the transaction which has finished successfully after the most recent checkpoint, and recovers the associated file.

[1] F2FS calls blocks holding the file mapping information as a *node block*.

3 Design

We define three constraints which the transactional log-structured filesystem should satisfy: (i) Multi-File Transaction, (ii) Stealing and (iii) Transaction-aware Garbage Collection. We develop a transactional log-structured filesystem, exF2FS, that satisfies these constraints. The key technical components of exF2FS are Membership-Oriented Transaction, Stealing enabled Transaction, and Shadow Garbage Collection. Each component is summarized below.

Membership-Oriented Transaction (Section 3.1): The transaction of F2FS cannot span multiple files since it maintains the dirty pages of a transaction in a per-inode basis. In this study, we develop a new transaction model, called *Membership-Oriented Transaction*. In Membership-Oriented Transaction, the filesystem defines *Transaction File Group*, a set of files whose updates need to be handled as a transaction and maintains the dirty pages of a transaction for each transaction file group. In Membership-Oriented Transaction, a transaction can span multiple files and the application can explicitly specify the files that are subject to the transaction.

Stealing enabled Transaction (Section 4): For Stealing, the page reclamation procedure is overhauled so that the result of the page reclamation can be undone when the filesystem reclaims the dirty page of the uncommitted transaction. With Stealing enabled Transaction, the proposed filesystem can support large size transactions, e.g. hundreds of files with tens of GBs of data, with a small amount of memory.

Shadow Garbage Collection (Section 5): Garbage collection can make the dirty page associated with an uncommitted transaction durable and can checkpoint the updated file mapping prematurely before the transaction commits. We develop *Shadow Garbage Collection* to isolate the garbage collection from the uncommitted transaction.

3.1 Membership-Oriented Transaction Model

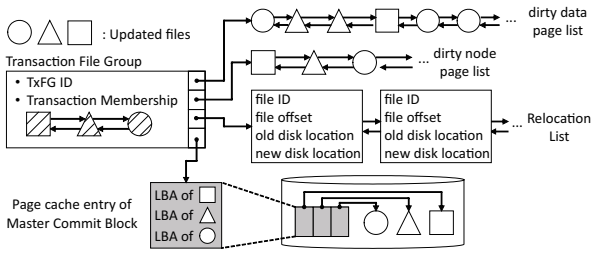


Figure 2: Concept of a Transaction: Transaction File Group, Dirty Page List, Relocation List and Master Commit block

In this work, we propose a new transaction model called *Membership-Oriented Transaction*. In this model, we define the new kernel entity, *Transaction File Group*. Transaction File Group is a set of files whose updates need to be treated

as a single transaction, and consists of Transaction Membership (a set of inodes), dirty page list, Relocation List, and Master Commit Block, as in Fig. 2. We use hash table for the namespace of Transaction File Group objects, which is widely used to organize the namespace for the kernel objects, e.g. semaphore and pipe [19].

With Transaction File Group, the application can specify the files that need to be included in the transaction. The dirty page list is a set of dirty pages for the transaction member files. There are two separate dirty page lists: the dirty data page list and the dirty node page list. A Relocation List is a set of Relocation Records. Relocation Record contains an information for the evicted page: file ID, file offset, old disk location, and new disk location. Master Commit Block holds the disk locations of the last node blocks for each file in the transaction membership. Transaction File Group, along with the Master Commit Block, allows the transaction to span multiple files. Relocation List is used for Stealing and Shadow Garbage Collection.

3.2 Transaction API's

```
1 id = create_tx_file_group();
2 for (i = 0; i<3; i++)
3     add_tx_file_group(db[i], id);
4 start_tx_file_group(id);
5 write(db[0], buf, 4096);
6 write(db[1], buf, 4096);
7 write(db[2], buf, 4096);
8 commit_tx_file_group(id);
9 delete_file_group(id);
```

Figure 3: Multi-file transaction in exF2FS

The application creates a Transaction File Group with an explicit call. When an application creates a Transaction File Group, and ID of the Transaction File Group is returned to the application. The application can add or remove a file to and from the Transaction File Group. To avoid a conflict between ongoing transactions, we forbid the application to add or to remove a file to and from the Transaction File Group in an ongoing transaction. When the transaction creates a file, the newly created file inherits the membership from the parent directory, which is called *Membership Inheritance*. Membership Inheritance saves the file created by the transaction from the transaction conflict since the newly created file is added to the Transaction File Group before it becomes externally visible. When the directory is removed from the Transaction File Group, child files who inherited the membership are also removed from the Transaction File Group.

The application specifies the ID of the Transaction File Group when it starts the transaction. When the transaction starts, the filesystem sets the flag at the inodes of the transaction member files denoting that the files are associated with the ongoing transaction. The application specifies the ID of the Transaction File Group when it calls for the transaction commit. exF2FS offers the API's for transaction abort and transaction delete. When the application calls for deleting a Transaction File Group, the Transaction File Group and the associated objects are deallocated if there is no ongoing transaction for the Transaction File Group, If there is an ongoing transaction when the application calls for deleting the transaction file group, exF2FS first aborts the transaction and then deletes the Transaction File Group. Table 1 illustrates the API's and pseudo-code of exF2FS, respectively.

In exF2FS, a transaction can include a directory update such as `rename()`, `unlink()`, and `create()`. The F2FS transaction does not support the directory update in the transaction.

3.3 Commit and Abort

When the transaction updates the file in the transaction file group, it inserts the updated page cache entry to the dirty data page list of the Transaction File Group.

In committing a transaction, the filesystem prepares the dirty data pages, the dirty node pages and the Master Commit Block for transaction commit. First, the filesystem inserts the dirty data pages in the dirty page list to the active data segment and obtains the disk location for each dirty data page. Second, the filesystem updates the associated node pages with the new disk location of each data page, inserts the updated node pages to the dirty node page list and determines the disk location for each dirty node page. Third, the filesystem allocates the Master Commit Block and stores the disk location of each node page in the dirty node page list at the Master Commit Block. The filesystem then sets `FSYNC_BIT` flag at the Master Commit Block.

Once these steps are complete, exF2FS flushes the dirty data pages, the dirty node pages and Master Commit Block. It ensures that the Master Commit Block becomes durable only after the data blocks and the node blocks become durable. *Master Commit Block* is the key component to fabricate the dirty pages of the multiple files into a single multi-file transaction. After the Master Commit Block becomes durable, the filesystem scans the Relocation List and invalidates the old disk locations of the Relocation Records. The details about the Relocation Record will be explained in Section 4.3.

If the transaction aborts, all entries in the dirty page list are discarded and the dirty page list becomes empty. When the aborted transaction has the evicted pages, the file mapping information is revoked to its original location based upon the Relocation Records.

When the system crashes, the recovery module performs rollback recovery and places the filesystem state to the most recent checkpoint. Then, exF2FS performs roll-forward recovery; it scans the log starting from the last logging offset recorded at the checkpoint. When it encounters Master Commit Block, the recovery module examines it and identifies the disk locations of the node blocks of the files in the transac-

API		Arguments	Return value	Description
Transaction File Group	`create_tx_file_group`	None	`int key`	Create a transaction file group
	`delete_tx_file_group`	`int key`	`int err`	Deallocate a transaction file group corresponding to the `key`
	`add_tx_file_group`	`int fd` `int key`	`int err`	Add a file `fd` to a transaction file group corresponding to the `key`
	`remove_tx_file_group`	`int fd` `int key`	`int err`	Remove a file `fd` from a transaction file group corresponding to the `key`
Transaction	`start_tx_file_group`	`int key`	`int err`	Start a transaction corresponding to the `key`
	`commit_tx_file_group`	`int key`	`int err`	Commit a transaction corresponding to the `key`
	`abort_tx_file_group`	`int key`	`int err`	Abort a transaction corresponding to the `key`

Table 1: API's in exF2FS

tion. Then, the recovery module of exF2FS uses roll-forward recovery routine of stock F2FS to recover the file associated with each node block. If the system crashes before the Master Commit Block becomes durable, the transient state of the transaction that was in-memory is completely lost. Through this recovery mechanism, exF2FS guarantees the atomicity and the durability of the transaction.

3.4 Concurrency Control and Isolation

Not being a full-fledged DBMS, we use coarse file-granularity concurrency control; a file can belong to only one Transaction File Group at a time. When adding a file to the Transaction File Group, the application checks if it is already in another Transaction File Group. If the file is already in another Transaction File Group, `add_tx_file_group` returns an error.

We leave the isolation support to the application as the other transactional filesystems do [11, 47, 72]. As the general purpose filesystem, it is difficult to meet all different levels of isolation requirements from a wide variety of applications at the same time. We carefully consider that the limited support of the filesystem for the isolation becomes redundant at best, unless the isolation level supported by the filesystem is well aligned with the isolation level required by the application. Text editor, application installer, git and the compaction of LSM-based key value store do not require the isolation [10]. SQLite and MySQL implement the multiple levels of isolation by themselves [49, 68]. In these applications, the limited support of filesystems for the isolation cannot be of much help. TxFS supports the isolation of "Repeatable Read" [4, 27]. It is overly strong for Text editor, and is too relaxed for some applications, such as "Serializable Read" in SQLite. SQLite must implement isolation of "Serializable Read" in its own database layer using the shared lock [67] even when using TxFS as the underlying filesystem. Filesystem support for the isolation has a cost. According to our experiments, the isolation support of TxFS renders 10% performance overhead due to the overhead of creating the shadow copies of the updated pages in the transaction. However, one limitation resulting from the absence of isolation support is that other processes cannot concurrently add, delete, or rename files in a directory that is included in another process's transaction. Supporting concurrent directory modifications is left for future work.

4 Stealing in the Filesystem Transaction

Stealing denotes the buffer management policy that allows the eviction of dirty pages of the uncommitted transaction [59]. The Steal policy in DBMS and the page reclamation of the Operating System (or the filesystem) [41] are the different manifestations of the same essential behavior: evicting a dirty page to the disk and freeing up the physical memory. While the two share the essential behavior, the two lie at the other end of extreme. For Stealing in the database transaction, DBMS prohibits the evicted dirty page from being externally visible (isolation) and/or undoes the Steal in case of transaction abort (atomicity). When the OS reclaims the file-backed dirty page, the result of the page eviction becomes externally visible and cannot be undone. In the journaling filesystem, the old file block is overwritten with the evicted page and in the log-structured filesystem, the old file block of the evicted page becomes unreachable due to the file mapping update.

4.1 Stealing and the Filesystem

The support for Stealing in the existing transactional filesystems bears substantial room for improvement. None of the TxFS [27], F2FS [34], Isotope [65], and Libnvmmio [11] support Stealing in the transaction. TxFS cannot support Stealing in a transaction due to its fundamental design limit. TxFS's support for transaction is built on top of EXT4 journaling. EXT4 journaling pins the log blocks in memory until the journal transaction commits. EXT4 limits the size of the journal transaction (256 MB by default). When the size of a journal transaction reaches its limit, the EXT4 journaling module commits the journal transaction. In EXT4, the dirty pages associated with a single system call can be split into two or more journal commits. TxFS must prohibit this from happening since it can make the transient state of the transaction durable prematurely, compromising the atomicity of the transaction. For atomicity guarantee, TxFS simply aborts the transaction when the transaction size exceeds its limit. F2FS pins the dirty pages of a transaction in memory until it commits. F2FS aborts *all* outstanding transactions [35] when the dirty pages of an uncommitted transaction exceeds a certain threshold (15% of the total physical page frames by default). CFS supports stealing [47]. However, CFS relies on a non-existent

transactional block device [32] for its support for Stealing. AdvFS [72] supports Stealing with the commodity hardware. AdvFS uses the writable file clone for the transactional updates. When the transaction commits, the filemap is updated to refer to the updated file blocks that are written in out-of-place manner. This nature allows AdvFS to freely support Stealing. However, a transaction in AdvFS can fragment the file since the filesystem deletes the old file blocks each time the transaction commits. The file defragmentation overhead of AdvFS is yet-to-be known. Our analysis on the AdvFS is limited since AdvFS is proprietary filesystem and the source code of its transaction module is not publicly available.

4.2 Delayed Invalidation and Node Page Pinning

In this study, we enable Stealing in the filesystem transaction. The log-structured filesystems [39,60,63] evict the dirty pages as follows: the evicted page is written to the new disk location, the old disk location of the evicted page is invalidated and the file mapping (node page in F2FS) is updated to refer to the new location of the associated file block. This page eviction routine cannot be used with Stealing for two critical reasons. First is the invalidation of the old disk location. Being invalidated, the old file block can be garbage collected and can be recycled before the transaction commits. If the old file block is recycled before the transaction commits, the transaction cannot be revoked when the transaction aborts. Second is the premature checkpoint of the updated node page. When the dirty page is evicted, the updated node page which contains the updated file mapping can be checkpointed if the filesystem runs the periodic checkpoint operation before the transaction commits. Then, the updated node page checkpointed to the disk refers to the new disk location of the evicted page of the uncommitted transaction. If the filesystem crashes before the transaction commits, the recovery module can recover the evicted page of the uncommitted transaction with respect to the most recent file mapping found on the disk. Subsequently, the filesystem can be recovered to the incorrect state.

There are two key issues that need to be addressed for supporting Stealing-enabled Transaction in the log-structured filesystem: (i) prohibit the old disk location from being garbage collected until the transaction commits and (ii) prohibit evicted pages of uncommitted transactions from being recovered after the system crash. To address the first issue, we propose *Delayed Invalidation*. In Delayed Invalidation, after evicting the dirty page from the uncommitted transaction, the filesystem postpones invalidating the old disk location until the transaction commits. To address the second issue, we propose *Node Page Pinning*. In Node Page Pinning, the filesystem pins the updated node page until the transaction commits to prohibit the updated node page from being checkpointed prematurely.

For Delayed Invalidation and Node Page Pinning, we introduce a new in-memory object, *Relocation Record*. Relocation Record holds the information associated with the page eviction. Relocation Record contains the file block ID (inode number and file offset), the old disk location, and the new disk location of the file block of the evicted page. With Relocation Record, the filesystem invalidates the old disk location asynchronously, not when it evicts dirty page but when it commits the transaction. Each transaction file group maintains a set of Relocation Records called the *Relocation List*. The filesystem creates the Relocation Record and appends it to the Relocation List when it evicts the dirty page in the transaction.

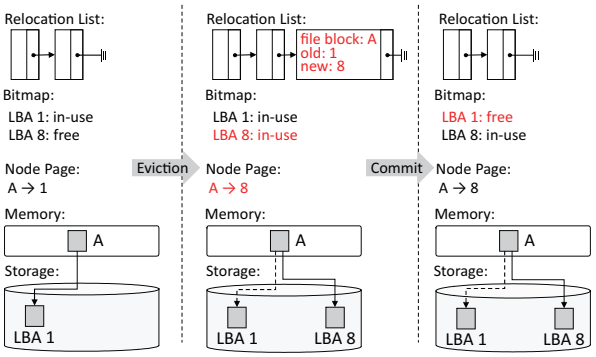


Figure 4: Delayed Invalidation: LBA 1 is invalidated not when the page is evicted but when the transaction commits.

Fig. 4 illustrates an example of stealing in exF2FS. The dirty page of the file block A is mapped to LBA 1 at the beginning. File block A is evicted to LBA 8. The node page in memory is updated to map file block A to LBA 8. The block bitmap for LBA 8 is set. The block bitmap for LBA 1 is *not* invalidated at the time of eviction due to Delayed Invalidation. The filesystem creates the Relocation Record and inserts the newly created record to the Relocation List. The newly created Relocation Record contains the file block ID (file block A), the old (LBA 1) and the new location (LBA 8) of the evicted block. Since LBA 1 is evicted to the disk, it is removed from the dirty page list of the associated Transaction File Group. When the transaction commits, LBA is invalidated and the updated node page is made durable.

4.3 Commit and Abort in Stealing

When the transaction commits, the filesystem makes the old location of the evicted page no longer reachable. Before it starts flushing the dirty pages, the filesystem scans the Relocation List in chronological order and invalidates the old disk locations of the evicted blocks (Delayed Invalidation). Once this finishes, it flushes the dirty data pages of the transaction. After the dirty pages become durable, the filesystem unpins the node page that has been updated in eviction and inserts it to the dirty node page list. Then, the filesystem flushes the

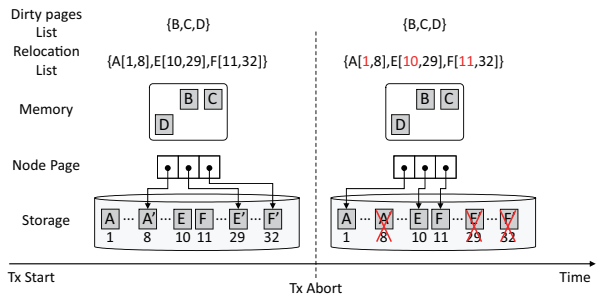


Figure 5: Stealing and Transaction Abort

dirty node pages. The transaction commits successfully if and only if the Master Commit Block becomes durable.

When the filesystem aborts the transaction, the filesystem scans the Relocation List in reverse chronological order. For each Relocation Record, the filesystem invalidates the new disk location and reverts the node page in memory to map the file block to the old disk location. After the node page is reverted, it is unpinned. Fig. 5 illustrates an example. At the time of abort, three pages have been evicted: A, E and F. The old location and the new location of page A corresponds to 1 and 8, respectively. In abort, the filesystem reverts the node pages for A, E and F to refer to page 1, 10 and 11, respectively, based upon the Relocation List. It also, invalidates the bitmap for the new disk locations, LBA 8, LBA 29 and LBA 32.

When the system crashes, Delayed Invalidation may leave the allocated but unreachable filesystem block. Delayed Invalidation temporarily leaves both the old and the new disk locations valid, from when the page is evicted until when the transaction commits. If the system crashes during this period, the filesystem can be recovered to the state where both old and new disk locations are valid but where only the old disk location is mapped to the file. If this happens, the new disk location needs to be collected through `fsck` [44] (offline) or through its online variant [17].

5 Transaction-aware Garbage Collection

We say that the garbage collection *conflicts* with the transaction if the garbage collection module selects a disk block which is associated with the dirty page of the uncommitted transaction as a victim for migration.

In this study, we develope a transaction-aware garbage collection technique called *Shadow Garbage Collection*. The Shadow Garbage Collection transparently migrates the victim block associated with the uncommitted transaction without any side effect to the transaction. F2FS performs the garbage collection in a transaction-aware manner but with substantial room for improvement; F2FS aborts *all* outstanding transactions when the garbage collection conflicts with any of the uncommitted transactions in the system [79].

5.1 Garbage Collection and the Transaction

The log-structured filesystem performs the garbage collection either in the foreground or in the background. Background garbage collection cannot conflict with the transactions since it runs only when the filesystem is idle. Here, the garbage collection implicitly denotes foreground garbage collection unless noted otherwise. The log-structured filesystem performs the garbage collection as follows. (i) First, the filesystem checkpoints the filesystem state (pre-GC checkpoint). (ii) The garbage collection module then selects the victim segment. (iii) Next, the garbage collection module migrates the valid blocks in the victim segment to the destination segment. This updates the associated file mapping to refer to the new disk location of the victim block. (iv) Finally, the filesystem checkpoints the updated state of the filesystem (post-GC checkpoint). The garbage collection module repeats step (ii) and step (iii) until it reclaims enough free segments. Pre-GC and post-GC checkpoints are essential in any log-structured filesystem to maintain its consistency against an unexpected filesystem failure.

In F2FS and a few other log-structured filesystems [38, 39, 77], the garbage collection module uses the page cache to migrate the victim disk block to the new location. In migrating the victim block, the garbage collection module first checks if the victim block exists in the page cache. There can be only one page cache entry for a single disk block. It is not possible to fetch the old data block into the page cache entry if the associated disk block already exists in the page cache. If the page cache entry for the victim block exists, the garbage collection module blindly writes the existing page cache entry to the destination without fetching the victim block from the disk. In this course, the garbage collection module may write the dirty page cache entry of the uncommitted transaction to the destination. After the garbage collection module migrates the victim disk block to the destination, the associated file mapping in the memory is updated to refer to the new disk location. Once the migration finishes, the garbage collection performs a checkpoint to make the state of the filesystem durable. As a result, the updated file mapping that refers to the new disk location of the victim block (dirty pages of the uncommitted transaction) becomes durable before the transaction commits. If the system crashes after the garbage collection finishes but before the transaction commits, the recovery module recovers the dirty pages of the uncommitted transaction. The atomicity of the transaction is then compromised.

5.2 Shadow Garbage Collection

In Shadow Garbage Collection, we reserve a set of page cache entries for garbage collection. We call this region *Shadow Page Cache*. When a victim block is associated with the uncommitted transaction, the garbage collection module uses Shadow Page Cache instead of generic page cache, to migrate

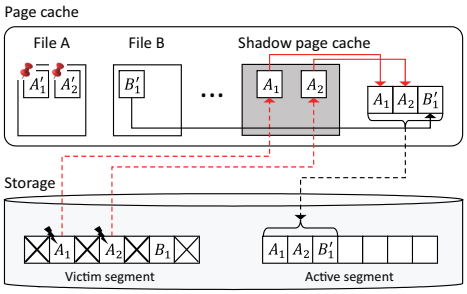


Figure 6: Shadow Garbage Collection: migrating A_1, A_2 and B_1. All are modified in memory to A'_1, A'_2 and B'_1, respectively. A_1 and A_2 are associated with an uncommitted transaction.

the victim block and the associated node block to the destination. Using the Shadow Page Cache in migrating the victim block to the destination, the filesystem prohibits the garbage collection from prematurely persisting the dirty pages of the uncommitted transaction. Fig. 6 illustrates an example of Shadow Garbage Collection. The disk block A_1, A_2 and B_1 are updated in the page cache to A'_1, A'_2 and B'_1, respectively. A_1 and A_2 are being modified by the transaction. The garbage collection module selects the disk block A_1, A_2 and B_1 as victims. In Shadow Garbage Collection, for migrating A_1 and A_2, the garbage collection module fetches A_1 and A_2 (the original version before the update) to Shadow Page Cache and flushes them to the destination. For migrating B_1, the garbage collection module uses the generic page cache since it is not associated with the transaction. Subsequently, it writes B'_1 (the updated version of B_1) to the destination segment.

The garbage collection can conflict with the uncommitted transaction in two ways; (i) the victim block can be associated with the evicted page by Stealing (type E, Evicted) and (ii) the victim block can be associated with the cached page (type C, Cached). When the victim block is associated with the evicted page, it can correspond to either the original file block before the update (type EO, Evicted and Old) or the updated file block (type EN, Evicted and New). When the victim block is associated with the cached page, the victim block corresponds to the original file block before the update (type CO, Cached and Old). Note that the victim block of type CN (Cached, New) cannot exist.

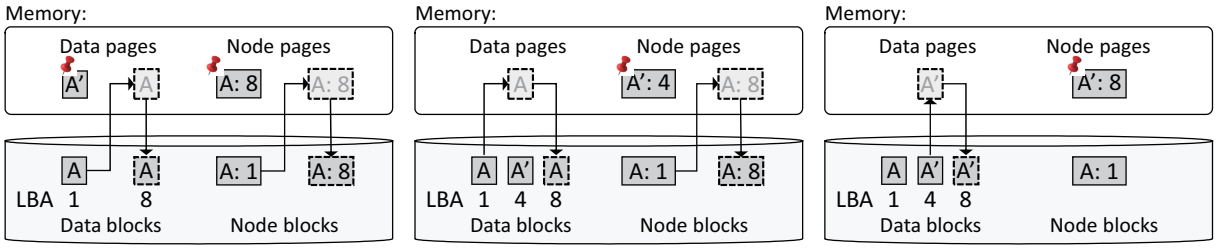
For each type of victim block, the Shadow Garbage Collection elaborately applies a different mechanism in migrating the victim block and the associated node block.

Type CO. When the victim block corresponds to old (O) version of the cached block (C) of the uncommitted transaction, we use the Shadow Page Cache in migrating the victim block and in storing the updated node block to the new disk location. In updating and storing the associated node block, the Shadow Garbage Collection updates the node block read from the disk, not the node block which has already been in the page cache. The node block in the page cache may have been updated since it is read from the disk and may contain transient file mapping that should not be made durable. After the garbage collection module finishes migrating both the victim block and the updated node block, it updates the node page in the page cache with the new file mapping. When the transaction aborts or the system crashes, the victim block at the migrated location can be recovered using the updated node block stored on the disk. An example of this can be seen in Fig. 7(a). File block A has been in LBA 1 and is updated in memory to A'. The disk block LBA 1 is selected as the victim. It is migrated to LBA 8 with shadow page caching. The associated node block is read into the Shadow Page Cache and is updated to map to LBA 8. Then, the updated node page is flushed to the disk. After both the victim block and the node block are flushed, the in-memory node block of file block A is updated to map to LBA 8.

Type EO. When the victim block corresponds to the old (O) version of the evicted page (E), we use the Shadow Page Cache in migrating the victim block and in storing the updated node block to the new disk location. Recall that the evicted page does not have the associated page cache entry (data page) and the associated node page is pinned in memory until the transaction commits due to Node Page Pinning. In migrating the victim block of type EO, the filesystem migrates the victim block using the Shadow Page Cache. For the node block update, we use the on-disk version of the node block as in the case of migrating the type EO victim block. After the Shadow Garbage Collection module finishes migrating the node block to the destination, it updates the node block pinned in memory with the updated file mapping. An example of this is illustrated in Fig. 7(b). The dirty page of the uncommitted transaction was evicted to LBA 4. File block A is migrated from LBA 1 to LBA 8. Shadow Page Cache is used to migrate the victim block and the associated node block. The node block that maps A is updated from "A:1" to "A:8" and flushed to the disk. The node page that maps the location of the dirty file block (A') of the evicted page remains unchanged in the page cache (A':4) and is pinned in memory.

Type EN. When the victim block corresponds to the new (N) version of the evicted page (E), we use the generic page cache in migrating the victim block to the new disk location. We can use the generic page cache, not Shadow Page Cache, in migrating the victim block since the victim block holds the most recent copy of the file block. The garbage collection module updates the node page in the page cache with the new file mapping after it migrates the victim block to the new location. An example is shown in Fig. 7(c). The updated file block of the evicted page A' is migrated from LBA 4 to LBA 8. Here, generic page cache (not Shadow Page Cache) is used to migrate the victim block. After the migration completes, the garbage collection module updates the associated node page in the page cache from "A':4" to "A':8".

When the garbage collection migrates the disk block associated with the evicted page, the garbage collection updates



(a) CO (LBA: 1→8) (b) EO (LBA: 1→8) (c) EN (LBA: 4→8)

Figure 7: Shadow Garbage Collection, A: original file block, A': updated file block. A rectangle with light grey background denotes the Shadow Page Cache.

the Relocation Record after the migration finishes. When the victim block is associated with the old disk location and the new disk location of the evicted page, it updates the old disk location field and new disk location field of the Relocation Record, respectively.

In implementing the Shadow Garbage Collection, we use an existing `META_MAPPING` object in Linux as the Shadow Page Cache. `META_MAPPING` is a special purpose `address_space` object, which is dedicated to cache the filesystem metadata [18]. Exploiting the existing `META_MAPPING` object as Shadow Page Cache, Shadow Garbage Collection does not require any new data structure for Shadow Page Cache in the kernel. Garbage collection of exF2FS (and also F2FS) reclaims the free blocks in a segment-granularity. Memory overhead for Shadow Garbage Collection corresponds to the size of a single segment, 2MB.

6 Applications with exF2FS

In this section, we explain how applications can exploit the transactional support from the underlying filesystem.

SQLite : Fig. 8(a) illustrates the implementation of the multi-file transaction in stock SQLite and in modified SQLite ported for exF2FS. In the stock SQLite's multidatabase transaction, the SQLite separately logs the updates to individual journal files and logs the global state of the transaction at the master journal file. In exF2FS, SQLite can implement its multi-file transaction with a single filesystem transaction eliminating the need for separately logging the individual database updates to the journal files.

Compaction in RocksDB: Fig. 8(b) illustrates the compaction in stock RocksDB and the compaction in RocksDB ported for exF2FS. In exF2FS, RocksDB can replace the multiple flushes of a compaction with a single filesystem transaction. In exF2FS, RocksDB can selectively exclude the `LOG` file from compaction transaction. It saves RocksDB from flushing the updates of the `LOG` file in making the result of compaction durable. The `LOG` file contains debugging information which is not an essential part of the compaction [21].

```
// without transaction support
write(/d/mj);
fdatasync(/d/mj);
fdatasync(/d);
while (/d/db[@]) {
  write(/d/log[@]);
  fdatasync(/d/log[@]);
  write(/dir/log[@]);
  fdatasync(/dir/log[@]);
  write(/dir/db[@]);
  fdatasync(/dir/db[@]);
}
unlink(/d/mj);
fdatasync(/d);
```

```
// with transaction support
while (/d/db[@])
  add_tx_file_group(tfg,
                    /d/db[@]);
start_tx_file_group(tfg);

while (/d/db[@]) {
  write(/d/db[@]);
}

commit_tx_file_group(tfg);
```

(a) Multi-database transaction in SQLite

```
// without transaction support
write(/d/LOG);
while (/d/sst[@]) {
  open(/d/newsst[@]);
  write(/d/newsst[@]);
  fsync(/d/newsst[@]);
  close(/d/sst [@]);
  write(/d/LOG);
}
fsync(/d);
write(/d/MANIFEST);
fsync(/d/MANIFEST);
write(/d/LOG);
```

```
// with transaction support
write(/d/LOG);
add_tx_file_group(tfg,/d);
start_tx_file_group(tfg);

while (/d/sst[@]) {
  create(/d/newsst [@]);
  write(/d/newsst[@]);
  write(/d/LOG);
}

commit_tx_file_group(tfg);
write(/d/LOG);
```

(b) Compaction in RocksDB

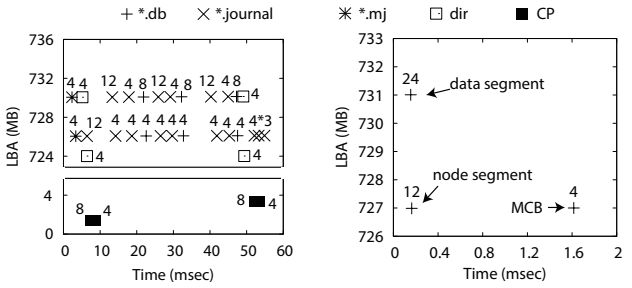
Figure 8: SQLite and RocksDB: with transaction support from the filesystem

7 Evaluation

Here, we evaluate the transaction feature of exF2FS. We implement exF2FS in Linux kernel 4.18. exF2FS is compared to three other filesystems: EXT4, F2FS, and TxFS [27]. TxFS [27] is the most recently published transactional filesystem based upon EXT4. TxFS was developed in Linux 3.18.22 and is not stable. For fair comparison, we re-implement only the atomicity and durability feature of TxFS on Linux 4.18.

Two storage devices were used in our experiment: Samsung 850 PRO [51] and Intel Optane 900P [29]. The 850 PRO and the Optane renders 1-2 msec and sub 10 μsec flush latency, respectively. We used a machine with an Intel CPU i7-9700K (3.60GHz, 4 core) and 64GB memory.

7.1 SQLite



(a) SQLite Transaction in F2FS (b) SQLite Transaction in exF2FS

Figure 9: IO trace: A multi-file transaction with three `insert()`'s in SQLite: F2FS vs. exF2FS. Record size: 100 Bytes. The number in each mark represents the number of KB written, Device: Samsung 850 PRO

Block level IO: We examine the raw IO behavior of the multi-file transaction in vanilla SQLite over F2FS and in SQLite with a multi-file transaction of exF2FS. Fig. 9(a) is the IO trace in vanilla SQLite over F2FS. A multi-file transaction consists of three `insert()`'s to three different database files. In vanilla SQLite, fifteen `fdatasync()` calls, two filesystem level checkpoints and a total of 32 write requests to the storage occur, taking 55 msec to complete a transaction. Fig. 9(b) illustrates the IO trace of SQLite's multi-file transaction when built with the multi-file transaction of exF2FS. There are three writes: one for the data blocks, one for the node blocks, and one for the master commit block, and takes 1.6 msec for a transaction. exF2FS resolves the excessive flush call problem.

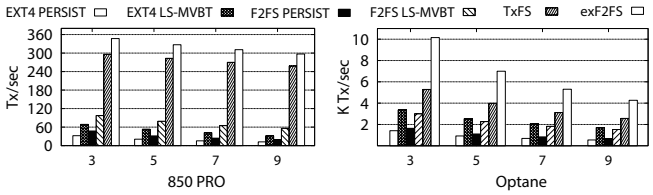


Figure 10: Transaction Throughput (Mobibench [3]-SQLite, `insert` operation), # of databases in a transaction = 3, 5, 7, 9

Throughput : We test the SQLite performance under the different SQLite journal modes and under different filesystems. For SQLite journal modes, we use PERSIST mode, and LS-MVBT [37]. PERSIST mode is the most popular journaling mode in SQLite. LS-MVBT [37] is the fastest SQLite journaling scheme known to the public. For the filesystem, F2FS, EXT4, exF2FS and TxFS are used, and Mobibench is used to generate the workload [3]. We port SQLite to use the transaction of exF2FS and TxFS. The results are shown in Fig. 10. In `insert` performance, exF2FS improves the throughput by as much as 24× against stock SQLite with PERSIST mode in F2FS (nine database files in a transaction, 850 PRO).

FS	Tput (KIOPS)	# of `fsync()`	# of cpt'n	compaction latency (sec)		
				Mean	99.9%	99.99%
F2FS	21.8	6135	892	18	153	373
exF2FS	40.8	622	622	7	50	51
EXT4	32.9	5873	862	9	48	88

Table 2: Throughput, total number of `fsync()`'s, total number of compactions, and compaction latency. cpt'n: Compaction

Let us compare the transaction performance of exF2FS against TxFS. As the storage gets faster, the performance benefit of exF2FS becomes more substantial than that of TxFS. In 850 PRO, SQLite exhibits 10% better performance in exF2FS than TxFS. In Optane, SQLite exhibits 100% better performance in exF2FS. The difference between exF2FS and TxFS are further elaborated in Section 7.4.

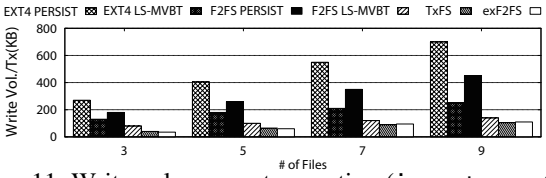


Figure 11: Write volume per transaction (`insert` operation), Number of database files in a transaction = 3,5,7,9

Write Volume: In all six transaction support methods, exF2FS creates the smallest amount of write (Fig. 11). Compared to F2FS with SQLite with PERSIST mode journaling, exF2FS with SQLite on the multi-file transaction generates 1/6 of the writes.

7.2 RocksDB Compaction

We found that using the transaction of exF2FS in RocksDB compaction produces two significant benefits: the performance improvement and the ability to handle the large size transaction. The YCSB benchmark (workload-A) is run for RocksDB. In this workload, a single compaction of RocksDB can create up to 13.3 GB of dirty pages with 198 SSTable files. Filesystems that pin the updated pages of the transaction in memory cannot perform RocksDB compaction as a transaction [27,39,65]. Here, the performance of transaction based RocksDB over exF2FS is compared with vanilla RocksDB over stock F2FS. The size of the memtable and the maximum size of the SSTable are both 64 MB. Key and value size are 23 Bytes and 1KB, respectively. Initially, RocksDB is populated with 50 M operations (55 GB). Then, YCSB-A is run with 50 M operations.

The performance results are summarized in Table 2. exF2FS improves YCSB performance by 87% against F2FS: 40.8 KIOPS vs. 21.8 KIOPS. On average, the compaction latency in exF2FS is 40% of the compaction latency in F2FS: 7 sec vs. 18 sec. The root cause for the performance and the latency difference is the number of `fsync()` calls. In F2FS, a single compaction creates seven `fsync()`'s on average, while

in exF2FS, a single compaction is executed with a single transaction which is equivalent to one `fsync()`.

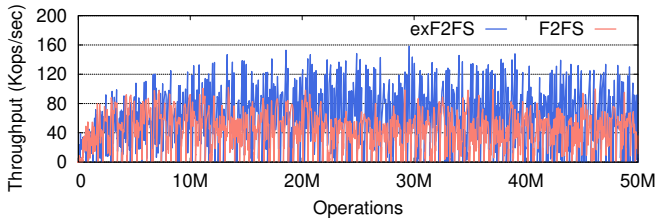
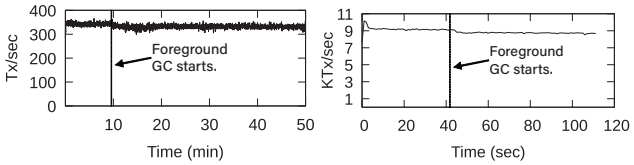


Figure 12: RocksDB Throughput in exF2FS vs. F2FS, YCSB workload A, a total of 50 M operations (read:write = 1:1), window size: 1 sec

We examine the throughput of RocksDB in exF2FS and F2FS (Fig. 12). The throughput is collected at one second intervals. Fig. 12 clearly shows that in RocksDB, exF2FS renders superior throughput behavior to F2FS. In this workload, 12% of the compactions are executed with stealing. On average, each compaction creates 100K dirty pages (400 MB) and 6K pages (24 MB) are evicted.

7.3 Garbage Collection



(a) 850 PRO (b) Optane

Figure 13: Throughput of multi-file transaction under foreground garbage collection in action (Mobibench [3]-SQLite, three `inserts` per transaction, record size = 100Byte)

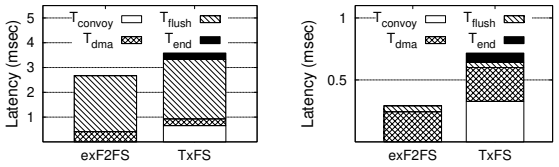
In F2FS, the transaction aborts when the garbage collection module selects one of its blocks as a victim block. In exF2FS, the transaction does not abort. However, the transaction is suspended until the garbage collection finishes when it encounters foreground garbage collection. Here, we examine how the garbage collection of exF2FS interferes with the throughput and latency of the foreground application. We also examine the throughput of the multi-file transaction (three inserts). The results are presented in Fig. 13. First, we mark the time when the foreground garbage collection is triggered. From then, the foreground garbage collection is triggered once every hundred transactions on average. With foreground garbage collection, the performance decreases by about 5%. Each foreground garbage collection reclaims a single free segment. With the foreground garbage collection, the tail latency (@99.9%) of the multi-file transaction has increased from 300 μsec to 470 μsec in Optane.

7.4 exF2FS vs. TxFS

We examine the detailed behavior of the transaction in exF2FS and TxFS. We use Mobibench [3] and generate the multi-file transaction in SQLite (`insert()`'s to three databases per transaction, record size: 100 Byte). While far from being complete, the analysis here provides a useful clue on how the log-structured filesystem and the journaling filesystem can fundamentally differ in supporting the transaction.

7.4.1 Convoy and Context Switch Overhead

In this section, we examine the latency of committing a transaction in exF2FS and TxFS. In 850 PRO and Optane, the commit latencies in exF2FS are 80% and 40% of those in TxFS, respectively. The latency difference between exF2FS and TxFS becomes more significant as the storage speed increases.

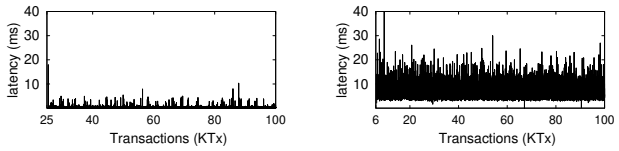


(a) Samsung 850 PRO (b) Intel Optane

Figure 14: Latency of multi-file transaction: exF2FS vs. TxFS (T_{convoy}: prepare for the commit, T_{dma}: time to transfer the blocks in the transaction, T_{flush}: time to make the blocks durable, T_{end}: wrap up the commit)

The latency to commit a transaction is partitioned into four components for detailed analysis: (i) prepare for commit (T_{convoy}), (ii) DMA transfer (T_{DMA}), (iii) flush (T_{flush}) and (iv) wrap up (T_{end}). The details of these are illustrated in Fig. 14. In exF2FS, the time for preparing a commit (T_{convoy}) includes preparing the Master Commit Block, constructing the IO commands and dispatching them to the storage. In TxFS, the time for preparing a commit (T_{convoy}) includes not only the time for preparing the journal descriptor block, constructing the IO commands and dispatching them to the storage, but also the time for writing *the unrelated data blocks to the disk*, the convoy [7]. T_{convoy} overhead is substantial in TxFS accounting for as much as 50% of the total commit latency (Optane). On the other hand, it is almost non-existent in exF2FS. This is due to the compound journaling of EXT4 [71]. EXT4 merges the updated metadata from multiple file operations into a single running transaction to increase the throughput of the filesystem journaling. Due to compound journaling, EXT4 can flush a large amount of unrelated dirty pages in an `fsync()` [30].

When the transaction is executed with the other metadata intensive applications, the convoy overhead of compound journaling becomes far more severe. Here, we examine the



(a) exF2FS, μ=169 usec (b) TxFS, μ=5.7 msec

Figure 15: SQLite: Latency of transaction with three inserts in SQLite, ten `varmail` threads [43] in the background, Optane, μ: average latency

Filesystems	Write Size	4KB	8KB	16KB	32KB
TxFS	Write	12GB	6GB	2.5GB	2.5GB
exF2FS	Volume	3GB	2GB	1.5GB	1.1GB

Table 3: Write Amplification of Transactional Write: Total Write Volume in writing 1 GB to a file (allocating write)

transaction latency of exF2FS and TxFS with a metadata intensive application, `varmail` benchmark [43], running in the background. Fig. 15 shows the result. The average transaction latency of TxFS is 34× that of exF2FS: 5.7 msec vs. 169μsec.

In exF2FS (or in F2FS), the filesystem commits the transaction in its own context. In TxFS (or in EXT4), the filesystem delegates the journal commit to the JBD thread, and the overhead of registering the committed blocks for the checkpoint and the context switch overhead, T_{end}, is non-negligible. T_{end} accounts for as much as 10% of the commit latency in TxFS while it does not exist in exF2FS. Due to the overhead of convoy and the context switch inherent in EXT4, exF2FS renders better transaction performance than TxFS.

7.4.2 Double Write and journal metadata overhead

We examine the write amplification of exF2FS and TxFS. The transactional write size varies from 4 KB to 32 KB and the total write volume is examined. Table 3 summarizes the result. In writing 1 GB with 4 KB atomic write, exF2FS writes 3 GB to the storage while TxFS creates 12 GB. In exF2FS, a 4 KB transactional write accompanies a 4 KB write for the node block and a 4 KB write for the Master Commit Block. In TxFS, a 4 KB transactional write (allocating write) journals four log blocks (superblock, inode table, data block, block bitmap), all of which are later checkpointed to their original locations. A double write overhead compound by the overhead of page granularity journaling renders a 12× write amplification in a 4 KB allocating write of TxFS. In exF2FS, the write amplification is 3× under the same workload. When the transaction size is 32 KB, exF2FS and TxFS render 1.1× and 2.5× write amplification, respectively. In this experiment, exF2FS does not perform any garbage collection. If it were included, it may render a larger write amplification. Unless the garbage collection amplifies the write volume by more than 2×, exF2FS renders less write volume than TxFS.

8 Related Work

Transaction support can be implemented in different layers of the software stack. TxOS [57] and QuickSilver [61] implement transaction support as a native kernel service. A transactional filesystem can readily be built using the interface offered by TxOS [26]. There are several kernel level filesystems that support transaction, such as AdvFS [72], TxFS [27], Valor [66], Transactional NTFS from Microsoft (TxF) [46], Failure-atomic msync() [55], and BTRFS [14]. OdeFS [24] and Inversion [54] are built as a user level filesystem and they rely on existing DBMS to realize an ACID property of the filesystem operation. CFS [47]'s crash consistency support is built on top of the transactional block device, X-FTL [32]. BVSSD [28], MARS [12], TxFlash [58], and Isotope [65] offer block device level transaction support. Libnvmmio [11] uses a user level log for its transaction support.

The degree of ACID support comes at the cost of the implementation complexity. Some works support full ACID (Atomicity, Consistency, Isolation and Durability) property [14, 27, 46, 66]. Some filesystems drop isolation support and support only ACD [47, 55, 72]. F2FS drops the durability and supports only AC in its atomic write [34]. By leaving the isolation support to the application, exF2FS limits the code changes to the local filesystem. TxOS requires a few 100K LOC [57]. Limiting the transaction support to the filesystem, TxFS reduces the required code changes to one tenth, 5K LOC. By exploiting the atomic write feature of F2FS and excluding the isolation support, exF2FS achieves its transaction support with 1.5K LOC.

9 Conclusion

In this work, we successfully address the three major issues of transaction support in log-structured filesystems: multi-file support, stealing and garbage collection. With the transactional log-structured filesystem proposed in this work, we can greatly simplify the application programming and can substantially improve the application performance in many popular applications including SQLite, RocksDB, and application installation.

Acknowledgements We are deeply indebted to our shepherd, Peter Macko, for helping shape the final version of this paper. We are also grateful to the anonymous reviewers for their comments that have greatly improved this paper. We also thank Seungyong Cheon, Jinsoo Yoo, Sundoo Kim, and Wonjong Lee for discussions and comments on earlier iterations of this work. This work was supported by IITP, Korea (grant No. 2018-0-00549 and No. 2018-0-00503), NRF, Korea (grant No. NRF-2020R1A2C3008525), and Samsung Electronics (IO201209-07867-01).

References

[1] http://vimdoc.sourceforge.net/htmldoc/recover.html.

[2] The manual of maildir. https://web.archive.org/web/19971012032244/http://www.qmail.org/qmail-manual-html/man5/maildir.html.

[3] Mobibench. https://github.com/ESOS-Lab/Mobibench.

[4] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. In *Proc. of 16th International Conference on Data Engineering (ICDE)*, 2000.

[5] 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 *Proc. of USENIX Annual Technical Conference (ATC)*, 2019.

[6] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, DL Moal, G Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proc. of the 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[7] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. The convoy phenomenon. *ACM SIGOPS Operating Systems Review*, 13(2):20–25, 1979.

[8] Jakob Blomer, Carlos Aguado-Sánchez, Predrag Buncic, and Artem Harutyunyan. Distributing LHC application software and conditions databases using the CernVM file system. *Journal of Physics: Conference Series*, 331(4), 2011.

[9] Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[10] Deka Ganesh Chandra. BASE analysis of NoSQL database. *Future Generation Computer Systems*, 52:13–21, 2015.

[11] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2020.

[12] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proc. of 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of the 1st ACM Symposium On Cloud Computing (SOCC)*, 2010.

[14] Jonathan Corbet. Supporting transactions in btrfs, November 2009. https://lwn.net/Articles/361457/.

[15] Anton Kuijsten Cristiano Giuffrida, Călin Iorgulescu and Andrew S. Tanenbaum. Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer. In *Proc. of 27th Large Installation System Administration Conference (LISA)*, 2013.

[16] Pia Malkani Daniel Ellard, Jonathan Ledlie and Margo Seltzer. Passive NFS tracing of email and research workloads. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[17] David Domingo and Sudarsun Kannan. pFSCK: Accelerating File System Checking and Repair for Modern Storage. In *Proc. of 19th USENIX Conference on File and Storage Technologies (FAST '21)*.

[18] elixir.bootlin.com. Definition of `META_MAPPING`. https://elixir.bootlin.com/linux/v4.18/source/fs/f2fs/f2fs.h#L1476.

[19] elixir.bootlin.com. ipc/util.c. https://elixir.bootlin.com/linux/latest/source/ipc/util.c#L171.

[20] Nick Elprin and Bryan Parno. An Analysis of Database-Driven Mail Servers. In *Proc. of 17th Large Installation System Administration Conference (LISA)*, 2003.

[21] Facebook. Rocksdb Compaction. https://github.com/facebook/rocksdb/wiki/Compaction.

[22] Facebook. RocksDB homepage. http://rocksdb.org/.

[23] Facebook. Rocksdb MANIFEST. https://github.com/facebook/rocksdb/wiki/MANIFEST.

[24] Narain H Gehani, Hosagrahar V Jagadish, and William D Roome. Odefs: A file system interface to an object-oriented database. In *Proc. of 20th International Conference on Very Large Data Bases (VLDB)*, 1994.

[25] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of apple desktop applications. *Transactions on Computer Systems (TOCS)*, 30(3):10:1–10:39, August 2012.

[26] Yige Hu, Youngjin Kwon, Vijay Chidambaram, and Emmett Witchel. From Crash Consistency to Transactions. In *Proc. of ACM HotOS*, 2017.

[27] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. TxFS : Leveraging File-System Crash Consistency to Provide ACID Transactions. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2018.

[28] Ping Huang, Ke Zhou, Hua Wang, and Chun Hua Li. BVSSD: Build built-in versioning flash-based solid state drives. In *Proc. of the 5th Annual International Systems and Storage Conference (SYSTOR)*, 2012.

[29] intel.com. Intel optane ssd 900p series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-900p-series.html.

[30] Daeho Jeong, Youngjae Lee, and Jinsoo Kim. Boosting Quasi-asynchronous I/O for Better Responsiveness in Mobile Devices. In *Proc. of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[31] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O Stack Optimization for Smartphones. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2013.

[32] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: transactional FTL for SQLite databases. In *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

[33] Dongui Kim, Chanyeol Park, Sang-Won Lee, and Beomseok Nam. BoLT: Barrier-optimized LSM-Tree. In *Proc. of the 21st International Middleware Conference (MIDDLEWARE)*, 2020.

[34] Jaegeuk Kim. F2FS: support `atomic_write` feature for database. https://lkml.org/lkml/2014/9/26/19.

[35] Jaeguek Kim. f2fs: limit # of inmemory pages. https://patchwork.kernel.org/project/linux-fsdevel/patch/20171019021516.65627-1-jaegeuk@kernel.org/#21076797.

[36] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proc. of 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[37] Wook-Hee Kim, Beomseo Nam, Dongil Park, and Youjip 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.

[38] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[39] 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.

[40] Wongun Lee, Keonwoo Lee, Hankeun Son, Wookhee Kim, Beomseok Nam, and Youjip Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2015.

[41] Henry M Levy and Peter H Lipman. Virtual memory management in the VAX/VMS operating system. *Computer*, 15(03):35–41, 1982.

[42] Paul McDougall. Microsoft pulls buggy windows vista sp1 files. *Information Week*, 2008. https://www.informationweek.com/software/microsoft-pulls-buggy-windows-vista-sp1-files.

[43] Richard McDougall and Jim Mauro. Filebench, 2005.

[44] Marshall Kirk McKusick, Willian N Joy, Samuel J Leffler, and Robert S Fabry. Fsck-The UNIX File System Check Program. *Unix System Manager's Manual-4.3 BSD Virtual VAX-11 Version*, 1986.

[45] Rémy Evard Michail Gomberg and Craig Stacey. A Comparison of Large-Scale Software Installation Methods on NT and UNIX. In *Proc. of the Large Installation System Administration of Windows NT Conference*, 1998.

[46] Frederic Miller and Agnes Vandome. *NTFS*. Alpha Press, 2009.

[47] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2015.

[48] Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system, 2002.

[49] mysql.com. Transaction Isolation Levels. https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html.

[50] Rebecca Nelson, Atul Shukla, and Cory Smith. Web Browser Forensics in Google Chrome, Mozilla Firefox, and the Tor Browser Bundle. In *Digital Forensic Education*, pages 219–241. Springer, 2020.

[51] news.samsung.com. Samsung Electronics leads consumers into the new era of multi-terabyte SSDs with Launch of 2-TB 850 PRO and 850 EVO. https://news.samsung.com/us/samsung-electronics-leads-consumers-into-the-new-era-of-multi-terabyte-ssds-with-launch-of-2-tb-850-pro-and-850-evo/.

[52] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. SHARE interface in flash storage for relational and NoSQL databases. In *Proc. of 2016 International Conference on Management of Data (SIGMOD)*, 2016.

[53] Sehyeon Oh, Wook-Hee Kim, Jihye Seo, Hyeonho Song, Sam H Noh, and Beomseok Nam. Doubleheader Logging: Eliminating Journal Write Overhead for Mobile DBMS. In *Proc. of 2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020.

[54] Michael A. Olson. The Design and Implementation of the Inversion File System. In *Proc. of USENIX Winter*, 1993.

[55] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync () a simple and efficient mechanism for preserving the integrity of durable data. In *Proc. of 8th ACM European Conference on Computer Systems (EUROSYS)*, 2013.

[56] Android Police. The Pixel 3 uses Samsung's super-fast F2FS file system, October 2018. https://www.androidpolice.com/2018/10/10/pixel-3-uses-samsungs-super-fast-f2fs-file-system/.

[57] Donald E Porter, Owen S Hofmann, Christopher J Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proc. of 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[58] Vijayan Prabhakaran, Thomas L Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proc. of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[59] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems, Chapter 16.7.1 Stealing Frames and Forcing Pages*. McGraw-Hill, 2000.

[60] 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.

[61] Frank Schmuck and Jim Wylie. Experience with transactions in QuickSilver. In *Proc. of 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[62] Margo I Seltzer. Transaction support in a log-structured file system. In *Proc. of IEEE 9th International Conference on Data Engineering (ICDE)*, 1993.

[63] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. An Implementation of a Log-Structured File System for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[64] Kai Shen, Stan Park, and Meng Zhu. Journaling of Journal is (Almost) Free. In *Proc. of 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[65] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Isotope: Transactional Isolation for Block Storage. In *Proc. of 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[66] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proc. of 7th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[67] SQLite.org. Isolation in sqlite. https://www.sqlite.org/isolation.html.

[68] SQLite.org. Pragma read_uncommitted. https://www.sqlite.org/pragma.html#pragma_read_uncommitted.

[69] SQLite.org. Pragma statements, 2012. http://www.sqlite.org/pragma.html#pragma_journal_mode.

[70] Jan Stender, Björn Kolbeck, Mikael Högqvist, and Felix Hupfeld. BabuDB: Fast and Efficient File System Metadata Storage. In *Proc. of International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2010.

[71] Stephen C Tweedie et al. Journaling the Linux ext2fs filesystem. In *Proc. of Annual Linux Expo*, 1998.

[72] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya S Mannarswamy, Terence Kelly, and Charles B Morrey III. Failure-Atomic Updates of Application Data in a Linux File System. In *Proc. of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[73] Ingo Weber, Hiroshi Wada, Alan Fekete, Anna Liu, and Len Bass. Supporting Undoability in Systems Operations. In *27th Large Installation System Administration Conference (LISA)*, 2013.

[74] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[75] Matthew J Weinstein, Thomas W Page Jr, Brian K Livezey, and Gerald J Popek. Transactions and synchronization in a distributed operating system. *ACM SIGOPS Operating Systems Review*, 19(5):115–126, 1985.

[76] Youjip Won, Sundoo Kim, Juseong Yun, Dam Quang Tuan, and Jiwon Seo. Dash: Database shadowing for mobile dbms. *In Proc. of 45th International Conference on Very Large Data Bases (VLDB)*, 12(7):793–806, 2019.

[77] David Woodhouse. JFFS: The journalling flash file system. In *Proc. of Ottawa Linux Symposium*, 2001.

[78] Charles P Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID semantics to the file system. *ACM Transactions on Storage (TOS)*, 3(2), 2007.

[79] Chao Yu. f2fs: avoid stucking GC due to atomic write. https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1667312.html.

A Log-Structured Merge Tree-aware Message Authentication Scheme for Persistent Key-Value Stores

Igjae Kim* *UNIST, KAIST*
J. Hyun Kim *UNIST*
Minu Chung *UNIST*
Hyungon Moon † *UNIST*
Sam H. Noh *UNIST*

Abstract

Persistent key-value stores (KVSs) are fundamental building blocks of modern software products. A KVS stores persistent states for the products in the form of objects associated with their keys. Confidential computing (e.g., Intel Software Guard Extensions (SGX)) can help KVS protect data from unwanted leaks or manipulation if the KVS is adapted to use the protected memory efficiently. The characteristics of KVSs accommodating a large volume of data amplify one of the well-known performance bottlenecks of SGX, the limited size of the protected memory. An existing mechanism, Speicher, applied common techniques to overcome this. However, its design decision does not scale because the required protected memory size increases rapidly as the KVS receives additional data, resulting from the design choice to hide the long latency of Merkle tree-based freshness verification. We find that the unique characteristics of the log-structured merge (LSM) tree, a data structure that most popular persistent KVSs have, help reduce the high cost of protected memory consumption. We design TWEEZER on top of this observation by extending RocksDB, one of the most popular open-source persistent KVSs. We compare the performance of TWEEZER with the reproduced version of Speicher. Our evaluation using the standard `db_bench` reveals that TWEEZER outperforms Speicher by 1.94∼6.23× resulting in a reduction of slowdown due to confidential computing from 16∼30× to 4∼9×.

1 Introduction

Persistent key-value stores (KVSs) are a cornerstone of modern software products. Many cloud services, such as Netflix [41], Facebook [58] and Uber [18], use these as a storage engine for large-scale data processing [17, 50] or database management systems [34, 36, 59]. Accordingly, KVSs are responsible for securely maintaining service data, including user credentials and private information. Thus, cloud-based services are motivated to protect their KVSs with the strongest mechanism available. This paper presents a KVS protected through hardware-based confidential computing (e.g., Intel SGX (Software Guard Extensions) [26]). While our work is not the first such work, our study is unique in that our solution is 1) tailored to the *log-structured merge* (LSM) tree, 2) general in that our solution is not tied to any particular hardware support, and 3) superior in performance (by up to 6.23×) compared to the state-of-the-art.

Hardware-based confidential computing offers strong security guarantees to such KVSs. Most KVSs run on public cloud services, leaving their content potentially open to anyone with control of the cloud platform's privileged software or physical machines. Confidential computing allows the KVSs to exclude these complex software layers and any hardware but the processor chip itself from the *trusted computing base* and rely only on the correctness of the processor implementation. The execution context within the processor chip is protected with access control mechanisms and those on external memory are protected cryptographically by encryption and the *message authentication code (MAC)*. Such a protected execution environment is commonly called an *enclave*.

This appealing security guarantee comes at the cost of performance. Among others, the cryptographic protection of external memory content introduces limitations in external memory usage. The confidentiality guarantee requires the in-memory data to be encrypted, while the integrity guarantee requires MAC computation and verification. As the cost of MAC increases with the total amount of memory that is available to an enclave, providing processors with large memory to an enclave becomes prohibitive. Thus, such memory, which is called the *enclave page cache (EPC)*, typically is available only in 128 MB or 256 MB [25] capacity depending on the choice of design and implementation. An enclave may use memory beyond EPC capacity, but the pages that do not fit in the EPC must be paged out of the EPC with similar cryptographic protection. Also, applications may access these memory pages only if the pages are loaded back into the EPC. Therefore, applications must be carefully redesigned to

*Work done mostly as an undergraduate student at UNIST.
†Corresponding Author

minimize such EPC paging and may have to store large data chunks with manual protection. This requirement has motivated many popular applications to be tailored to the enclave protection model [6, 9, 12, 16, 23, 31, 47, 54, 56].

For a persistent KVS to be protected through hardware-based confidential computing, it must be tailored considering the EPC limitations, as it is a memory-heavy application dealing with a large (e.g., more than tens of gigabytes) amount of data. Many persistent KVSs use the LSM tree for data in storage, accompanied by in-memory caches (e.g., *MemTable*) and write-ahead logs (WAL). The LSM tree and WAL must be manually protected with encryption and MAC, and the MemTable must also be tailored to efficiently employ EPC because it is relatively large by default (e.g., 64 MB). Speicher is the first work in this direction where it presents a design to efficiently protect the three large data structures that persistent KVSs commonly have [6]. The design divides the MemTable into two and places only the smaller one with more frequent access in EPC. The other two data structures are also protected with encryption and MAC with the Merkle tree [19]. However, this design choice slows down the KVS by up to 32.5× as the large Merkle tree induces longer latency for data retrieval from the LSM tree and increases the use of EPC pages by other caches as our analysis will show (§7.2).

This paper presents TWEEZER, which shares the same goal as Speicher. Similar to Speicher, TWEEZER is an extension of RocksDB [58], a popular LSM tree-based persistent KVS, that uses the MAC scheme tailored for LSM trees to run efficiently in an SGX enclave. However, TWEEZER is different from Speicher in that we make three critical design decisions on top of these invariants.

First, TWEEZER ensures the freshness of an LSM tree without constructing a Merkle tree spanning across its *sorted string tables (SSTables)*. This is possible by leveraging the principle that an LSM tree-based KVS comprises many SSTables, each containing many key-value pairs and remains immutable once built until it is compacted. Thus, if an SSTable is authenticated with a unique key and the key is never reused, an attacker cannot find other pieces of data anywhere other than the SSTable to perform the replay attack (§5.2).

Second, the uniqueness and invariant ordering of keys in each *data block* enable TWEEZER to encrypt and authenticate each key-value pair separately without losing the capability of detecting replays within an SSTable. We find that the invariant ordering among and within the data blocks enables TWEEZER to detect any attack on freshness without the Merkle tree generated for each SSTable (see §5.3).

Third, we find the classic hash chain [51] to be a good fit for authenticating the two logs: the WAL and MANIFEST logs. Hash chains allow TWEEZER to authenticate the logs without the trusted counters, which Speicher relies on, as well as to create as many new log entries as needed (§5.4).

We implement TWEEZER by extending RocksDB 6.14 [58] and using Scone [4], a library operating system designed to run unmodified applications in an SGX enclave. Besides the LSM tree-tailored message authentication scheme, we adopt the design choices for Speicher [6] for MemTable. We also reproduce Speicher for a comparison study due to the lack of an open-source version and demonstrate that the reproduced version provides similar performance.

Our experimental study using `db_bench`, the standard benchmark used for RocksDB, indicates that TWEEZER achieves the expected performance gain and EPC efficiency. When tested with extensive data, TWEEZER outperforms Speicher by 1.94∼6.23× depending on the workload and the data size. Evaluations using the same benchmark configuration that Speicher was evaluated with also exhibit similar performance benefits (1.91∼3.94×). Our analysis reveals that this improvement is primarily due to the 5.24∼7.57× reduction in EPC paging frequency.

2 Background

2.1 Intel SGX

Intel SGX [26] provides an execution environment called an enclave that has a protected memory region called the EPC [14] for programs that need protection. Only the program running in the enclave can access its EPC content that is cached in the CPU cache. When the data must be evicted to external memory, data are encrypted and authenticated using MAC by the *memory encryption engine* [27]. Thus, even strong attackers, who can replace external memory, cannot obtain or corrupt the EPC content and leave it undetected.

This memory protection mechanism is a well-known performance bottleneck [4, 12, 31]. The SGX computes the keyed *hash MAC (HMAC)* for each cache line, composes a modified version of the Merkle tree [22], and keeps its root within the CPU hardware to ensure the freshness of EPC content stored on external memory. Each cache replacement operation is accompanied by MAC verification using the Merkle tree to ensure that the EPC as a whole remains as written by the enclave. Partially for this reason, the size of the hardware-protected EPC is limited (e.g., 128 MB in general, 256 MB in recent releases [24]). For real-world applications that need larger memory, SGX provides paging of EPC, but the encryption and MAC verification make this paging expensive as well. Consequently, an application as-is that is not tailored to this policy suffers from significant performance overhead [6, 31].

Another source of performance penalty is the increased system call overhead. An enclave runs as part of a user process as a separated execution context from which an additional context switch is required to invoke system calls. Therefore, most applications running on an enclave adopt asynchronous system calls as a performance optimization [4, 6, 38, 43, 62, 63, 68], where an application creates a thread that stays in the user's context with the role to mediate system calls from the enclave. TWEEZER adopts this by running on SCONE [4].

2.2 LSM-based Key-Value Stores

RocksDB [58] is an open-source persistent KVS that is widely used in production and that uses the LSM tree [42] as its data structure for the key-value pairs in storage. The four critical components of RocksDB are the MemTable, SSTable, WAL, and MANIFEST log.

The MemTable is designed to reside in memory and stores recently added key-value pairs using a skip list for fast lookup. Every `put` operation fills the MemTable before the data are flushed to a persistent medium. If the size of the MemTable becomes larger than a configurable threshold, it is marked as immutable, and another MemTable is created to serve the following writes. At the same time, RocksDB triggers a background flush thread to move the immutable MemTable to a persistent medium in the form of an SSTable.

The new SSTables generated from a series of MemTables constitute Level 0 of the LSM tree. Any new read request must look up all of the SSTables in Level 0 because any SSTable could contain the key. Thus, RocksDB needs to keep retained the number of SSTables in Level 0, and thus, triggers an operation called *compaction* when the number of SSTables at Level 0 exceeds a configurable threshold. A compaction thread running in the background selects several SSTables, deletes duplicated keys, and compacts them to create a new SSTable stored at the lower level, Level 1 here, of the LSM tree in storage. The resulting levels satisfy an additional property of ordering. The compaction procedure ensures that one key appears at each level at most once (except for Level 0), and every SSTable is sorted, allowing the KVS to look up, at most, one SSTable per level to find a key-value pair.

One SSTable comprises several sub-blocks including *index blocks* and *data blocks*. The index block contains a sorted sequence of *index keys*. The ith index key is larger than or equal to the keys in ith data block and smaller than the keys in $i+1$th data block. At the end of an SSTable is a *footer block* containing padding to align the SSTables and a magic number marking the end of an SSTable. Speicher stores the MACs of the SSTable's key-value pairs in this footer block, resulting in increased EPC usage when the KVS becomes large.

2.3 Speicher

Bailleu et al. [6] were the first to study the problem of running RocksDB efficiently on an enclave and designed Speicher by adapting RocksDB. Speicher uses the Merkle tree to authenticate LSM tree by computing a MAC for each data block and building the Merkle tree on top. They propose three design changes to an LSM tree-based KVS considering the characteristics of the enclave and its protected memory, EPC. First, the MemTable must be adapted to reduce the EPC usage. Speicher redesigned MemTable so that a large portion of it, the values on leaves, are stored explicitly outside the EPC with cryptographic protection. This design change improves KVS throughput by reducing the number of EPC paging that Speicher causes. Second, the I/O calls must be handled at the user level by another thread to avoid leaving the enclave context on every call. Speicher runs with its own direct I/O library based on Intel SPDK [1], which reduces the cost of additional context switches. Third, the KVS should be properly timestamped to defeat the rollback and forking attacks. Speicher uses its own asynchronous monotonic counter that wraps the synchronous SGX monotonic counter because they could not use the SGX counter directly.

3 Threat Model

We assume a strong attacker could acquire complete control of a system running TWEEZER, except for the enclave's context that is protected by SGX. They may have obtained such control by exploiting a known vulnerability in the cloud provider's system or as an insider responsible for maintaining them. In particular, such attackers can read or modify the contents of memory or storage that the user's KVS uses, except for those in EPC that the SGX protects. However, the attacker cannot directly query the KVS because the KVS does not accept the attacker as an allowed client. The design and implementation of such an authentication protocol is a well-studied problem and orthogonal to the design of TWEEZER. We also do not aim to propose a new remedy to address implementation bugs that the current implementation of Intel SGX is known to have, potentially nullifying its security guarantee completely [11, 13, 60, 65, 67] as these are not fundamental flaws in its security model and will be fixed in future releases. This aligns with the assumptions made by most existing mechanisms built on Intel SGX [6, 9, 12, 16, 23, 31, 47, 54, 56].

4 Overview

TWEEZER is a persistent KVS running on an SGX enclave. To users, TWEEZER provides all operations that persistent KVSs usually implement as an extension of RocksDB [58]. The only additional requirement for users is to retrieve and keep a pair of cryptographic keys (§5.5) and place *heartbeat* transactions (§6). The key pair is required for TWEEZER to recover its data in case of a crash and the heartbeat transactions provide rollback resilience.

TWEEZER is built on top of the advances made by an earlier work, Speicher [6] (see §2.3), with three additional design decisions (D1∼D3 below).

D1. TWEEZER creates and associates one unique MAC key with each SSTable as shown in Figure 1 (①). Whenever TWEEZER stores data outside the SGX-protected memory, it computes the MAC to store along with the data to later verify the freshness. Among these data are the LSM tree, which comprises many SSTables in storage. The large size of this LSM tree, which contains almost all key-value pairs, could

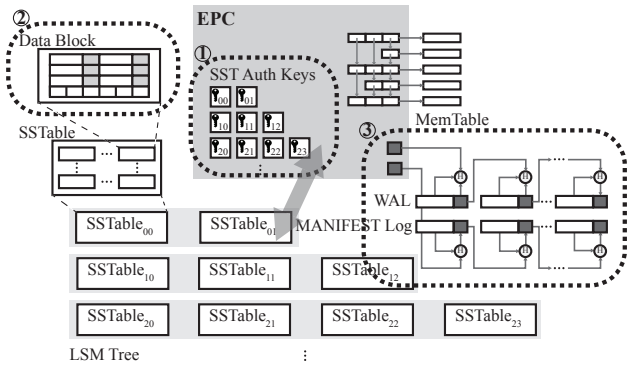


Figure 1: An overview of design choices made for TWEEZER.

make the MAC computation expensive. The best-known way to ensure the freshness of this large chunk of data is to build a Merkle tree [22], as Speicher does, at the cost of potentially long latency. TWEEZER avoids composing the Merkle tree spanning the entire LSM tree by associating a unique MAC key to each SSTable by taking advantage of the three properties of LSM trees: the immutability of each SSTable, the uniqueness of a key in each level, and the sorted keys in each data block. We elaborate on these design details in §5.2 and provide an in-depth security analysis in §6.

D2. We associate a MAC with each key-value pair rather than with each data block in an SSTable, as shown in Figure 1 (②). For encrypting and computing the MAC, the data block could be a natural unit. The SSTables are supposed to reside in storage optimized for block-level access, and RocksDB fetches and caches the key-value pairs at this granularity. What renders this design choice potentially inefficient is the limited size of EPC. For the desired security guarantee, the data block must reside in the EPC while being accessed, consuming valuable EPC space. This setup does not incur a significant performance cost when the KVS serves a relatively small data set and is configured to have only a small block cache. However, the blocks in EPC quickly become a performance bottleneck when the KVS requires a larger block cache to accommodate more data [69], which could quickly exhaust the small EPC. TWEEZER reduces this *read amplification* in EPC usage by encrypting and authenticating each key-value pair separately. This design choice enables TWEEZER to save EPC space and use non-EPC memory more effectively as a cache for SSTables. One drawback of this design choice is the increased use of storage space because the fine-grained encryption makes the subsequent per-block compression unproductive and increases the number of MACs stored in SSTables. We evaluate and discuss this in §8 (Figure 12).

D3. We overcome the absence of trusted counters [28] in the latest Intel SGX using hash chains (③ in Figure 1). Another performance-critical piece of data in persistent KVSs is the WAL that a KVS builds in storage to recover recently updated key-value pairs after a crash. The logs must be protected with encryption and MAC because they are supposed to reside in storage for persistence. Appropriate encryption and MAC computation provide confidentiality and integrity guarantees, but freshness requires each log entry to be associated with additional data. Speicher proposed to use the trusted counter [28] that an earlier version of the SGX SDK had, but which has been discontinued [21, 28]. Hence, TWEEZER constructs a hash chain to protect the content and the order of the logs. The hash chain alone is not enough to prevent the rollback attack, so TWEEZER requires the user to place a heartbeat transaction to timestamp the KVS version and use it later to verify that a snapshot of TWEEZER is the latest one. We elaborate on this aspect in §5.4.

5 Design and Implementation

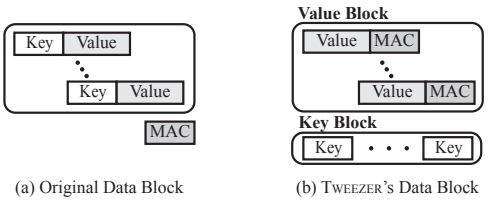
5.1 Data Encryption

TWEEZER manually encrypts all data that are stored outside EPC and decrypts them only within EPC. For example, TWEEZER ensures the SSTable content remains encrypted in both storage and memory and decrypts them only within EPC when it obtains a key-value pair from the SSTable. We use AES GCM mode with 256-bit key as the encryption scheme. As such, TWEEZER encrypts all data stored outside EPC to protect their confidentiality. For the rest of this section, we focus on how TWEEZER ensures freshness with the authentication scheme tailored for LSM trees.

5.2 Authentication with Per-SSTable Key

TWEEZER computes the HMAC of SSTables to later verify their freshness. When TWEEZER creates a new SSTable in the process of compaction or flush, it generates a new secret authentication key that is used exclusively for the particular SSTable (see ① in Figure 1) and stores it in EPC and the MANIFEST. TWEEZER then computes a MAC for each piece of data in the newly created SSTable (§5.3) and stores the MAC along with the encrypted data in the SSTable file. When TWEEZER reads the SSTable later to obtain a key-value pair, it computes the MAC again using the authentication key of the SSTables kept in EPC and compares it with the MAC stored along with the key-value pair to determine if the data has been maliciously corrupted or not.

The use of SSTable-unique keys and a set of invariant checks enable TWEEZER to guarantee the freshness of key-value pairs using HMAC. This HMAC is strong enough to prevent an attacker from generating correct MAC for an arbitrary piece of data because the correct MAC computation requires the secret key. However, a strong attacker who can obtain all pairs of data and MAC can replay the collected pairs. For example, if a KVS uses a single key to generate MAC for multiple SSTables ($SST_0, \cdots, SST_n$), the attacker can obtain pairs of



(a) Original Data Block (b) TWEEZER's Data Block

Figure 2: The structure of the original data block (left) and TWEEZER's data block (right).

SSTables and their MACs, $(SST_0, MAC_0), \cdots, (SST_n, MAC_n)$ and present one of the pairs to bypass the verification. Such a KVS using one key for multiple SSTables accepts this replayed pair because the computed MAC matches the presented MAC. To defeat this, the KVS must have a means to detect the replay, such as the Merkle tree.

TWEEZER avoids composing a Merkle tree spanning across all SSTables leveraging the uniqueness of the key for each SSTable and the immutability of SSTables. In general, the use of distinct keys suffices to prevent the replays across the set of data authenticated with different keys. However, this still does not prevent the replays within the data sharing one key because the attacker still has multiple pieces of data that they can potentially switch or replay. In particular, an attacker may reuse an older version of the data chunk (*temporal replay*) or the data chunks authenticated with the same key (*spatial replay*). TWEEZER prevents temporal replay against each SSTable by taking advantage of their immutability. By design, an update to an existing key-value pair in an LSM tree does not modify the SSTable. Instead, the new pair is stored in one of the SSTables at a lower level. The new pair is thus authenticated with a different key and the attacker cannot use the older pair to simply roll back the update. In other words, the attacker does not have an older version of key-value pairs authenticated with the same key because TWEEZER has never computed such MACs. We further discuss how TWEEZER prevents spatial replay in §5.3.

The performance improvement of Merkle tree-less authentication comes from the lower EPC usage. The large size of LSM tree makes MAC caching an essential design choice. To avoid a series of MAC computations along the Merkle tree for every SSTable read, the known good MAC for each data block must be cached within EPC. Speicher implicitly makes this design choice by storing the MAC in the footer block of each SSTable that RocksDB keeps within memory, within the EPC when it runs in an enclave (i.e., on Scone [4]). The cost of retaining the MACs remains small when a KVS accommodates a small number of SSTables, but increases quickly as the number of SSTables increases with the size of the KVS. The additional EPC usage for each SSTable varies depending on the configuration, but is roughly about 840 KB for each 64 MB SSTable, when the data block size is 4 KB and value size is 128 B. This roughly becomes a total of 840 MB if the KVS contains about 64 GB of key-value pairs, even assuming that it has no duplicates. This becomes a significant overhead considering the size of EPC, which is either 128 MB or 256 MB. Another option is to compute MACs along the Merkle tree whenever the KVS obtains a new block from storage, but this will significantly increase read latency considering the cost of a typical HMAC computation. The use of per-SSTable key enables TWEEZER to avoid storing too much data in EPC when serving a large KVS, as we show in §7.2 (Figure 10).

5.3 Fine-grained Authentication

TWEEZER authenticates each key-value pair individually to avoid read amplification. An SSTable is composed of many data blocks that are typically as large as 4 KB, containing 3∼28 key-value pairs (Figure 2a). This is a design choice considering the storage devices that are optimized for burst data transfers. Speicher chose this as the unit of encryption and authentication. It computes one MAC for each data block as shown in Figure 2a and stores the result in the SSTable's footer block. However, the encryption and authentication cost makes the inherent read amplification more expensive because the KVS must compute the MAC for the entire data block even when it reads only one key-value pair. In addition, such verified data must reside in EPC to avoid repeating the expensive verification, consuming invaluable EPC space. The block-level decryption and authentication limit the potential location of the block cache to EPC, and this could become a scalability bottleneck when the KVS is to contain large data.

In TWEEZER, we slightly rearrange the data block structure as shown in Figure 2b. This is similar to the key-value separation approach first proposed by Lu et al. [35], but rearranged for fine-grained encryption and authentication. Specifically, each data block is composed of one *value block* and one *key block*. The value block contains all values in the data block along with the corresponding MAC computed from both the key and value. The *key block* contains the sequence of keys along with the offsets of their values in the value block. This restructuring comes with two benefits. First, TWEEZER does not need to verify the freshness of the entire data block to obtain a single key-value pair, reducing read latency. Second, TWEEZER can place the block cache in untrusted memory, which is free from size limitations, because it can directly read a single key-value pair from an encrypted data block.

TWEEZER utilizes the LSM tree's invariant ordering to verify the freshness of the key-value pairs. Each data block in RocksDB's SSTable is composed of a sequence of ordered key-value pairs, and so are the keys in TWEEZER's key block. TWEEZER reads a data block when it fails to find the key in the MemTable or the SSTables at higher levels. In this process, TWEEZER firstly consults the index blocks that it keeps within EPC in plain text. By RocksDB design, each data block B_i in the LSM tree is associated with an index key in the index block, k_i. That is, the keys found in a data block B_i are not smaller than its index key k_i and not larger

Algorithm 1 Key Ordering

Input: i — The index of data block
Output: Returns *true* if the data block satisfies invariant.

```
1:  procedure CHECKORDERING(i)
2:      keyBlock = GetKeyBlock(i)
3:      firstKey = keyBlock.head
4:      lastKey = keyBlock.tail
5:      keyLowerBd = GetIndexKey(i)              ▷ Obtain index key in EPC
6:      keyUpperBd = GetIndexKey(i + 1)
7:      ret = true
8:      if firstKey < keyLowerBound then
9:          ret = false
10:     if lastKey ≥ keyUpperBound then
11:         ret = false
12:     for j in 1 ... keyBlock.length − 2 do
13:                 ▷ for each key in the key block except the first and last
14:         if keyBlock[j − 1] ≥ keyBlock[j] then
15:             ret = false
16:         else if keyBlock[j] ≥ keyBlock[j + 1] then
17:             ret = false
18:         else
19:             continue
20:     return ret
```

than the index key k_{i+1} of the next data block B_{i+1}. Utilizing this, TWEEZER performs a binary search on the index keys to obtain the data block potentially containing the key it is looking for.

To find the key from the obtained data block, TWEEZER decrypts and checks the ordering (Algorithm 1) of its key block, before looking for the key. If found, TWEEZER uses the offset associated with the key to obtain the encrypted value with the MAC. TWEEZER then computes MAC using the SSTable's authentication key, the queried key, and the value. By comparing this with the stored MAC, TWEEZER verifies the freshness of the key-value pair. If TWEEZER fails to find the key, it determines that the key does not exist in the level and moves on to the next level. Although TWEEZER does not perform MAC-based authentication for the key block, the ordering check (Algorithm 1) effectively mitigates any fault attack to deceive TWEEZER that a key does not exist in a data block. As discussed earlier, all keys in a data block must be larger than its index key and smaller than the next block's index key. TWEEZER checks this invariant by comparing the first and last key of a key block with the index keys (line 3–7). Subsequently, it compares each key in the key block with the neighboring keys (line 13–21) to ensure the ordering within the key block. As a result, any attempt to inject a fault to a key block will make TWEEZER interpret the key block as a different list of keys making the list highly unlikely to satisfy the ordering invariant. While the probability of a successful attack is not zero, as data blocks are small and the key space large, its probability will be very low. In more detail, when TWEEZER uses b-byte keys and the difference between the two index keys is D, the chance of a successful attack is roughly as small as $\frac{D}{2^{8\times b}}$. When D is 2^{64} and b is 16, this is about $5.42 \cdot 10^{-20}$ (2^{-64}). Thanks to this invariant-based freshness protection, the cost of reading a key-value pair becomes as small as one decryption of a key block, one decrypting of a value entry, and one MAC computation of a key-value pair. This is roughly $10\times$ smaller than the potential cost of a block-level scheme when a block contains 10 key-value pairs because the MAC operation dominates read performance.

This fine-grained encryption and authentication make it natural to place the block cache outside EPC, which TWEEZER does. This *untrusted block cache* is expected to be beneficial when TWEEZER is to accommodate large data in its LSM tree. By default, the block caches are placed inside EPC and Speicher left this in EPC as well because the loaded block is not encrypted. This does not become a performance bottleneck when a KVS accommodates a small amount of data. However, RocksDB is often configured to have a large block cache in production to serve a large amount of data, and the in-EPC block cache will not scale under this condition. The block caches may still be placed outside EPC, but this will significantly increase the block cache hit latency because every single cache hit triggers a decryption and verification of the whole data block. In contrast to this approach, the fine-grained encryption and authentication approach that TWEEZER takes enables it to take only a small portion of data into EPC from the block that resides in the untrusted memory outside EPC.

5.4 Protecting Logs with Hash Chains

TWEEZER ensures the integrity and freshness of the WAL and MANIFEST log using the classic hash chain [51, 52]. This hash chain is a good fit to protect those two data chunks because both are append-only lists and the freshness verification is performed only upon recovery. When TWEEZER starts to run either from an empty KVS or after recovery, it generates a nonce, considers the nonce as the first MAC (M_0), and creates a cryptographic key for MAC computation. For each new log entry (e_i), TWEEZER concatenates the encrypted data entry with the previous log ($M_{i-1}||E(e_i)$) to compute the next MAC (M_i) and stores it along with the encrypted data. The encrypted key-value pair becomes the data entry for the WAL, and the encrypted new MANIFEST becomes the data entry for the MANIFEST log. Like it does for each SSTable (see §5.2), TWEEZER generated a unique key to protect logs from replay attack. This use of a unique key prevents the attacks from replaying an entire log chain using an older one. The replayed log will be verified using a newer key, which is different from the one used for generating the older chain. Due to the differences in keys, the replayed MACs are not considered genuine ones, and TWEEZER recognizes this as a result of malicious manipulation. This hash chaining sufficiently prevents any attack on the hash chain's integrity and freshness as further discussed in §6.

We chose to use this hash chain for log protection rather than Speicher's mechanism that relies on the trusted counter for two reasons. First, the trusted counter that Speicher relies

on increments only once every 60 ms [6]. This limits the number of new log entries the KVS can create outside EPC to one per 60 ms, which is about 23.4 per second [6]. This is much lower than the expected number of write requests that a KVS is expected to serve. Speicher inevitably delays persisting new key-value pairs to overcome this limitation. In contrast, the hash chain mechanism does not suffer from this limitation. Second, support for the trusted counter on server platforms is not yet stable and its availability varies depending on the system configuration [21]. SGX is designed to use the trusted counter provided by the accompanying *Trusted Platform Module*, but not all server platforms have it. Furthermore, the SGX SDK for Linux does not provide the API as well [28]. TWEEZER's approach using the hash chain is, therefore, a more portable way to protect the logs.

5.5 Root of Trust

TWEEZER binds the confidentiality and integrity of its data to a pair of cryptographic keys and MAC computed from the MANIFEST. TWEEZER users retain these securely (e.g., in a physically isolated local machine) for full protection. TWEEZER uses the cryptographic key and MAC to recover data from the encrypted backup and to verify the backup's freshness. While running, TWEEZER uses these root keys to encrypt and authenticate the MANIFEST log that contains the KVS metadata. The other keys (§4) that TWEEZER uses are kept within the MANIFEST on persistent storage, residing in EPC during run time. This design choice allows TWEEZER to use the keys without significant delay and can later obtain a copy of those keys from the root key pairs and the MANIFEST file when it loads the data from a snapshot.

5.6 Primitive Operations

This section describes how TWEEZER execute the primitive operations for handling the requests.

PUT. TWEEZER handles a PUT request by inserting the key-value pair to WAL for persistence and to MemTable for efficient lookup. The new key-value pair is first encrypted with the dedicated log key, and the resulting data is used for computing a MAC along with the MAC of the previous entry in WAL (see §5.4). The encrypted pair is stored in WAL along with the computed MAC. TWEEZER follows a procedure similar to RocksDB's when it inserts a key-value pair to its MemTable, except for the cryptographic operations. TWEEZER's MemTable is located in both the EPC and untrusted memory, as proposed by Speicher (see §2). TWEEZER finds out the place in the untrusted memory where the value from the new pair will be stored using the internal nodes in EPC and store the encrypted value there. The MAC for the newly stored value is kept within EPC for verification of authenticity later.

GET. Upon receiving a GET request that accompanies a key, TWEEZER first looks up the MemTable within the EPC to determine if the key exists in the MemTable. If the key is found, TWEEZER obtains the encrypted value from the untrusted memory and MAC from EPC. The obtained MAC is then compared with the expected one computed using the key kept in EPC. Only if the stored MAC matches the computed one does TWEEZER consider the obtained value as a genuine one and respond to the request with it. If the key is not found from the MemTable, TWEEZER traverses the LSM tree as RocksDB does to find the pair with the requested key or determine that the key does not exist. TWEEZER finds an SSTable that is likely to contain the requested key like unmodified RocksDB, from the lowest level of the LSM tree, using the filter blocks and index blocks cached in EPC, with the sanity checks described in §5.3. From the data block, TWEEZER obtains the key block containing all keys, decrypts it in EPC, and finds the requested key. TWEEZER continues to the next level of the LSM tree if it fails to find the key from the key block. Otherwise, if the key is found, TWEEZER speculates that the key-value pair is stored in the current data block and obtains the encrypted key-value pair along with its MAC from the value block. TWEEZER verifies the obtained pair's authenticity using the authentication key for the SSTable (see §5.2), and responds to the client with the value if the computed MAC matches the stored one.

Range. As in RocksDB, TWEEZER handles range queries by first creating iterators and then traversing the data blocks in multiple levels. TWEEZER finds the starting key and initializes the iterators on each level by performing the same operations for handling GET requests. For each traversal, TWEEZER determines the latest version of the key-value pair like RocksDB, by checking the MemTable and then the LSM tree. If the key exists in MemTable, TWEEZER verifies its authenticity and decrypts the value as it does to handle a GET request. The case where TWEEZER finds the key-value pair from the LSM tree is also handled similarly, and TWEEZER verifies the absence of a key at a certain level as described in §5.3.

Recovery. TWEEZER follows the same recovery scheme that RocksDB implements, with the additional decryption and verification using the pre-shared credentials (i.e., keys and MAC). For this, TWEEZER takes the credentials as inputs in addition to the files constituting the KVS. The first piece of data that TWEEZER decrypts and verifies are the MANIFEST logs as discussed earlier (§5.5). As a result, TWEEZER obtains the latest MANIFEST that contains the structure of TWEEZER across the files and cryptographic keys needed to decrypt and verify the rest of the data chunks. In particular, TWEEZER obtains these keys from the recovered and verified MANIFEST update log called *version edit*. Each version edit contains the changes made to the KVS structure such as SSTable creation, SSTable deletion, log entry creation or log entry deletion. TWEEZER extends these records with the

additional keys that it uses such as the per-SSTable keys or log key. Aside from the additional decryption or verification steps, recovery is done following RocksDB's scheme. After recovery, TWEEZER provides the remote user with heartbeat data that represents the exact version of the snapshot (see §6).

6 Security Analysis

This section discusses in-depth about how TWEEZER ensures the integrity and freshness of its data against potential attacks.

Replay Attack. Reuse of existing encrypted data-MAC pairs is a common attack strategy against data integrity. To TWEEZER, this is the only way for attackers to pass the MAC-based verification procedure. Under our threat model the attackers can obtain these data-MAC pairs stored outside EPC because they are assumed to have full access to the memory content outside the EPC as well as the storage content. With this strong capability, an attacker may aim to replay TWEEZER's data chunks such as MANIFEST log, WAL, a whole SSTable, or individual key-value pairs.

Log Replay. The first two targets, MANIFEST log and WAL, are protected by the hash chain. TWEEZER and the remote user are assumed to have the key pairs for encryption and MAC computation along with the nonce that TWEEZER uses as the first hash. When TWEEZER recovers from a snapshot, TWEEZER correctly determines if each log entry is a replayed block or not through the MAC verification for the following reason. To replay the ith block b_i from the list of log entries $b_0, \cdots, b_n$ and pass the verification procedure, the attacker must generate or obtain MAC M'_i computed from $h_{i-1}||b'_i$ using the correct MAC key, where b'_i is the replayed block and h_{i-1} is the correct MAC of the previous (i.e., $(i-1)$th) block. However, the attacker cannot obtain such M'_i because of the uniqueness of the MAC key and the blocks in the log. The only data chunks with the corresponding MAC computed using the MAC key are the log entries. Therefore, the attacker can only choose one from $b_0, \cdots, b_n$ as the b'_i. If the attacker chooses b_j as b'_i, the only MAC available to the attacker is the one computed from $h_{j-1}||b_j$, which does not pass the verification procedure because $h_{j-1} \neq h_{i-1}$ when $j \neq i$.

Key-Value Pair Replay. TWEEZER recognizes any attack against the latter two (a whole SSTable and an individual key-value pair) when it verifies their freshness using the MAC. An attacker's strategy in this scenario can be classified into three groups. First, the attacker may try to replace one SSTable as a whole with another. TWEEZER detects this attempt when it obtains data blocks from the SSTable and verifies the block through MAC computation. Similar to the earlier scenario, the attacker cannot obtain the appropriate MAC because the key-value pair that the attacker aims to replay has never been used to compute a MAC with the target SSTable's key. Each SSTable is authenticated with its unique key, so the MACs associated with key-value pairs in another SSTable are considered incorrect by the verification procedure. Second, an attacker could try to replay data chunks within one SSTable. TWEEZER recognizes this using the invariant ordering of SSTables in an LSM tree as discussed in §5.3. If the replay is somehow performed within a key block, the attacker inevitably breaks the ordering. If the replay switches two keys k_1 and k_2 and k_1 is to appear earlier than k_2, the replay makes k_2 appear earlier than k_1, breaking the invariant ordering. Duplicating a key is not an option as well because it breaks the uniqueness principle. The last strategy that the attacker can choose is to replay across the key blocks within an SSTable, but it violates the property of the index key, which partitions the set of keys an SSTable contains into contiguous and mutually disjoint ranges.

Rollback Attack. TWEEZER ensures that the user has the latest version of its data at the granularity of heartbeat transactions. A strong attacker that we assume may place a rollback attack where they take a snapshot of TWEEZER's data at some point and later present to TWEEZER or its remote user as the genuine and latest version. The online rollback attack that an attacker performs while TWEEZER is running is infeasible because the attacker cannot replace the data stored within EPC. An offline attack in which the attacker replaces TWEEZER's files with an older version could, however, be a realistic threat. To thwart such attacks, TWEEZER relies on periodic interaction with the user to timestamp the versions by periodically issuing a write transaction to TWEEZER. Later, these resulting key-value pairs are used to determine the TWEEZER snapshot version. These additional timestamps provide rollback-resilience because the other verification mechanisms prevent the attacker from forging a fake snapshot. When given a snapshot to recover from, TWEEZER and its user verify its freshness using the root key pairs, starting from the MANIFEST log. As discussed in §5.6, an attacker who does not have these key pairs cannot make any modifications to any older TWEEZER snapshot version. The only remaining option is to present an exact copy of an older version, but the user correctly determines the copy's version from the key-value pairs from the heartbeat transactions after the verified recovery.

Existence Attack. TWEEZER also detects any attempt to deceive it into believing that an SSTable does not contain a particular key when, in fact, the SSTable has it. The LSM tree design strengthens this attack if successful because TWEEZER may consider an older version of the key-value pair found in a lower level. The LSM tree-based KVSs handle update requests by adding the new key-value pair to the higher level of the LSM tree and leave the older one in a lower level. An attacker performing the existence attack must first find the key block that contains the victim key and forge a valid key block passing TWEEZER's check. The confidentiality that TWEEZER ensures using encryption prevents this first step, which leaves an attack to an unknown key as the only remaining option. However, this option is also highly unlikely because of TWEEZER's invariant check, as discussed in §5.3.

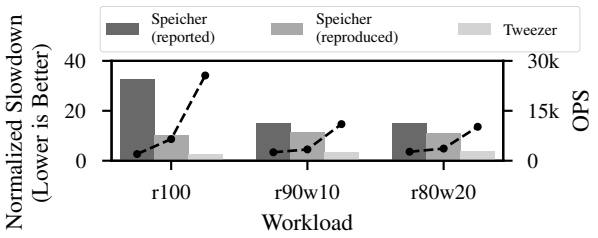


Figure 3: The normalized slowdown of TWEEZER and reproduced Speicher, along with the reported slowdown of Speicher [6], where the unmodified RocksDB is the baseline. The absolute throughput is also presented as lines using the y axis on the right.

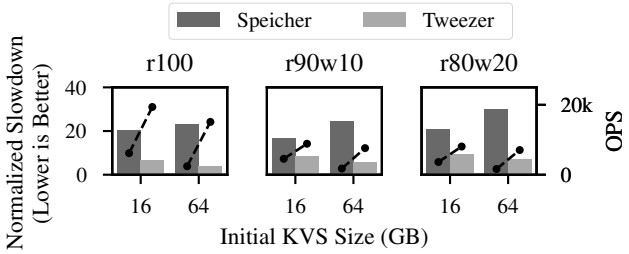


Figure 4: Normalized slowdown (lower is better) relative to the original RocksDB on SGX of TWEEZER.

7 Performance Evaluation

Environment. We evaluate the performance of TWEEZER on a machine with Intel Xeon E-2288G and 64GB of DRAM. The CPU has 32 KB instruction and data caches, 256 KB of L2 caches and a 16 MB shared L3 cache. The CPU also implements Intel SGX for confidential computing and AES-NI to speed up AES block cipher. The system runs Ubuntu 18.04 with Linux Kernel 4.15. For every cryptographic operation, we used OpenSSL 1.1.1i. Specifically, we chose AES GCM 256 as the block cipher scheme to protect the confidentiality of the data, GHASH to compute MACs for the logs and MemTable, and HMAC with SHA3-384 to compute MACs for SSTables. We followed the schemes that Speicher used to rule out the performance impact of cryptographic schemes when we compare TWEEZER and Speicher. Note that, unlike the encryption or GHASH, the HMAC computation does not benefit from hardware acceleration because the CPU that we use does not have hardware extensions to accelerate for SHA computation. We built both TWEEZER and the reproduced Speicher based on RocksDB version 6.14.

Benchmarks. We evaluate TWEEZER using `db_bench` with three workloads, r100, r90w10, r80w20 each of which is composed of 100% reads; 90% reads and 10% writes; and 80% reads and 20% writes; respectively. The key size is 16 B, the SSTable size is 64 MB, and 5 million key-value pairs were used, as done in Speicher [6]. The block size is either 4 KB, which is the default of RocksDB, or 32 KB, what Speicher used for its evaluation. In some experiments, we use `db_bench` to create KVSs as large as 16 GB and 64 GB, and use them to evaluate the performance of TWEEZER on a practical setup.

Reproducing Speicher. For comparison, we reproduced Speicher by extending RocksDB because Speicher is not open-sourced. As discussed in §1, TWEEZER adopts some of Speicher's design decisions to save EPC space and relies on Scone for asynchronous system calls. As such, the reproduced Speicher shares these aspects with TWEEZER in our implementations. Figure 3 shows the normalized throughput of TWEEZER, the reproduced Speicher, and the original Speicher as bars, and the absolute throughput as lines. Hereafter, all normalized results are normalized to the baseline RocksDB, presented with the absolute throughput. The experimental results advocate that our reproduction of Speicher is reasonable in that it exhibits similar or better performance characteristics compared with the reported number. In this experiment, we issue 5 million transactions starting from a KVS filled with 5 million entries, set the value size to 1024 B, and set the block size to 32 KB to replicate the experiments as close to those of the original setting [6]. We note that the difference in experimental setup may have also contributed to the better performance that replicated Speicher exhibits compared to the original. Speicher was evaluated with on a machine with Xeon E3-1270 v5, which has smaller (8 MB) shared L3 cache and smaller EPC (128 MB), albeit the size of main memory is the same. Larger EPC and caches potentially reduce the number of cryptographic operations while Speicher runs, reducing the overhead of storing data within EPC. Regarding the results, we observe that TWEEZER outperforms Speicher by 1.91∼3.94× despite the fact that the KVSs run with smaller amount of data. The 5 million entries are actually small enough that EPC paging does not occur, favoring Speicher considerably.

7.1 Throughput

Point Lookups. Figure 4 shows the normalized throughput of TWEEZER and Speicher on three workloads from `db_bench` with varying initial KVS sizes and 1024B values. The block size is set to 4 KB, which is the default and the best for the original RocksDB. Starting from the KVS images that we created using `db_bench`, we issue 5 million transactions to measure the throughput. Under the tests using these large KVSs, TWEEZER consistently outperforms Speicher by 1.94∼6.23×, reducing the slowdown from 16∼30× to 4∼9×. Our observation (§7.2) suggests that this performance gap is primarily due to EPC paging. As the KVS size increases, Speicher's footer cache in EPC becomes larger and causes frequent EPC paging. The use of per-SSTable keys reduces the amount of data that must be kept within EPC, enabling TWEEZER to avoid the frequent EPC paging.

Range Query. We evaluate the range query performance

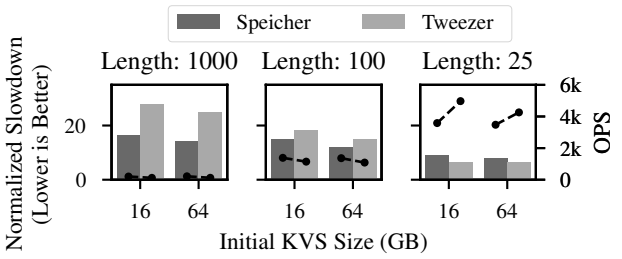


Figure 5: Normalized throughput of TWEEZER and Speicher on range queries with varying length. Length refers to the number of key-value pairs being accessed for each range query.

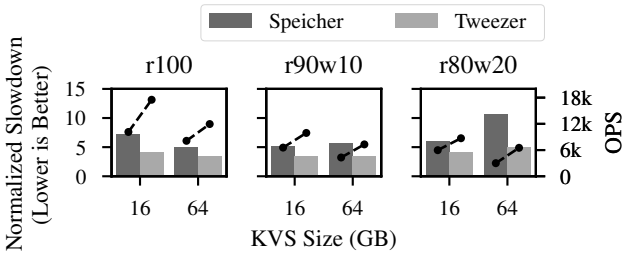


Figure 6: Normalized performance of TWEEZER and Speicher with large (32 KB) data blocks.

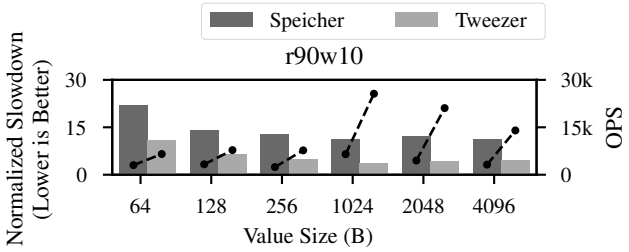


Figure 7: The normalized performance of TWEEZER and Speicher when running for different value sizes.

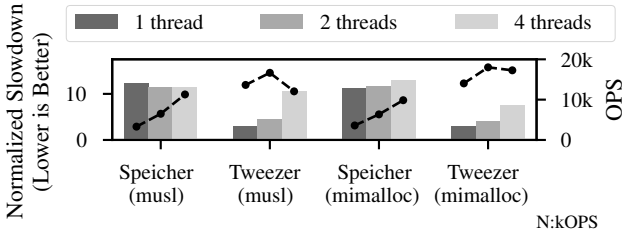


Figure 8: Normalized throughput of TWEEZER and Speicher with two different memory allocators, musl and mimalloc, as the number of threads increases.

of TWEEZER and Speicher using `seekrandom` benchmark in `db_bench`, with 1024 B values and 32 KB data blocks. As Figure 5 shows, TWEEZER exhibits higher throughput than Speicher for short queries, but the advantage diminishes as the query length increases. This result is due to the fine-grained authentication (§5.3) that is optimized only for point lookups. When obtaining a key-value pair, the cryptographic cost is smaller in TWEEZER than Speicher that decrypts and authenticates a whole data block even for a single request. However, this whole-block decryption and authentication become less costly when handling range queries because Speicher decrypts and authenticates the block only once for multiple pairs. Unlike this, TWEEZER has no choice but to handle range queries like a sequence of point lookups, thus authenticating the key-value pairs separately.

Block Sizes. The data block size in an SSTable can be configured and may affect throughput. While our experiments on point lookup performance used the default value of 4 KB as this setting results in the best performance for the baseline RocksDB, Speicher, in their experiments, used 32 KB blocks. Thus, we perform the same experiments obtained for Figure 4 except with the block size set to 32 KB. From the results in Figure 6, we observe that Speicher performance improves considerably due to the reduction in EPC usage (see §7.2). Despite this, we see that TWEEZER still outperforms Speicher by 1.46∼2.17×.

Value Sizes. Figure 7 shows how the value size affects the performance of TWEEZER and Speicher. For these experiments, we used the same setup as the comparison study in Figure 3 except for the value sizes. Overall, we observe that TWEEZER outperforms Speicher by 1.82∼4.70× in all configurations because TWEEZER also, in part, suffers from read amplification, which diminishes as the value size increases, resulting in reduced slowdown.

Number of Threads. Figure 8 shows the normalized throughput of TWEEZER and Speicher as the number of threads increases. Speicher scales similarly to RocksDB, but TWEEZER's slowdown increases as the number of thread increases. According to our analysis, this is due to the default heap allocator of Scone, musl [49] that does not scale as the number of threads increases. TWEEZER's shows scalability when we replace musl with mi-malloc [39] but the benefit was limited because Scone [55] does not support the thread local storage model that mi-malloc uses. Nevertheless, TWEEZER outperforms Speicher by 1.78× when running with 4 threads.

Untrusted Block Cache. Fine-grained authentication (§5.3) enables TWEEZER to place the block cache in untrusted memory, outside EPC. Having its block cache outside the EPC is beneficial when TWEEZER starts to serve a large KVS in which larger block cache could help reduce the average read latency. Figure 9 presents the normalized throughput of TWEEZER and Speicher as we change the block cache sizes from 8 MB (default) to 128 MB and 256 MB using the same setup as the comparison study in Figure 3. Speicher's performance overhead increases as the block cache size increases because the additional block caches cause more EPC paging. Speicher places all block cache content in EPC as it does not make any adjustment to the block cache management, increasing EPC usage and resulting in more EPC paging. On the contrary, TWEEZER does not suffer from the increased

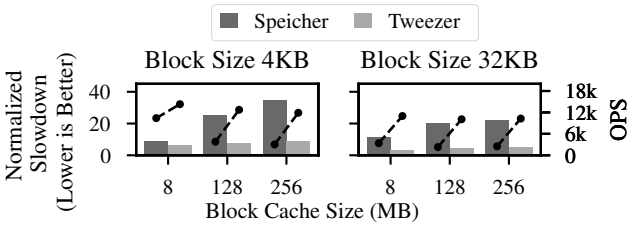


Figure 9: Normalized throughput of TWEEZER and Speicher as the block cache size increases.

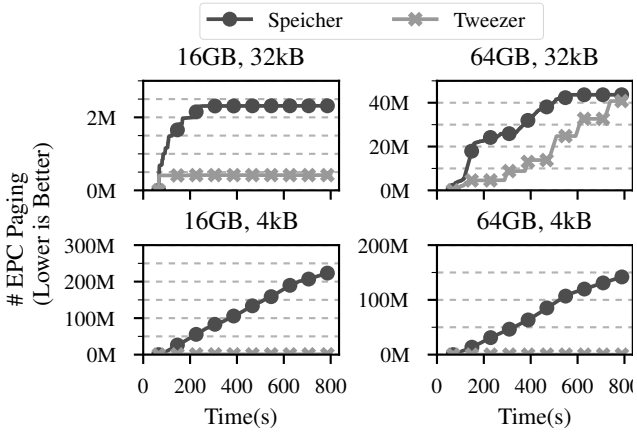


Figure 10: Cumulative number of EPC paging while running TWEEZER and Speicher with varying block sizes (4 KB and 32 KB) and initial KVS sizes (16 GB and 64 GB).

number of EPC paging because it places the block cache outside the EPC with the same cryptographic protection as the blocks in the SSTables. TWEEZER still does not benefit from the block caches as the absolute numbers show, however, due to the relatively small benefit that the block cache brings to TWEEZER compared to the unmodified RocksDB. Block cache miss penalty is high in unmodified RocksDB because it decompresses the retrieved data block on cache misses. The cache hit latency is long on TWEEZER, which additionally decrypts and authenticates the retrieved pairs.

7.2 EPC Usage

EPC Paging. We obtained the number of EPC paging using `sgxtop` [32]. Figure 10 shows the cumulative number of EPC paging observed while running the two configurations of the benchmarks used for the experiment in §7.1. Specifically, we accumulated all observed EPC paging from each run after the recovery because neither TWEEZER nor Speicher is designed to optimize the recovery phase and both experience a large number of EPC paging. When the block size is configured to 4 KB (the bottom two in Figure 10), which is the default and exhibits better performance for TWEEZER, Speicher suffer up to 430× more EPC paging, due to the cached MACs in the footer blocks. On the contrary, when we configure the block

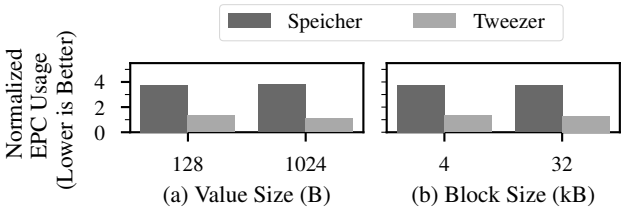


Figure 11: Normalized table cache sizes while running TWEEZER and Speicher with the r90w10 workload.

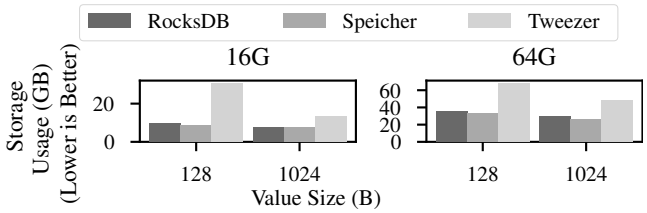


Figure 12: The amount of data stored in file system when we store the same number of key-value pairs.

size to be 32 KB, we observed that the cumulative number of EPC paging that TWEEZER experiences approaches that of Speicher, even though TWEEZER still outperforms Speicher in terms of throughput (§7.1). According to our analysis, this additional EPC paging comes from background compaction. Compared with Speicher, we observe certain periods in time in which TWEEZER's cumulative EPC paging suddenly increases. This is due to the additional memory consumption by the background compaction, which also uses EPC space to process decrypted blocks. It is worth noting that these EPC paging numbers were obtained from the runs reported in Figure 6. That is, TWEEZER still shows much higher performance albeit the EPC pagings due to the compaction. This is because the compaction does not usually block the transaction processing. TWEEZER's fine-grained authentication (§5.3) could enable compaction with encrypted SSTables and reduce these peaks, but we leave it as future work.

Amount of Data in EPC. The amount of data in EPC is another measure that shows the potential density of EPC paging over time. Programs using more EPC are likely to experience more EPC misses and longer EPC access time on average. To compare the amount of data that TWEEZER and Speicher store in EPC, we measure the size of the table cache that contains metadata for SSTables and resides in memory. The size of the table cache is a good estimate of the amount of data in EPC because the table cache is the largest component in EPC by design. Figure 11 shows the results for the workload with 90% reads as we vary the values sizes and block sizes. We observe that Speicher's table cache is 3.71∼4.17× larger compared to that of RocksDB while that of TWEEZER's is only 1.08∼1.35× larger. In other words, Speicher uses a table cache that is 2.84∼3.08× larger compared to TWEEZER. This shows that our design choice of using per-SSTable key (see

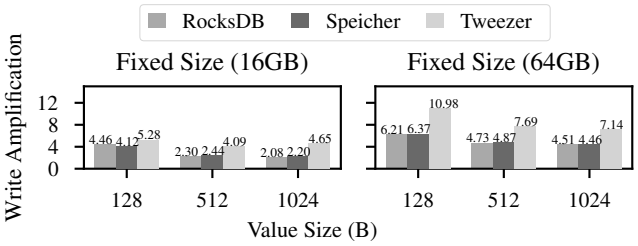


Figure 13: The amount of data that RocksDB, Speicher, and TWEEZER write to storage as the value size changes, normalized to the total size of key-value pairs that they received.

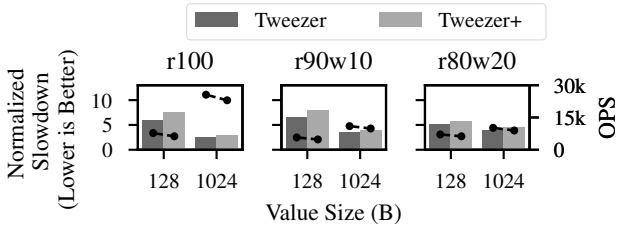


Figure 14: Throughput of TWEEZER without (TWEEZER) and with (TWEEZER+) the strong key block authentication.

§5.2) helps reduce the amount of data in EPC, contributing to reduced EPC paging.

7.3 Storage Blowup

Increased Storage Usage. One drawback of fine-grained authentication is the increase in storage usage due to the less productive compression after encryption and individual authentication for each key-value pair. As discussed in §4, TWEEZER's fine-grained authentication renders RocksDB's block compression less effective because the data is encrypted before compression, unless TWEEZER employs a specially crafted encryption and compression scheme [29, 53]. To understand this drawback quantitatively, we evaluate the corresponding storage cost by measuring the size of aggregated SSTables, with compression, constituting the KVS in varying configurations used for the evaluations in §7.1. Figure 12 shows the results, and we see that TWEEZER experiences 1.77∼3.45× storage overhead. This overhead in size increases as the value size decreases because the MAC size remains the same for each key-value pair.

Write Amplification. We also measure the amount of data that Speicher and TWEEZER write to storage and Figure 13 shows the result. We normalized the amount of written data to the number of key-value pairs that each KVS accommodates to compare their impact on write amplification. As expected, write amplification decreases as the value size increases when running unmodified RocksDB or Speicher because the amount of metadata is proportional to the number of entries. When the total size of key-value pairs is fixed, they write less metadata because they store fewer entries as the value size increases. The write amplification of TWEEZER also decreases, but much less than the other two. We presume that this is primarily due to the entropy of data in data blocks. Unlike Speicher, TWEEZER encrypts data blocks before compression, rendering compression less effective. On the evaluation with 16 GB KVS, write amplification even increases when the value size increases from 512 B to 1024 B.

8 Discussion

Data Obliviousness. TWEEZER is not designed to be data-oblivious. That is, TWEEZER is not provably immune to side-channel leakage through data-dependent access patterns. For example, attackers could learn the following information. By observing the changes of encrypted values in MemTable stored outside the EPC, attackers could learn that a write request was made and handled. However, cryptographic protection prevents the attacker from revealing or faulting the content. The relationship between leaves are also under protection in that the internal nodes are stored within EPC. Only a successful side-channel attack against the enclave [10, 33, 48, 66] could reveal such a relationship. Access patterns within an SSTable reveals relationships between the queried key and the index keys. TWEEZER does not shuffle the data blocks in an SSTable, and an attacker can determine from which data block TWEEZER found the queried key through the access pattern. Combining these two, an attacker can infer the likely range of the queried key, for example, how many keys in the SSTable would be larger than the queried key. TWEEZER could mitigate this inference by shuffling the data blocks.

Larger KVS. Our study shows that TWEEZER needs to be tailored further to have better efficiency in EPC usage when the KVS size becomes larger. TWEEZER introduces much smaller amount of additional in-memory data that must be held in EPC compared with Speicher. However, the amount of data that TWEEZER holds in EPC still increases as the KVS size increases because of some data (e.g. index block) that TWEEZER still caches within EPC, as an extension of RocksDB. We leave the optimization of RocksDB metadata to further reduce this EPC usage as future work.

Key Block Freshness. As discussed in §5.3, TWEEZER does not provably prevent the fault attack against the key block, although it is highly unlikely for an attacker to successfully perform the attack. This is due to the lack of MAC-based verification of the key blocks. As an alternative design choice, TWEEZER can be strengthened by computing and verifying MAC for the key blocks as well. Figure 14 is the result of our experiments showing the performance overhead of this design choice. As expected, the additional authentication does incur performance overhead (TWEEZER+). TWEEZER+ is

11%∼24% slower than TWEEZER depending on workloads and value sizes. The overhead increases with smaller value size because the key block size increases as the value size decreases.

9 Related work

This work is closely related to existing attempts to tailor various important applications to Intel SGX [3, 6, 12, 16, 23, 31, 54, 56, 61] as well as research on securing database systems including KVSs [15, 45].

Running Unmodified Applications on SGX. Haven [7, 8], SCONE [4], Graphene-SGX [64], Panoply [57], SGX-LKL [46] are systems designed to help unmodified applications to run on an enclave. As suggested by the authors, these enabled us to quickly work on tailoring a persistent KVS for Intel SGX. In particular, TWEEZER has been implemented and tested on SCONE. However, it is worth noting that TWEEZER can run on any of the aforementioned systems because TWEEZER does not make any assumptions on features unique to SCONE.

Persistent KVSs on SGX. As we have repeatedly discussed, Speicher [6] is the closest to our work in that it is designed to boost the performance of a persistent KVS on SGX, taking RocksDB as an example. Speicher contributes three new design features to achieve this goal, but fails to scale to large KVSs. While TWEEZER adopts many of the ideas proposed by Speicher, we propose a new message authentication scheme and restructures the data block to alleviate the scalability issue. Furthermore, TWEEZER uses a hash chain mechanism to protect persistent logs allowing for a solution that is not bound to platforms that support trusted counters. Enclage [61] is also close to TWEEZER in that it is designed to be an SGX-based secure storage engine but does not take integrity protection into account.

In-memory KVSs on SGX. ShieldStore [31] studies the design options to adapt an in-memory KVS for SGX. Compared with TWEEZER, ShieldStore is designed for in-memory KVSs and still relies on the Merkle tree for freshness. Similar to ShieldStore, EnclaveCache [12] and Avocado [5] are also designed to use SGX to protect in-memory KVSs.

Cryptographic Approaches. CryptDB is one of the pioneering systems in which unmodified database queries are proxied and handled by encrypted backend [45]. CryptDB adopts various cryptographic schemes including homomorphic encryption [20] and focuses on confidentiality guarantees. Dory goes beyond confidentiality guarantees and mitigates access pattern-based leakage, providing authenticity relying on distributed trust [15]. TWEEZER tackles the same problem at a lower level compared with these approaches in that many relational database systems use RocksDB-like persistent KVSs as storage engines. One weakness of TWEEZER, when compared to Dory, is the lack of data obliviousness. To overcome this, TWEEZER has to be strengthened with oblivious search indices [40] or file system operations [2].

Log Protection. Protection of logs from rollback attacks have long been an important problem. One of the well-known mechanisms is the hash chain [44, 51, 52] that TWEEZER adopts to protect the WAL and MANIFEST logs. However, the hash chain cannot guarantee freshness against potential rollback attacks across crashes and recoveries as discussed in Memoir [44]. Memoir overcomes this limitation and relies on local trusted non-volatile memory. Verena [30] also addresses a similar problem by using a hash server. Compared with these, TWEEZER's approach is similar to Verena in that it relies on the user, who sends heartbeat transactions to timestamp versions. ROTE is designed solely to address this weakness of requiring a trusted component to defeat the rollback attack by using multiple enclaves [37]. TWEEZER can adopt this to provide rollback resilience without relying on the heartbeat packets.

10 Conclusion

This paper presented TWEEZER, an LSM tree-based persistent key-value store tailored for confidential computing by taking advantage of the LSM tree design principles. The unique invariants that the LSM tree introduces, being a data structure optimized for storage devices, enables TWEEZER to avoid constructing a large Merkle tree to protect the integrity and freshness of the key-value pairs. Our experiments with the implementation of TWEEZER and a reproduction of a pioneering work, Speicher, shows that this new MAC scheme for the LSM tree brings considerable performance benefits. Our implementation of TWEEZER outperforms Speicher on point lookups (e.g., by 1.91∼6.23×) in all evaluation settings, and in particular, the ones with large (16∼64 GB) KVSs. We anticipate that our findings and open-sourced implementation from this work will motivate further improvements in this direction to secure our data on these key-value stores.

Acknowledgment

We thank the anonymous reviewers and our shepherd, Patrick P. C. Lee, for their constructive reviews and comments. This work was partly supported by 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), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2021-0-01817, Development of Next-Generation Computing Techniques for Hyper-Composable Datacenters), and the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. NRF-2021R1F1A1050311).

Availablity

TWEEZER is available on https://github.com/cssl-unist/tweezer.

References

[1] The storage performance development kit (spdk). https://spdk.io/.

[2] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[3] Jinwoo Ahn, Junghee Lee, Yungwoo Ko, Donghyun Min, Jiyun Park, Sungyong Park, and Youngjae Kim. Diskshield: A data tamper-resistant storage for intel sgx. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 799–812, 2020.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. Scone: Secure linux containers with intel sgx. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[5] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. Avocado: A secure in-memory distributed storage system. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, July 2021.

[6] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. Speicher: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST)*, February 2019.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–26, 2015.

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.*, 33(3), August 2015.

[9] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, pages 1–13, 2016.

[10] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[11] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyperthreading side channels on sgx with contrived data races. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 178–194. IEEE, 2018.

[12] Lixia Chen, Jian Li, Ruhui Ma, Haibing Guan, and Hans-Arno Jacobsen. Enclavecache: A secure and scalable key-value cache in multi-tenant clouds using intel sgx. In *Proceedings of the 20th International Middleware Conference*, pages 14–27, 2019.

[13] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 7–18, 2017.

[14] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[15] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. Dory: An encrypted search system with distributed trust. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1101–1119, 2020.

[16] Susanne Felsen, Ágnes Kiss, Thomas Schneider, and Christian Weinert. Secure and private function evaluation with intel sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 165–181, 2019.

[17] The Apache Software Foundation. Kafka streams. https://kafka.apache.org/.

[18] Uber San Francisco. Uber. https://www.uber.com/.

[19] Blaise Gassend, E Suh, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Caches and merkle trees for efficient memory authentication. In *Proceedings of ninth international symposium on high performance computer architecture*, 2003.

[20] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. Association for Computing Machinery.

[21] Greg. Platform service enclave and me for intel xeon server. https://community.intel.com/t5/Intel-Software-Guard-Extensions/Platform-Service-Enclave-and-ME-for-Intel-Xeon-Server/td-p/1173098.

[22] Shay Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptol. ePrint Arch.*, 2016:204, 2016.

[23] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. Sgx-box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 99–105, 2017.

[24] Intel. 10th generation intel core processor famillies datasheet volume 1. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/10th-gen-core-families-datasheet-vol-1-datasheet.pdf.

[25] Intel. Enclave memory measurement tool for intel software guard extensions (intel sgx) enclaves. https://software.intel.com/content/dam/develop/external/us/en/documents/enclave-measurement-tool-intel-sgx-737361.pdf.

[26] Intel. Intel sgx. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html.

[27] Intel. The intel sgx memory encryption engine. https://software.intel.com/content/www/us/en/develop/blogs/memory-encryption-an-intel-sgx-underpinning-technology.html.

[28] Intel. Unable to find alternatives to monotonic counter application programming interfaces (apis) in intel software guard extensions (intel sgx) for linux* to prevent sealing rollback attacks. https://www.intel.com/content/www/us/en/support/articles/000057968/software/intel-security-products.html.

[29] M. Johnson, P. Ishwar, V. Prabhakaran, D. Schonberg, and K. Ramchandran. On compressing encrypted data. *IEEE Transactions on Signal Processing*, 52(10):2992–3006, 2004.

[30] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-end integrity protection for web applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[31] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[32] Kevin Lahey. Monitoring intel sgx enclaves. https://fortanix.com/blog/2020/02/monitoring-intel-sgx-enclaves/.

[33] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, August 2017.

[34] Percona LLC. Mongorocks. https://www.percona.com/doc/percona-server-for-mongodb/3.4/mongorocks.html.

[35] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2016.

[36] MariaDB. Getting started with myrocks. https://mariadb.com/kb/en/getting-started-with-myrocks/.

[37] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. {ROTE}: Rollback protection for trusted execution. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1289–1306, 2017.

[38] Ines Messadi, Shivananda Neumann, Lennart Almstedt, and Rüdiger Kapitza. A fast and secure key-value service using hardware enclaves. In *Proceedings of the 20th International Middleware Conference Demos and Posters*, pages 1–2, 2019.

[39] Microsoft. mi-malloc. https://microsoft.github.io/mimalloc/.

[40] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[41] Netflix. Netflix. https://www.netflix.com/.

[42] Patrick O' Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O' Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[43] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, RS, April 2017.

[44] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings*

of the 32nd IEEE Symposium on Security and Privacy (Oakland), Oakland, CA, May 2011.

[45] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100, 2011.

[46] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. Sgx-lkl: Securing the host os interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[47] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[48] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, May 2021.

[49] et al. Rich Felker. musl libc. https://musl.libc.org/.

[50] Apache Samza. Apache samza. http://samza.apache.org/.

[51] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th USENIX Security Symposium (Security)*, San Antonio, TX, January 1998.

[52] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security (TISSEC)*, 2(2):159–176, 1999.

[53] Daniel Schonberg, Stark C. Draper, and Kannan Ramchandran. On blind compression of encrypted data approaching the source entropy rate. In *2005 13th European Signal Processing Conference*, pages 1–4, 2005.

[54] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[55] Scone. Scone sgx toolchain. https://sconedocs.github.io/SCONE$_{t}$oolchain/.

[56] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing NFV states by using SGX. In *Proceedings of the 1st ACM International Workshop on Security in SDN and NFV*, New Orleans, LA, March 2016.

[57] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*, 2017.

[58] Facebook Open Source. Rocksdb. https://rocksdb.org/, n.d.

[59] Facebook Open Source. A rocksdb storage engine with mysql. http://myrocks.io/, n.d.

[60] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure integrated circuits and systems*, pages 27–42. Springer, 2010.

[61] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. Building enclave-native storage engines for practical encrypted databases. *Proceedings of the VLDB Endowment*, 14(6):1019–1032, 2021.

[62] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. rkt-io: a direct i/o stack for shielded execution. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 490–506, 2021.

[63] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[64] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 645–658, 2017.

[65] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[66] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Breaking virtual memory protection and the sgx ecosystem with foreshadow. *IEEE Micro*, 39(3):66–74, 2019.

[67] Jinwen Wang, Yueqiang Cheng, Qi Li, and Yong Jiang. Interface-based side channel attack against intel sgx. *arXiv preprint arXiv:1811.05378*, 2018.

[68] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017.

[69] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David HC Du. Ac-key: Adaptive caching for lsm-based key-value stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, July 2020.

Practicably Boosting the Processing Performance of BFS-like Algorithms on Semi-External Graph System via I/O-Efficient Graph Ordering

Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang
The Chinese University of Hong Kong

Abstract

As graphs continue to grow to have billions of vertices and edges, the attention of graph processing is shifted from in-memory graph system to external graph system. Of the two the latter offers a cost-effective option for processing large-scale graphs on a single machine by holding the enormous graph data in both memory and storage. Although modern external graph systems embrace many advanced I/O optimization techniques and can perform well in general, graph algorithms that build upon Breadth-First Search (BFS) (a.k.a. BFS-like algorithms) still commonly suffer poor processing performance. The key reason is that the recursive vertex traversal nature of BFS may lead to poor I/O efficiency in loading the required graph data from storage for processing.

Thus, this paper presents I/O-Efficient Graph Ordering (IOE-Order) to pre-process the graph data, while better I/O efficiency in loading storage-resident graph data can be delivered at runtime to boost the processing performance of BFS-like algorithms. Particularly, IOE-Order comprises two major pre-processing steps. The first is Breadth-First Degree-Second (BFDS) Ordering, which exploits both graph traversal pattern and degree information to store the vertices and edges which are most likely to be accessed together for I/O efficiency improvement. The second is Out-Degree Binning, which splits the BFDS-ordered graph into multiple sorted bins based on out-degrees of vertices so as to 1) further increase I/O-efficiency for runtime graph processing and 2) deliver high flexibility in pre-caching vertices based on the memory availability. In contrast to the state-of-the-art pre-processing techniques for BFS-like algorithms, IOE-Order demonstrates better efficiency and practicability: It delivers higher processing performance by achieving higher I/O efficiency but entails much lower pre-processing overhead.

1 Introduction

Breadth-First Search (BFS) is the foundation of many popular and important graph algorithms (a.k.a. BFS-like algorithms) that share a common feature called recursive graph traversal. That is, given a starting set of vertices, their adjacent vertices (i.e., neighbors) will be explored recursively until all the connected vertices are visited. Due to this feature of exploration, BFS-like algorithms are useful in various domains, such as networking [13], bioinformatics [20, 32], social media [8, 23, 51], and others [24, 31, 38]. In addition, based on the survey [42], BFS-like algorithms are popular. Particularly, among 13 typical graph algorithms, Connected-Component is most widely used, and Shortest-Path and Betweenness-Centrality are also within the top five: They are all BFS-like. Moreover, the BFS-like recursive graph traversal also plays a critical role in many important graph mining algorithms such as Subgraph Searching and Pruning [26, 36].

However, BFS-like algorithms generally have poor locality of access. Specifically, compared with other graph algorithms such as PageRank [18, 39] and Sparse Matrix-Vector Multiplication [29] where all vertices are regularly visited, BFS-like algorithms only visit a subset of vertices at any given time. More seriously, it is very challenging to predict how the vertices are going to be visited given the fact that the BFS can start with any vertex. As a result, even till today, how to efficiently process graphs using BFS-like algorithms continues drawing a lot of attention in both academia and industry.

On the other hand, as graphs continue to grow and cannot fit in the memory of a machine, people start to leverage the massive storage to keep the enormous graph data for graph processing at low cost. Among several feasible solutions (which will be presented in Section 2.1 in details), the *semi-external graph system* is a popular option that demonstrates its capability of efficiently processing the large-scale graph in a single machine [53]. Due to the complex relationships (i.e., edges) among entities (i.e., vertices) in real-world graphs, the number of edges is typically significantly larger than that of vertices [53]. Thus, semi-external graph systems propose to keep the large-sized edge data in the massive storage for holding a large-scale graph at low cost, but maintain the small-sized vertex data in the faster memory for offering better performance of graph processing (that typically generates lots of small and random updates to the attributes of

vertices). Fortunately, since the memory space in commodity PCs nowadays is generally large enough to hold the vertices of most of large-scale graphs [2], semi-external graph system is regarded as a cost-effective model in graph processing and several excellent semi-external graph systems have been developed [25, 29, 44, 53].

To further improve the performance of loading edges from the slower storage, various effective I/O optimization techniques have been suggested and integrated in modern semi-external graph systems (which will be introduced in Section 2.1 in detail). However, these general techniques could only bring limited improvement to BFS-like algorithms due to the lack of consideration of the BFS's recursive graph traversal nature. Thus, Lee et al. try to tackle the poor performance issue of BFS-like algorithms via *pre-processing optimizations* of *ordering* and *pre-caching*. Specifically, ordering is a technique to re-order the graph to improve the locality of access, whereas pre-caching is to pre-load the data in memory which will not be changed during the entire execution (see Section 2.2 for details). Nevertheless, based on our evaluations, their designs for BFS-like algorithms still leave a substantial room for improvement due to the limited *I/O efficiency*; furthermore, they may even suffer the critical issue of limited practicability for spending considerable time on pre-processing compared to the improvement that they bring (see Section 2.3 for details).

To boost the processing performance of BFS while deliver high practicability, this paper proposes *I/O-Efficient Graph Ordering (IOE-Order)*, which comprises two steps to pre-process the graph, while better I/O efficiency in loading edge data can be achieved during graph processing. The first step is called Breadth-First Degree-Second (BFDS) Ordering. Specifically, BFDS not only exploits the graph traversal pattern to capture the global structure of a graph, but also, based on the global structure, keeps the neighbors of high in-degree vertices together so that more I/O requests with high I/O efficiency can be issued during graph processing.

The second step is Out-Degree (OutD) Binning. Under BFDS-ordered graph, OutD Binning further splits the edge data into multiple bins based on the sizes of edge lists (i.e., out-degrees of vertices). Additionally, all the bins are sorted and stored sequentially on the graph according to their average out-degrees. In this way, the vertices of small out-degree can be physically separated and then efficiently pre-cached, while loading data from the rest of bins could enjoy much higher I/O efficiency. Furthermore, the design of multiple bins in the graph provides high flexibility. That is, semi-external graph systems can easily pre-cache edge data starting from the bin with the smallest average out-degree based on the different amounts of memory in various machines.

We implement IOE-Order in C++ and evaluate its effectiveness on Graphene [29], which is an open-sourced, state-of-the-art semi-external graph system. In particular, we enrich Graphene to support pre-caching. Our evaluations based on billion-scale graphs reveal that, compared with the integrated solution of state-of-the-art pre-processing optimizations [28], IOE-Order delivers better efficiency and practicability. In terms of efficiency, IOE-Order can improve the processing time of various BFS-like algorithms by 18.8% on average and even up to 36.1%, thanks to its efficacy in increasing/optimizing I/O efficiency from 70.9% to 82.1% on average and even up to 98.8%. As for the practicability, IOE-Order entails much lower (i.e., 815.2× lower) online pre-processing overhead by enabling a flexible and efficient way to pre-cache edges with a holistic graph ordering.

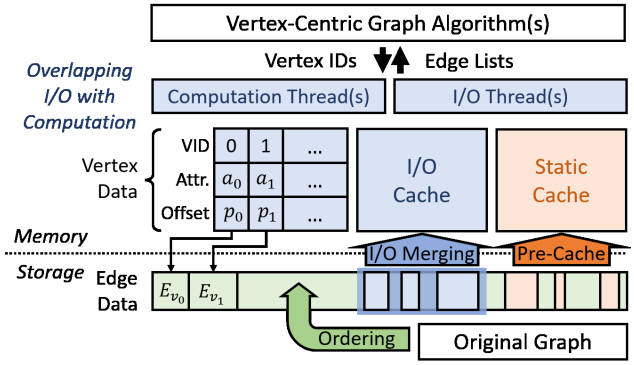
The rest of this paper is organized as follows: Section 2 presents the background and motivation regarding this work. Section 3 introduces the main design of I/O-Efficient Graph Ordering. Next, Section 4 demonstrates the evaluation results. Finally, Section 5 discusses the related work and Section 6 concludes this work.

2 Background and Motivation

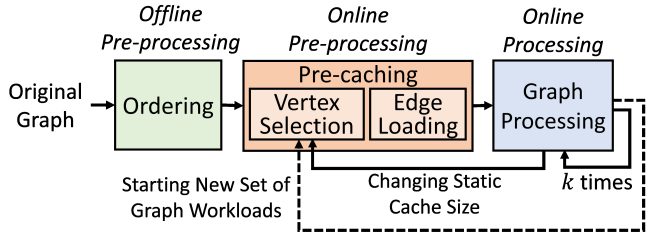
2.1 Semi-External Graph Processing

A graph generally comprises two sets of data: *vertex data* that consist of a set of vertices along with vertex attributes and *edge data* that describe the set of edges linking two vertices along with edge properties. That is, an edge describes the neighboring relationship of two connected vertices, and for an edge $e = (u, v)$ in a directed graph, v is referred to as the *out-neighbor* of u while u is referred to as the *in-neighbor* of v. The terms *out-degree* and *in-degree* further indicate the number of out-neighbors and in-neighbors for a given vertex respectively. In practice, in the vertex data, each vertex is assigned with a distinct value, called *vertex ID*, for identification purpose. Edge data, on the other hand, represent all the edges in the form of *edge list* that enumerates the neighbors' vertex IDs for a specific vertex, and all the edge lists are further sorted by vertex IDs. Thus, the edge list of a given vertex in the edge data can be easily indexed by the vertex ID.

As depicted in Figure 1(a), the semi-external graph system is introduced to cost-effectively process the graph data by 1) keeping the small-sized but frequently-updated vertex data in the faster memory while 2) storing the large-sized but mostly-read-only edge data in the cheaper storage. Particularly, in modern semi-external graph systems [25, 29, 44, 53], a vertex's attribute and file offset to its edge list are maintained in memory, and the entire edge data are stored in storage as file(s). On the other hand, the modern semi-external graph systems usually support the *push-style, vertex-centric programming model* [33] because of its ability to express a lot of graph algorithms and its ease for distributed and parallelized execution [21]. Under such programming model, graph algorithms are designed to iteratively specify and activate a subset of vertices (a.k.a. *active vertices*) that need to be processed in the following iteration. Thus, the modern semi-external graph systems typically provide the API to load all the edge lists of



(a) Typical System Architecture of Semi-External Graph System.



(b) Graph Processing Flow with Ordering and Pre-Caching.

Figure 1: An Overview of Semi-External Graph System.

active vertices from the edge data, so that graph algorithms can, based on the loaded edge data, efficiently update the vertex attributes in memory and generate a new set of active vertices for the next iteration of processing.

As presented in Figure 1(a) as well, to further improve the efficiency of loading edge data from storage, from bottom to top, modern semi-external graph systems also introduce several general I/O optimization techniques as follows:

I/O Merging. Solid-state drives (SSDs), which are adopted in most state-of-the-art semi-external graph systems, deliver better I/O throughput under I/O requests of larger sizes [29, 53]. Therefore, when the I/O requests are issued to retrieve the blocks in SSD, several small requests are typically merged into a large request for higher I/O throughput [29, 53]. This technique is called I/O merging. Taking FlashGraph [53] as an example, it merges 4KB I/O requests to consecutive blocks if possible, so an I/O request actually issued by FlashGraph could typically range from 4 KB up to many MBs. By contrast, Graphene [29] issues 512 B I/Os and aggressively merges them to close (but not necessary to be consecutive) blocks and forms a larger I/O request (up to 16 KB) and submits a great amount of asynchronous I/Os to saturate I/O throughput.

I/O Cache. After the completion of I/O requests, the loaded edge lists are typically kept in the I/O cache, which is a small amount of user-space memory managed by the semi-external graph system, and wait for being processed. Different systems usually have their strategies to manage the I/O cache. For instance, FlashGraph [53] adopts traditional page cache management strategy (i.e., g-clock algorithm) to evict the less-frequently-accessed data from the I/O cache. On the other hand, since Graphene [29] issues fine-grained 512 B I/Os, it directly discards the loaded edge lists, instead of keeping them for future use, in I/O cache after being processed to better utilize I/O cache.

Overlapping I/O with Computation. The mainstream semi-external graph systems typically overlap I/O with computation to have a significant improvement in performance. To realize this functionality, *asynchronous I/O* is one key technique since it can allow a thread to do the computation while there are several ongoing I/Os in the background [53]. Another key technique is to leverage the *multi-thread programming* that enables the separation of computation jobs and I/O jobs in different threads so that both type of jobs can be done parallelly [29].

2.2 Pre-processing for BFS-like Algorithms

Although modern semi-external graph systems integrate several general I/O optimization techniques, BFS-like algorithms are still notorious for their poor processing performance [28]. The reason is twofold: First, the I/O optimization techniques introduced in Section 2.1 are for general graph algorithms. Therefore, their designs do not particularly favor the recursive graph traversal nature of BFS. Second, real-world graphs usually have irregular structure and follow power-law distribution [16]. In other words, the edge lists of a vertex's neighbors tend to be scattered across the edge data, and their actual sizes are much smaller than the block granularity of I/O requests.

To alleviate the poor processing performance of BFS-like algorithms on semi-external graph system, Lee et al. look for the opportunity of *pre-processing* the graph data [28]. As shown in Figure 1(b), before processing the graph by BFS-like graph algorithms, they propose to first pre-process the graph by two stages: *ordering* and *pre-caching*.

Ordering. In general, ordering is a common technique to convert a graph into a new one by re-assigning vertex IDs and re-ordering the edge lists in the edge data based on the newly assigned vertex IDs [28]. Due to this nature, typically, ordering only needs to be applied once per graph and can be done by using a powerful server. Thus, we refer to the ordering stage as *offline pre-processing*.

In contrast to the existing ordering techniques that are mainly designed for in-memory graph systems [33, 37], Lee et al. introduce *Neighborhood Ordering (Norder)* [28] to achieve better access locality for semi-external graph systems. Its objective is to assign neighboring vertices with close vertices IDs so that the edge lists of neighboring vertices can be thereby stored closely in the edge data for the reduction of I/O cost. In practice, Norder minimizes the standard deviation of the neighboring vertex IDs by performing the BFS with depth level bound [28] starting from the highest in-degree vertex. The depth level is bounded to two because they empirically found it to be effective for overall performance.

Pre-caching. Static cache is a common technique, which can be found in many system designs [15, 30, 47], to reduce the number of issued I/Os. Unlike traditional cache which replaces data upon cache miss, the data in static cache are pre-loaded and will not be evicted during the whole execution.

Lee et al. utilize the static cache to selectively pre-cache some edge data before processing the graph [28]. Since a graph is usually analyzed many times (i.e., number k in Figure 1(b)) for obtaining various information [52], the pre-cached data can benefit the graph processing for multiple rounds until all the graph workloads are completed. Please note that, since the edge data must be adaptively pre-cached based on the static cache size and graph workloads at runtime, the pre-caching stage needs to be re-performed whenever the available memory space for static cache changes or a new set of graph workloads launches. Thus, we refer to the pre-caching stage as *online pre-processing*.

As illustrated in Figure 1(b), the pre-caching stage can be further divided into two steps: *vertex selection* and *edge loading*. Particularly, the step of vertex selection has a direct impact on how I/Os can be reduced, since this step determines which edge data concerned with the selected vertices are going to be pre-cached (from the storage) during the step of edge loading. Thus, to reduce small and random I/Os during graph processing, Lee et al. propose *Greedy Vertex Selection (GVS)* to pick out the vertices whose edge lists are not stored in the same I/O block as their siblings (i.e., vertices sharing the same in-neighbor) [28].

2.3 Motivation: I/O Efficiency of BFS-like Algorithms

Despite the fact that Norder and GVS indeed achieve noticeable performance improvement for BFS-like algorithms, the question of *how close we are from the optimal processing performance* still remains. Thus, this section will answer this question, through a series of theoretical modelling and practical evaluations, from a new perspective: *I/O efficiency*.

Since BFS-like algorithms generally show higher demands for I/O than computation [29] and modern semi-external graph systems typically overlap the I/O with computation [29, 53], the I/O performance basically dominates the overall processing performance of BFS-like algorithms. Thus, the *processing performance* (denoted as *Proc. Perf.*) of BFS-like algorithm on a semi-external graph system can be first expressed as Equation 1: That is, the processing performance is proportional to the number of bytes actually processed by the BFS-like algorithm (denoted as *Proc. Bytes*) but is in inverse propotation to the total time spent on I/O (denoted as denotes the number of bytes actually processed by the BFS-like algorithm and *I/O Time*).

$$\textit{Proc. Perf.} \propto \frac{\textit{Proc. Bytes}}{\textit{I/O Time}} = \frac{\textit{Trans. Bytes}}{\textit{I/O Time}} \times \frac{\textit{Proc. Bytes}}{\textit{Trans. Bytes}} \tag{1}$$

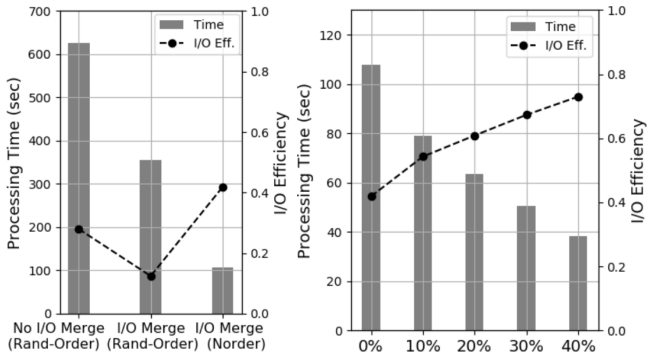
Where:

$$\textit{I/O Throughput} = \frac{\textit{Trans. Bytes}}{\textit{I/O Time}}, \tag{2}$$

$$\textit{I/O Efficiency} = \frac{\textit{Proc. Bytes}}{\textit{Trans. Bytes}}. \tag{3}$$

If we introduce the total number of transferred bytes (denoted as *Trans. Bytes*) into Equation 1, the processing performance can be further expressed into the product of two critical components: *I/O throughput* and *I/O efficiency*. As expressed in Equations 2 and Equation 3, I/O throughput represents the total number of transferred bytes (*Trans. Bytes*) within the total time spent on I/O (*I/O Time*), while I/O efficiency indicates the ratio of the total number of processed bytes (*Proc. Bytes*) to the total number of transferred bytes (*Trans. Bytes*).

From Equation 1, we can learn that *the key to optimize processing performance is by simultaneously maintaining high I/O throughput and high I/O efficiency*. However, in practice, there usually exists a trade-off between them. Taking Graphene [29] as an example, on one hand, it proposes to exploit fine-grained I/O granularity to avoid low I/O efficiency, but on the other hand, leverages I/O merging to achieve high I/O throughput. Thus, to see how the I/O merging affects the overall processing performance of BFS-like algorithms in practice, Figure 2(a) presents the results of performing a BFS algorithm on Graphene [29] using the large-scale uk2007 graph [48], which comprises nearly four billions of edges. In this figure, the x-axis indicates whether the I/O merging is enabled and the ordering algorithm used to re-order the evaluated graph, while the y-axis demonstrates the overall processing performance (in terms of the total processing time) and the I/O efficiency. It can be clearly observed that although the I/O merging can effectively reduce the total processing time by 43%, it also degrades the I/O efficiency by 12.4% when the evaluated graph is randomly re-ordered (denoted as Rand-Order).



(a) I/O Merging and Ordering. (b) Norder and GVS (with I/O Merging).

Figure 2: A Series of Evaluations of Running BFS Algorithm on Graphene [29] with `uk2007` Graph [48].

Figure 2(a) also reveals how the state-of-the-art Norder algorithm helps to improve the I/O efficiency with the I/O

merging enabled. Particularly, we can clearly observe that Norder indeed shows its efficacy to further reduce the total processing time by 69.7% against Rand-Order. Nevertheless, the actual I/O efficiency under Norder is still quite low, which is only 41.9%; that is, almost 60% of the bytes are loaded but useless over all the transferred bytes.

To further understand whether the state-of-the-art GVS can help improving the I/O efficiency, we vary the static cache size to be 0% (i.e., no static cache), 10%, ..., to 40% of the edge data size in uk2007 graph [48] and have the graph re-ordered by Norder. As shown in Figure 2(b), as the static cache size keeps increasing, GVS can not only keep decreasing the total processing time but also have the potential to improve the I/O efficiency. This is because GVS aims to reduce the number of small and random I/Os during graph processing. In other words, when more edge data are pre-cached by GVS in a static cache of larger sizes, more small and random I/Os to the pre-cached edge data can be completed in the faster static cache for preventing incurring I/Os with low I/O efficiency. Nevertheless, such improvement comes at a huge cost and demand for the static cache size. As shown in Figure 2(b), only when the static cache size is up to 40% of the evaluated edge data size, a nearly 75% (specifically, 73.1%) of the I/O efficiency can be achieved eventually. This not only exposes the ineffectiveness of GVS in utilizing the static cache, but even makes semi-external graph system still costly to process BFS-like algorithms on large-scale graphs.

Table 1: Pre-processing Time of Norder and GVS (seconds).

Static Cache Size	Ordering (Norder)	Vertex Selection (GVS)	Edge Loading (based on GVS)
10%	51.7	2,914.7	16.3
20%		4,708.3	17.6
30%		6,061.0	18.7
40%		7,053.4	19.6

There is even one more thing that may further limit the practicability of the existing the pre-processing techniques, particularly GVS, if the pre-processing overhead is considered. Table 1 shows the pre-processing time of Norder, GVS, and the edge loading time based on GVS. It can be clearly observed that, although the pre-processing time of Norder is about 51.7 seconds, the ordering stage is actually an offline preprocessing optimization which only introduces an one-shot overhead. By contrast, GVS, which is online pre-processing optimization, requires up to 7,053 seconds (which is almost 67.17× of the total processing time of BFS) when the static cache size is as large as 40% of the edge data. Given the fact that GVS needs to be re-performed whenever the size of static cache changes or a new set of graph workloads launches, such high time complexity may make GVS fail to reduce the end-to-end graph processing time especially when the number of workloads (i.e., number k in Figure 1(b)) is not large enough to amortize the huge overhead of GVS.

3 I/O-Efficient Graph Ordering

3.1 Overview

Based on the investigations presented in Section 2.3, we realize that optimizing the I/O efficiency of loading storage-resident edge data is crucial to boost the processing performance of BFS-like algorithms on semi-external graph system. Thus, this section introduces a new *I/O-Efficient Graph Ordering* that not only enables higher I/O efficiency for better processing performance by pre-processing the graph but also demonstrates great practicability by entailing low pre-processing overhead.

As shown in Figure 3, the proposed IOE-Order consists of two major steps to re-order the graph. The first step is *Breadth-First Degree Second (BFDS) Ordering* (see Section 3.2) that exploits both graph traversal pattern and in-degrees of vertices to re-assign the vertex IDs such that the edge lists of vertices, which are most likely to be together traversed by BFS-like algorithms, can be thereby closely stored in the edge data. This step can guarantee that higher I/O efficiency in loading the edge data involved in graph traversals can be achieved, even when the I/O merging is also enabled for high I/O throughput.

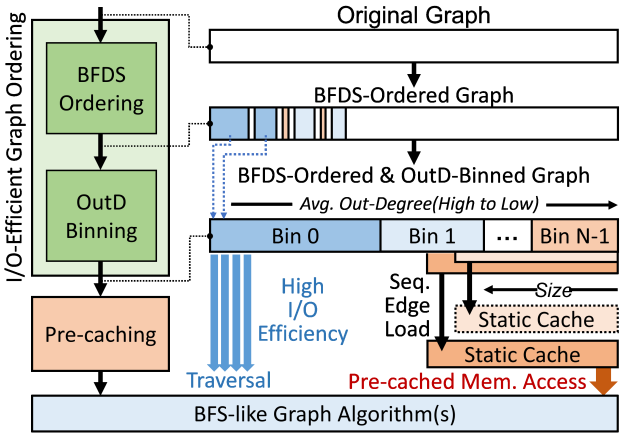


Figure 3: An Overview of I/O-Efficient Graph Ordering.

The second step, *Out-Degree (OutD) Binning* (see Section 3.3), is then introduced to split the BFDS-ordered graph into multiple bins based on the sizes of edge lists (out-degree), while edge lists within each bin are still sorted based on the BFDS-suggested order. Moreover, all these bins are sorted and stored sequentially on the edge data based on the their average out-degrees in descending order to form the final graph ordering. This step ensures that the small edge lists can be physically separated and then pre-cached, so that the I/O efficiency of loading those bins of large edge lists can be thereby improved.

Last but not the least, as illustrated in Figure 3, the final BFDS-ordered and OutD-binned graph also suggests a flexible and efficient way to utilize the static cache. Particularly,

we can easily and sequentially load the edge lists of bin(s) in the reverse order based on the available size of static cache. In other words, the final graph is a holistic graph ordering which cleverly embeds the suggestion of vertex selection to completely eliminate the online pre-processing overhead of selecting vertices for pre-caching edges.

3.2 Breadth-First Degree-Second Ordering

3.2.1 Design Concept

The state-of-the-art ordering (i.e., Norder [28]) for BFS-like algorithms mainly relies on the intuition that the neighboring vertices will be typically traversed together. This intuition, although it is generally correct, it may overlook the importance of graph traversal nature of BFS. Particularly, in contrast to the neighboring information which could only provide the local information regarding vertices, the graph traversal pattern can offer the global and structural view about the whole graph and imply that which vertices may have higher probabilities to be traversed earlier/late than others.

To examine how the graph traversal pattern can help with the improvement in I/O efficiency, let us consider a typical BFS starting with a given vertex v^*. Suppose that the graph is "coincidentally" re-ordered according to the BFS traversal pattern starting with v^*, the I/O efficiency in performing the BFS (starting with the vertex v^*) can theoretically reach the optimal (i.e., 100%). In such case, the I/O accesses to the storage-resident edge data will be like sequentially sliding over the storage space from the smallest vertex ID to the largest vertex ID.

However, the optimal ordering might not always be the case in practice, because BFS can start with any vertex. Fortunately, we observe that graph traversal pattern, along with the in-degree information of vertices, can provide us the following probabilistic hints about how vertices are going to be traversed earlier/late than others. First, high in-degree vertices have higher possibility to be visited earlier than low in-degree ones. Second, during graph traversal, the vertices which are recursively connected to high in-degree vertices are also likely to be traversed earlier. Thus, not only the high in-degree vertices but also the vertices recusively connected to high in-degree vertices shall be best stored with their respective neighbors closely in storage for delivering higher I/O efficiency.

Based on the above insights, we introduce the *Breadth-First Degree-Second (BFDS) Ordering* that performs the graph traversal with the consideration of the in-degree information of vertices to order a graph. Particularly, BFDS performs the graph traversal, starting with the highest in-degree vertex (which demonstrates the highest probability of being traversed earlier) but iteratively traverses the out-neighbors based on their in-degrees. In contrast to the conventional BFS, BFDS further sorts the active vertices in an iteration according to their in-degrees in descending order so that the out-neighbors of the "sorted" active vertices are assigned with consecutive vertex IDs in order. In a nutshell, BFDS not only orders the vertices based on the graph traversal pattern (i.e., "Breadth-First") but further gives higher priority to keep the neighbors of high in-degree vertices together (i.e., "Degree-Second"). Consequently, BFDS maximally keeps vertices which are likely to be traversed together, making the I/Os issued by BFS-like algorithms enjoy higher I/O efficiency.

3.2.2 Design Details

Algorithm 1 shows the proposed BFDS ordering. Lines 1-3 are for the initialization of BFDS. Specifically, Line 1 finds out the maximum in-degree vertex, and push it to be starting vertex in Line 2. Line 3 is the new re-assigned vertex ID.

Algorithm 1 Breadth-First Degree-Second Ordering

Input: Graph $G = (V, E)$;
Output: Mapping function Map

```
1:  Find out v_max_in to be the max in-degree vertex
2:  Active ← {v_max_in}
3:  NewID ← 0
4:  while Active is not empty do
5:      Sort Active based on in-degree in descending order
6:      for v ∈ Active do
7:          for u ∈ v.neighbors & u is not visited do
8:              NextActive ← u
9:              Map.add(u, NewID)
10:             NewID ← NewID + 1
11:         end for
12:     end for
13:     Active ← NextActive
14: end while
15: Assign the rest of unvisited vertices new IDs
```

Lines 4-14 are the core function of BFDS. To begin with, Line 5 will sort all the active vertices (denoted as *Active*) based on their in-degree in descending order. Line 6 iterates all the vertices from the highest in-degree one and tries to assign consecutive IDs to its un-visited neighbors from Lines 7-11. Particularly, if a neighboring vertex is not assigned to a new ID yet, it gets a new ID and becomes the active vertex of next iteration (denoted as *NextActive*) in Line 8. Finally, the while loop will end if *Active* is empty. Nevertheless, several isolated vertices that cannot be reached by the graph traversal are still not assigned with new IDs. As a result, Line 15 will assign the rest of unvisited vertices with new IDs.

As we can see, the time complexity of BFDS ordering is $O(|E|+|V|)$ combined with the overhead incurred by Line 5. Suppose the total iteration is n and there are V_i active vertices in iteration i, The total number of visited vertices is $\sum_{i=1}^{n} V_i \leq V$. The time incurred by Line 5 shows in the following.

$$\sum_{i=1}^{n} V_i \log V_i < \sum_{i=1}^{n} V_i \log V \leq V \log V$$

Therefore, the time complexity of BFDS is $O(|E|+|V|+|V|\log|V|)$. On the other hand, the time complexity of Norder [28] is roughly $O(|E|+|V|)$, which is slightly faster than BFDS. Nevertheless, BFDS shows significant improvement in performance against Norder in Section 4. Furthermore, ordering only needs to be done once per graph and can be completed in offline [50]. We believe BFDS is an effective ordering overall in practice.

3.3 Out-Degree Binning

3.3.1 Design Concept

Although BFDS improves the I/O efficiency of BFS-like algorithms, it is still hard to reach the optimum due to the irregular structure of real-world graphs. To further optimize I/O efficiency while offering the efficient pre-caching, we propose a simple yet effective design, namely *Out-Degree (OutD) Binning*, to refine the BFDS-ordered graph. Its key idea is to isolate those edge lists, which might be harder to be properly re-ordered by the BFDS Ordering and could be the root cause of lower I/O efficiency; however, the BFDS-suggested ordering is still maximally preserved within each bin to have high I/O efficiency of loading the rest of edge lists.

Our intuition, in practice, is that *the I/O efficiency of loading a vertex's edge list might relate to its number of out-neighbors (out-degree)*, since the out-neighbors of an active vertex will typically be processed by BFS-like algorithms at a time, which thereby leads to larger number of processed bytes (i.e., numerator) in Equation 3. By contrast, as the issued I/O size is generally large for achieving higher bandwidth, loading smaller number of out-neighbors of an active vertex from storage may most likely result in lower I/O efficiency.

With such intuition in mind, Figure 4 illustrates why and how OutD Binning avoids low I/O efficiency. Particularly, we consider a simple scenario that the BFDS-ordered graph is split into two bins: Large-OutD bin (for containing larger edge lists such as E_{L_0} and E_{L_1}) and Small-OutD bin (for containing smaller edge lists such as E_{S_0} and E_{S_1}), and the edge lists of Small-OutD bin are pre-cached in the static cache. In contrast to the existing GVS which may just simply pre-cache E_{S_0} and E_{S_1} into the static cache (without binning them), the potential benefits of OutD Binning are twofold: First, since E_{S_0} and E_{S_1} are already pre-cached in the static cache, when loading the block containing E_{L_0}, OutD Binning effectively avoids the redundancy in loading E_{S_0}; instead, E_{L_1} can be actively loaded with E_{L_0} to achieve higher I/O efficiency, since E_{L_0} and E_{L_1} are supposed to be traversed together based on the BFDS ordering. Second, OutD Binning enables a more efficient pre-caching process. That is, rather than executing time-consuming vertex selection and loading the edge lists randomly and inefficiently (as the existing GVS does), OutD Binning suggests that the pre-caching process can be efficiently performed by sequentially loading the edge lists from the Small-OutD bin. Moreover, it will be more beneficial to pre-cache the edge lists from the Small-OutD bin than that from the Large-OutD bin. This is because a larger number of vertices can be pre-cached in the same size of static cache, and each vertex is typically visited the same number of times by BFS-like algorithms.

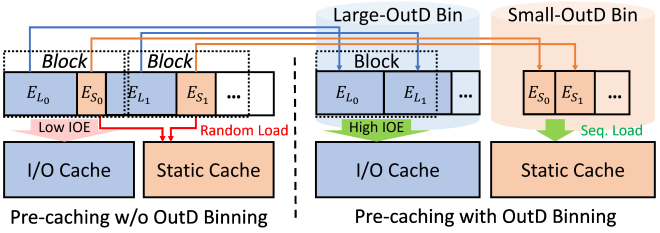


Figure 4: The Benefits of OutD-Binning.

3.3.2 Design Details

To make this design concept more practical, this section extends OutD Binning to split the BFDS-ordered graph into multiple bins. Furthermore, all bins are sorted and stored sequentially on the graph based on their average out-degrees in descending order. Thus, static cache can simply pre-cache the edge lists starting from the bin of the smallest average out-degree based on the available memory, making the pre-caching stage easy and efficient.

Nevertheless, bin size is important for overall performance. If the bin size is too small, the structure of BFDS-ordered graph could be destroyed, making the ordering less effective. If the bin size is much larger than the size of static cache, we could fail to pre-cache the critical edge lists, making static cache less effective. Thus, we propose to configure the bin sizes following the exponential decay of edge data size (as illustrated in Figure 5) since they are appropriate for large-scale graphs and perform well with or without static cache as shown in Section 4.2. Furthermore, we suggest to set the smallest bin to a size that is affordable by most of the computers (e.g., less than 2 GB) so that the whole smallest bin can be entirely pre-cached and the total number of bins could vary for graphs with different sizes.

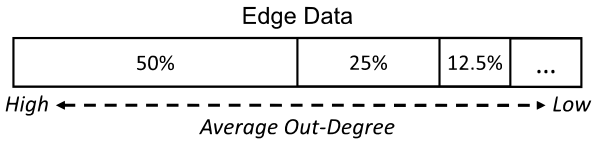


Figure 5: Layout of Exponential Decay Bin Sizes.

To order a graph with OutD Binning, we first sort all the vertices in ascending order based on their out-degrees so as to easily identify the small vertices. Next, starting from the smallest size of bin, we create bin by marking the vertices into the current bin. If the current bin is full, we will re-assign the vertex IDs based on the ordering of BFDS-optimized graph. This process keeps doing until all the bins are created.

Therefore, the time complexity of OutD Binning is $O(|V| \cdot log|V| + |V| + N \cdot |V|)$, where N is the total number of bins.

Besides the good performance, the advantage of utilizing the OutD Binning is two-fold. First, the graph optimized by OutD Binning provides high flexibility. That is, by providing multiple bins which demonstrate different I/O efficiency, OutD Binning embeds the results of vertex selection in graph; therefore, users can still easily pre-cache again without re-running OutD Binning because selection information preserves in the graph. On the contrary, GVS requires recomputation every time if the size of static cache changes or new set of graph workloads launches [28].

Second, OutD Binning enables efficient pre-caching. Particularly, OutD Binning stored the small edge lists together using bins. Thus, users can easily and efficiently pre-cache the edge lists based on the available memory in static cache. By contrast, GVS requires high complexity to select vertices in the pre-caching stage. The time complexity of GVS is $O(|E| + |V| + \sqrt{K \cdot M} + M \cdot log|V|)$, where the M is the static cache size and K is the parameter which is usually set to be hundreds or thousands [28]. Such time-consuming preprocessing might fail to reduce the end-to-end graph processing time especially when users do not have many workloads to amortize the huge overhead of GVS.

4 Evaluations

4.1 Evaluational Setup

This section conducts a series of evaluations to demonstrate the effectiveness of IOE-Order on Graphene. There are generally six classic BFS-like algorithms in the field [28], which are Breadth-First Search (BFS) [35], Single-Source Shortest Paths (SSSP) [19], All-Pair Shortest Path (APSP) [45], Weakly Connected Components (WCC) [19], Diameter Measurement (DIAM) [12], and Betweenness Centrality (BC) [7]. Nevertheless, some of them are similar to each others. For example, SSSP is implemented based on BFS, and DIAM consists of multiple rounds of BFS; besides, both of APSP and BC require the shortest paths from all vertices. Therefore, instead of evaluating all these six classical BFS-like algorithms, we only select BFS, APSP, and WCC, which demonstrate different behaviors of graph traversal. Details of the chosen algorithms are illustrated in the following:

Breadth-First Search (BFS) [35] is a typical algorithm for graph traversal. BFS begins with a user-input vertex and visits its neighbors by marking them to be the active vertices of next iteration. A visited vertex will not be visited again in the future. This procedure keeps going recursively until there is no more active vertex.

All-Pair Shortest Path (APSP) [45] computes the shortest paths from all the vertices in the graph. Due to the high complexity of computing SSSP from all vertices, an approximate approach is to randomly sample 32 source vertices, and calculate the distance by performing multi-source traversals from the sampled vertices.

Weakly Connected Components (WCC) [19] finds out the subgraphs that the vertices are all connected to each other. One way to implement WCC is to use BFS to detect the largest WCC first, and then explore the rest of smaller WCCs by exploiting label propagation.

We implement IOE-Order in C++ and evaluate its effectiveness using Graphene [29]. Notably, to the best of our knowledge, although there are several open-sourced, semi-external systems [25, 29, 53], Graphene [29] is the most state-of-the-art one that integrates multiple techniques for optimizing the graph processing on SSD; according to its paper [29], it performs the best against other existing semi-external systems (and even approaches the performance of in-memory systems such as Ligra [46] and Galois [37]). However, IOE-Order can be easily applied to other semi-external systems to further improve the performance of running BFS-like algorithms. Moreover, in contrast to the other systems [25, 53], it tackles the low I/O efficiency problem by leveraging 512-byte fine-grained I/O to read only the necessary data. Based on their designs, our method can further boost the processing performance of BFS-like algorithms by improving the I/O efficiency. Nevertheless, since Graphene does not support static cache, we realize the static cache as shown in Figure 1(a), where the static cache is implemented to be a separate memory space in addition to I/O cache.

To have a thorough comparison, we compare IOE-Order against the other three orderings incorporated with two vertex selection algorithms. In addition to Norder [28], we also add Rand-Order to be the baseline and In-degree Order (denoted as InD-Order), which re-assigns vertex IDs starting from the highest in-degree vertex. On the other hand, in addition to GVS [28], the other vertex selection algorithm is Out-Degree (denoted as OutD), which selects the vertices with small out-degree to keep as many edge lists in the static cache as possible. Because of the complexity to show many combinations of different designs, we use the notation Norder+GVS to represent the graph is ordered by Norder and the vertices are selected by GVS. By contrast, since the proposed method is a holistic optimization, we simply use IOE-Order to represent.

To show the accurate result, we run each graph algorithm five times and demonstrate the average result. However, BFS, in contrast to the other two algorithms, requires the user to input starting vertex. Thus, we randomly sample 32 starting vertices for BFS and report the average execution time. Table 2 lists all the graphs used for evaluation in this work. All of them are billion-scale, real-world graphs from webgraph [5,6]. Particularly, uk2007 [48] and gsh2015 [10] are web crawler graphs, while Twitter [11] is social graph. The largest graph in our experiment is gsh2015 [10], which contains 988 millions of vertices and 33.88 billions of edges.

All the experiments are conducted on HPE ProLiant DL560

Table 2: The Evaluated Graph Datasets.

Graph Name	Number of Vertices	Number of Edges
Twitter	42 M	1.4 B
uk2007	108 M	3.93 B
gsh2015	988 M	33.88 B

Gen10 server with Intel Xeon Platinum 8160 CPU and 1 TB DDR4-2666 memory on Debian GNU/Linux 9, and the storage device is Samsung SSD 860 EVO 1TB with SATA protocol. Please note that the actual memory used by Graphene is configured to be related to the size of edge data. Particularly, we fix the size of I/O cache to be 5% of the edge data, and vary the sizes of static cache to show the different results under various memory resources.

4.2 Evaluation of IOE-Order

We compare IOE-Order against all state-of-the-art optimizations. Figure 6 shows the overall comparison. Specifically, Figures 6(a), 6(c), and 6(b) depict the processing time of running BFS, APSP, and WCC, respectively, while Figures 6(d), 6(f), and 6(e) show the I/O efficiency of running the three algorithms in the same order. In each sub-figure, y-axis denotes the processing time or I/O efficiency and x-axis denotes the size of static cache, which ranges from 0% to 40% to the edge data size. Please note that 0% static cache means that there is no static cache but only I/O cache in Graphene. Since the loaded edge lists in static cache could be used for many times to amortize the pre-caching overhead, the results presented in Figure 6 skip the time spending on pre-caching stage, which will be discussed in detail in Section 4.3.

Overall speaking, ordering (0% static cache) has significant impact on performance. Not surprisingly, Rand-Order demonstrates the worst performance, while InD-Order averagely improves from Rand-Order by 43.6%. Norder, which is the state-of-the-art ordering optimization for BFS-like algorithms, performs better than InD-Order by 30.8%. IOE-Order further outperforms Norder by 16.5% and improve the I/O efficiency from 59.4% to 72.4% on average. Such improvement is due to BFDS ordering, which not only exploits high in-degree information but also traversal pattern to further effectively optimize the I/O efficiency.

Furthermore, we can generally observe the trend that ordering almost dominates the overall performance. That is, for most cases, given the same size of static cache, a better ordering will win no matter GVS or OutD is used. Nevertheless, GVS still outperforms OutD if the same ordering is adopted. Take Norder as an example, Norder+GVS averagely improves Norder+OutD by 10.2% in processing time. On the other hand, IOE-Order also demonstrates the effectiveness in pre-caching. Specifically, IOE-Order steadily improves Norder+GVS by 18.4%, 20.7%, 20.0%, and 18.2% regarding processing time on average as the static cache increases from 10% to 40%.

Since IOE-Order and Norder+GVS are generally the top two combinations which perform better than the others, we mainly focus on the comparison of these two in the following.

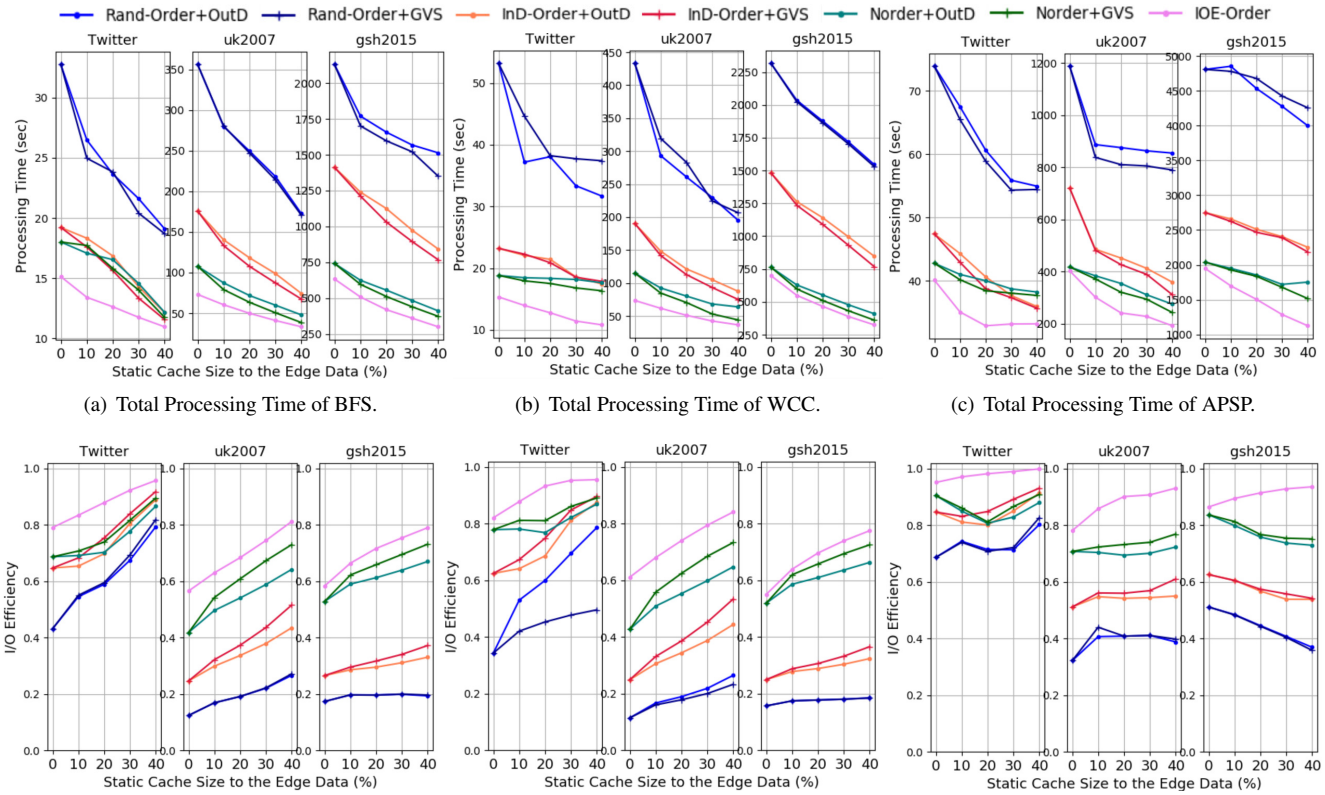
Details of BFS Evaluation. BFS, which is the foundation for all BFS-like algorithms, performs well with IOE-Order under all graphs. Averagely, IOE-Order improves 22.3%, 21.7%, 22.0%, 20.7%, and 17.3% against Norder+GVS when the size of static cache ranges from 0% to 40%. On the other hand, based on the various sizes of static cache, IOE-Order also provides the high I/O efficiencies, which averagely improves from 64.0% to 85.5%. We can see that the performance difference between IOE-Order and Norder+GVS becomes slightly smaller as the size of static cache increases. The reason is that, as more data are pre-cached, the performance gradually approaches optimum, making the optimization harder.

Details of WCC Evaluation. Generally speaking, in comparison with Norder+GVS, the processing time of IOE-Order improves averagely from 17.8% to 22.8% for various sizes of static cache. Since WCC can be decoupled into a BFS starting from a random vertex followed by label propagation, the results of WCC show a similar trend with BFS. Please refer to the discussion of BFS for more details.

Details of APSP Evaluation. Different from the other two algorithms, APSP randomly selects 32 source vertices for multi-source traversal, which requires more data from storage. Thus, APSP naturally shows higher I/O efficiency than the other two algorithms. Compared with Norder+GVS, IOE-Order averagely improves 4.7%, 14.4%, 19.0%, 20.5%, and 19.4% regarding processing time as static cache size ranges from 0% to 40%. Due to the nature of APSP, the performance difference between IOE-Order and Norder is small when there is no static cache. Nevertheless, as the size of static cache increases, GVS brings limited help in improving I/O efficiency because the I/O block issued by the graph system could contain the data which is already pre-cached in the static cache, which is likely to happen especially for the application like APSP requiring more data. Therefore, even if the processing time of Norder+GVS can still improve due to the more pre-cached data, the improvement of Norder+GVS is smaller than the improvement of IOE-Order. By contrast, since the pre-cached data are stored together in IOE-Ordered graph, the issued I/O blocks during graph processing will not contain those data, making I/O efficiency higher.

4.3 Overhead of Pre-processing

Table 3 shows the online pre-caching times of all graphs. First of all, GVS requires extremely long time to select vertices. For gsh2015 graph, it requires up to 83,131 seconds (which is around 112.19× of the total processing time of BFS) when the static cache size is 40% of the edge data. Such high time complexity might fails to reduce the end-to-end graph processing time especially when the number of graph workloads



(a) Total Processing Time of BFS. (b) Total Processing Time of WCC. (c) Total Processing Time of APSP.

(d) I/O Efficiency of BFS. (e) I/O Efficiency of WCC. (f) I/O Efficiency of APSP.

Figure 6: Overall Comparison among IOE-Order and The Other Methods.

Table 3: Online Pre-processing Time (seconds).

Graph	Twitter				uk2007				gsh2015			
Static Cache Size	10%	20%	30%	40%	10%	20%	30%	40%	10%	20%	30%	40%
Vertex Selection (OutD)	15.6	16.3	16.9	17.3	39.2	41.6	44.8	46.1	355.5	361.7	364.2	366.3
Edge Loading (OutD)	5.6	6.2	6.8	7.0	16.1	17.4	18.7	19.8	169.2	171.4	174.3	179.0
Vertex Selection (GVS)	1,181.5	1,750.3	2,096.9	2,350.3	2,914.7	4,708.3	6,061.0	7,053.4	31,781.2	53,491.6	70,388.6	83,131.4
Edge Loading (GVS)	5.5	5.9	6.4	6.6	16.3	17.6	18.7	19.6	173.2	180.3	185.6	191.4
Vertex Selection (IOE-Order)	*N/A (embedded in the step of OutD Binning)*											
Edge Loading (IOE-Order)	1.1	2.1	3.1	4.2	2.8	5.6	8.4	11.2	24.1	47.9	72.0	96.1

is not large enough to amortize the huge overhead of GVS. On the other hand, OutD demonstrates an efficient way for pre-caching, which only requires averagely 361.9 seconds for vertex selection and 173.5 seconds for edge loading for gsh2015. However, as shown in Section 4.2, OutD is less effective to benefit the graph processing than GVS. It might bring limited help when the number of graph workloads is large. Compared with the other methods, IOE-Order not only performs the best in graph processing but also enables very efficient pre-caching by eliminating the need of vertex selection. As a result, IOE-Order offers a more practical solution by bringing the greatest benefit regardless of the number of graph workloads.

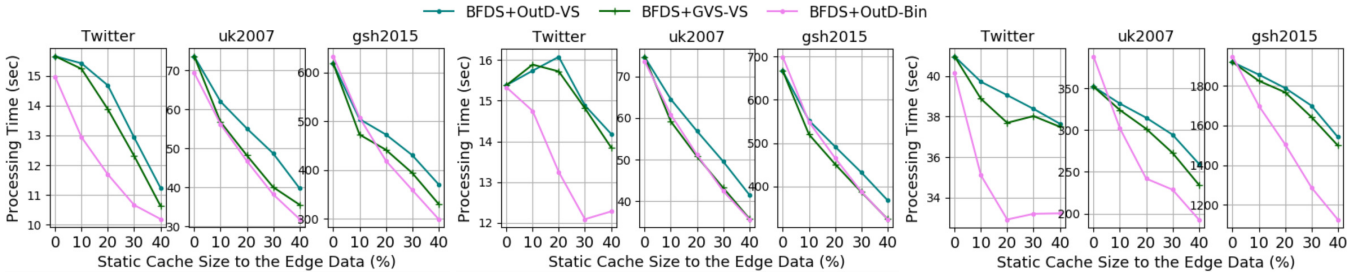
Table 4 shows the offline pre-processing times of all graphs. Since IOE-Order contains two steps to order a graph, the overhead of IOE-Order is around two to three times larger than the overhead of Norder. Fortunately, IOE-Order is inexpensive and brings more benefits than Norder as shown in Section 4.2. In particular, IOE-Order is not only 16.5% better on average but also embeds the result of vertex selection. Furthermore, since ordering is offline pre-processing, we regard IOE-Order as a more effective design than Norder.

Table 4: Offline Pre-processing Time (seconds).

Graph	Twitter	uk2007	gsh2015
Norder	24	52	483
IOE-Order (BFDS+OutD Binning)	57 (35+22)	130 (67+62)	1,580 (877+703)

4.4 Binning versus Vertex Selection

The following section will justify the benefit brought by the binning. Specifically, we show that the effectiveness of OutD Binning is better than OutD vertex selection. Although both



(a) Processing Time of BFS. (b) Processing Time of WCC. (c) Processing Time of APSP.

Figure 7: Comparison among OutD Binning, GVS-VS, and OutD-VS based on BFDS-ordered Graphs.

of designs are based on out-degree, binning can physically separate the pre-cached data, making the improvement better. Furthermore, compared with the time-consuming GVS, OutD Binning is not only much more efficient in pre-processing but even demonstrates obvious improvements for some specific cases. Please note that, since GVS needs to know how the graph is stored in storage to select vertices, it is extremely difficult to combine GVS with binning since binning will modify the order of graph all the time.

For better clarity, we denote OutD Binning as OutD-Bin, while GVS (OutD) vertex selection as GVS-VS (OutD-VS). Moreover, to have a fair comparison, we compare OutD-Bin against GVS-VS and OutD-VS based on the same ordering (BFDS ordering) with the same experiment setting as Section 4.2. Figure 7 depicts the processing times of all three algorithms, where x-axis denotes the static cache size and y-axis demonstrates the processing time. Because the space is limited and the trend of processing time and I/O efficiency is similar, we only show the results of processing time.

In general, OutD-Bin improves OutD-VS by -1.06%, 7.43%, 15.0%, 17.7%, and 17.0% as the size of static cache ranges from 0% to 40%. Since the binning will slightly damage the BFDS-ordered structure, BFDS with OutD-Bin performs marginally worse than BFDS when there is no static cache. Nevertheless, due to the effectiveness of physical separation, OutD-Bin performs gradually better than OutD-VS as the size of static cache increases. On the other hand, compared with GVS-VS, OutD-Bin improves 3.43%, 9.21%, 10.8%, and 10.0% on average as the size of static cache ranges from 10% to 40%. The major contribution of this improvement is due to APSP, where GVS-VS only brings limited help but OutD-Bin is still effective under the application with natural high I/O efficiency as discussed in Section 4.2. Particularly, the improvement can be up to 18.0% when static cache size is 40%. Furthermore, OutD-Bin performs better on Twitter, which also naturally shows higher I/O efficiency than the other graphs (as shown in Figure 6). Thus, we can also observe great improvement from GVS-VS to OutD-Bin under Twitter.

4.5 Impact of Bin Layout and Bin Sizes

As illustrated in Section 3.3, bin size for OutD Binning is crucial for overall performance. The following section shows the binning strategy of exponential decay (denoted as Exp. Decay) of the edge data size is generally better than the naïve binning strategy, which divides the graph into multiple equal-sized bins. Particularly, for naïve binning strategy, we create two graphs by setting the bin sizes to be 5% and 10% of edge data size; thus, there are totally 20 bins (denoted as 20-Bin) and 10 bins (denoted as 10-Bin) on the graphs. Due to page limitations, we only show the results of gsh2015, which is the largest graph evaluated in this paper.

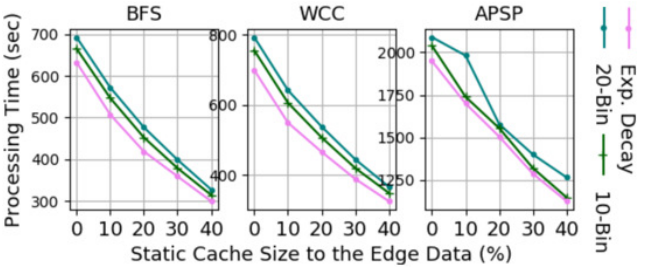


Figure 8: Gsh2015 with Different Binning Strategies.

Figure 8 depicts the processing times based on gsh2015, where x-axis denotes the static cache size to edge data and y-axis demonstrates the processing time. In general, since all binning strategies are able to capture the critical vertices, we can observe a similar improvement as the size of static cache increases. Thus, the performance under 0% static cache dominates the overall effectiveness. Particularly, Exp. Decay improves 10-Bin (20-Bin) by 5.7% (10.1%) on average. Furthermore, as shown in Section 4.4, Exp. Decay only slightly degrades by -1.06% compared with non-binning BFDS. As a result, binning strategy with exponential decay demonstrates a great way to not only preserve the structure of BFDS but also capture the critical vertices.

4.6 Evaluation on Non-BFS-like Algorithms

To discuss the impact of IOE-Order on non-BFS-like algorithms, this section further evaluates IOE-Order against other graph orderings with non-BFS-like algorithms. Due to page limitations, we only evaluate two popular non-BFS-like algorithms (i.e., PageRank and K-core), which represent two completely different access patterns, and only show the re-

sults of gsh2015 [10]. Specifically, PageRank [18, 39] ranks the importance of a webpage by repeatedly processing all vertices, whereas K-core [34, 43] detects the clustering structure of a graph by returning a maximal subgraph that consists of vertices of degree larger than K.

Table 5: Evaluation of Non-BFS-like Algorithms on Gsh2015.

Algo. \ Order	Random	In-Degree	Norder	IOE-Order
Pagerank	927.7	257.9	249.7	255.5
K-Core	795.7	534.2	371.1	314.1

Table 5 shows the total processing time (measured in seconds) based on gsh2015 when four different graph orderings are utilized with no static cache. Particularly, we can observe that IOE-Order also has the potential to improve (or at least not degrade) the performance of the two evaluated non-BFS-like algorithms. For PageRank, IOE-Order delivers similar processing time as in-degree ordering and Norder as all of them are designed to increase the locality of access for BFS-like algorithms instead of optimizing PageRank. As for K-core, IOE-Order outperforms the other three orderings (e.g., 15.4% better than Norder). This is because OutD-Binning of IOE-Order keeps the vertices of similar degrees together, resulting in good locality of access for K-core in selectively removing the vertices of less-than-K degrees.

5 Related Work

Fully External Graph Systems. Besides the semi-external systems presented in Section 2.1, many efforts have also been devoted to the development of fully external graph systems [1, 14, 17, 21, 27, 41, 49, 54]. In contrast to semi-external systems, fully external systems are very low-cost for storing *both* vertex and edge data in storage, and only require a small amount of memory for runtime graph processing. For example, GraphChi [27], which is the first fully external graph system, divides the whole graph into several partitions and loads them from storage into memory for further processing. Xstream [41] proposes to exploit edge-centric computation model to sequentially streams the edge data into memory. Based on edge-centric computation model, GridGraph [54] proposes 2D edge partitioning to further improve the performance by selectively accessing based on partition-level granularity.

Nevertheless, as these systems tend to issue large and sequential I/O to eliminate random access, they often suffer from the low utilization of loaded data when the graph algorithm only requires a small number of bytes. To tackle this issue, CLIP [1] proposes to increase the utilization of the loaded data by performing out-of-order execution to compute across future values. Particularly, CLIP re-computes the loaded partition to squeeze out all potential vertex updates [1]. LUMOS [49], on the other hand, further provides synchronous processing semantics for supporting synchronous graph algorithms based on the concept of future value computation. However, these works are proposed to increase the utilization of the loaded data for the systems leveraging sequential I/O. By contrast, IOE-Order is proposed to improve the I/O efficiency for semi-external systems that load only the necessary data on demand, and thus is orthogonal to the optimization of CLIP and LUMOS.

Graph Ordering. Ordering has been studied for a long time [3, 4, 9, 22] for in-memory graph systems [33, 37], which keep the whole graph in memory for processing, and is an important technique to improve the locality of access. For example, Pinar et al. leverage hypergraph to represent temporal or spatial localities so as to improve the performance of Sparse Matrixvector Multiplication [40]. Gorder optimizes the locality of vertex attribute updates to speed up CPU computation for general in-memory graph processing [50]. Nevertheless, they aims to optimize the computation order or vertex order, which are different from I/O optimization.

On the other hand, as discussed in Section 2.2, Norder [28] is proposed to optimize the processing performance of BFS-like graph algorithms on semi-external graph systems for reducing the I/O cost. IOE-Order has the same goal, but it further improves the I/O efficiency by exploiting the traversal pattern; moreover, OutD-Binning facilitates the pre-caching with lower pre-processing cost and higher flexibility.

6 Conclusion

This paper presents IOE-Order to practicably boost the processing performance of running BFS-like algorithms on semi-external graph systems by presenting a holistic graph ordering with two major pre-processing optimizations. Specifically, BFDS is proposed to leverage both graph traversal pattern and in-degree information to re-order the graph for higher I/O efficiency. Moreover, OutD-Binning is further introduced to refine and split the BFDS-ordered graph into multiple bins with different average out-degrees. Since all bins are further sorted and stored sequentially to form the edge data, OutD Binning not only enables high flexibility and efficiency in pre-caching the small edge lists but further increases the I/O efficiency in loading data from the rest of bins during graph processing. The evaluations show that, compared with the integrated solution of state-of-the-art pre-processing optimizations, IOE-Order delivers both high efficiency (by improving the processing time up to 36.1%) and high practicability (by entailing much lower pre-processing overhead) for various BFS-like algorithms.

Acknowledgments

We thank our shepherd, Nathan Beckmann, and all the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by The Research Grants Council of Hong Kong SAR (Project No. CUHK14208521).

References

[1] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 125–137, Santa Clara, CA, July 2017. USENIX Association.

[2] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Clip: A disk i/o focused parallel out-of-core graph processing system. *IEEE Transactions on Parallel and Distributed Systems*, 30(1):45–62, 2019.

[3] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31, 2016.

[4] Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar. Multithreaded clustering for multi-level hypergraph partitioning. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 848–859, 2012.

[5] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[6] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[7] Ulrik Brandes. A faster algorithm for betweenness centrality. The Journal of Mathematical Sociology, 25(2):163–177, 2001.

[8] Wei Chen and Shang-Hua Teng. Interplay between social influence and network centrality: A comparative study on shapley centrality and single-node-influence centrality. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 967–976, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[9] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, page 157–172, New York, NY, USA, 1969. Association for Computing Machinery.

[10] Gsh2015 dataset from WebGraph. http://law.di.unimi.it/webdata/gsh-2015/ , 2015.

[11] Twitter dataset from WebGraph. http://law.di.unimi.it/webdata/twitter-2010/ , 2010.

[12] Piotr Indyk Donald Aingworth, Chandra Chekuri and Rajeev Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). SIAM Journal on Computing, 28(4):1167–1181, 1999.

[13] Devdatt Dubhashi, Alessandro Mei, Alessandro Panconesi, Jaikumar Radhakrishnan, and Aravind Srinivasan. Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. *Journal of Computer and System Sciences*, 71, 03 2003.

[14] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-scale graph processing on emerging storage devices. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 309–316, USA, 2019. USENIX Association.

[15] Tiziano Fagni, Raffaele Perego, Fabrizio Silvestri, and Salvatore Orlando. Boosting the performance of web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Trans. Inf. Syst.*, 24(1):51–78, January 2006.

[16] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, October 2012. USENIX Association.

[17] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. New York, NY, USA, 2013. Association for Computing Machinery.

[18] Taher H. Haveliwala. Topic-sensitive pagerank. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, page 517–526, New York, NY, USA, 2002. Association for Computing Machinery.

[19] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. Communications of the ACM, 16(6):372–378, 1973.

[20] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, May 2001.

[21] 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.

[22] Konstantinos I. Karantasis, Andrew Lenharth, Donald Nguyen, Marás J. Garzarán, and Keshav Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 921–932, 2014.

[23] Jooho Kim and Makarand Hastak. Social network analysis: Characteristics of online social networks after a disaster. *International Journal of Information Management*, 38(1):86–96, 2018.

[24] Alec Kirkley, Hugo Barbosa, Marc Barthelemy, and Gourab Ghoshal. From the betweenness centrality in street networks to structural invariants in random planar graphs. *Nature Communications*, 9(1):2501, Jun 2018.

[25] Pradeep Kumar and H. Howie Huang. G-store: High-performance graph store for trillion-edge processing. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841, 2016.

[26] Chun-Yen Kuo, Ching Nam Hang, Pei-Duo Yu, and Chee Wei Tan. Parallel counting of triangles in large graphs: Pruning and hierarchical clustering algorithms. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6, 2018.

[27] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, October 2012. USENIX Association.

[28] Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam H. Noh, and Jiwon Seo. Pre-select static caching and neighborhood ordering for bfs-like algorithms on disk-based graph engines. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 459–474, Renton, WA, July 2019. USENIX Association.

[29] Hang Liu and H. Howie Huang. Graphene: Fine-grained IO management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, Santa Clara, CA, February 2017. USENIX Association.

[30] Zhen Liu, Philippe Nain, Nicolas Niclausse, and Don Towsley. Static caching of Web servers. In Kevin Jeffay, Dilip D. Kandlur, and Timothy Roscoe, editors, *Multimedia Computing and Networking 1998*, volume 3310, pages 179 – 190. International Society for Optics and Photonics, SPIE, 1997.

[31] Damien Magoni and Jean Jacques Pansiot. Analysis of the autonomous system network topology. *SIGCOMM Comput. Commun. Rev.*, 31(3):26–37, July 2001.

[32] Vladimir V. Makarov, Maxim O. Zhuravlev, Anastasija E. Runnova, Pavel Protasov, Vladimir A. Maksimenko, Nikita S. Frolov, Alexander N. Pisarchik, and Alexander E. Hramov. Betweenness centrality in multiplex brain network during mental task evaluation. *Phys. Rev. E*, 98:062413, Dec 2018.

[33] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[34] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *CoRR*, abs/1103.5320, 2011.

[35] Edward F. Moore. The shortest path through a maze. In Proceedings of the International Symposium on the Switching Theory, 1959, pages 285–292.

[36] Aida Mrzic, Pieter Meysman, Wout Bittremieux, Pieter Moris, Boris Cule, Bart Goethals, and Kris Laukens. Grasping frequent subgraph mining for bioinformatics applications. *BioData Mining*, 11(1):20, Sep 2018.

[37] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456–471, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Akinniyi Ojo, Ngok-Wa Ma, and Isaac Woungang. Modified floyd-warshall algorithm for equal cost multipath in software-defined data center. In *2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 346–351, 2015.

[39] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[40] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, page 30–es, New York, NY, USA, 1999. Association for Computing Machinery.

[41] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 472–488, New York, NY, USA, 2013. Association for Computing Machinery.

[42] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, December 2017.

[43] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. Streaming algorithms for k-core decomposition. *Proc. VLDB Endow.*, 6(6):433–444, apr 2013.

[44] Zhiyuan Shao, Jian He, Huiming Lv, and Hai Jin. Fog: A fast out-of-core graph processing framework. *International Journal of Parallel Programming*, 45(6):1259–1272, Dec 2017.

[45] Alfonso Shimbel. Structural parameters of communication networks. The bulletin of mathematical biophysics, 15(4):501–507, 1953.

[46] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, feb 2013.

[47] I. Tatarinov, A. Rousskov, and V. Soloviev. Static caching in web servers. In *Proceedings of Sixth International Conference on Computer Communications and Networks*, pages 410–417, 1997.

[48] uk-2007 dataset from WebGraph. http://law.di.unimi.it/webdata/uk-2007-01/ , 2007.

[49] Keval Vora. LUMOS: Dependency-driven disk-based graph processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 429–442, Renton, WA, July 2019. USENIX Association.

[50] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1813–1828, New York, NY, USA, 2016. Association for Computing Machinery.

[51] Junlong Zhang and Yu Luo. Degree centrality, betweenness centrality, and closeness centrality in social network. In *Proceedings of the 2017 2nd International Conference on Modelling, Simulation and Applied Mathematics (MSAM2017)*, pages 300–303. Atlantis Press, 2017/03.

[52] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. Cgraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 441–452, Boston, MA, July 2018. USENIX Association.

[53] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, February 2015. USENIX Association.

[54] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association.

DEPART: Replica Decoupling for Distributed Key-Value Storage

Qiang Zhang[1], Yongkun Li[1], Patrick P. C. Lee[2], Yinlong Xu[1,3], Si Wu[1]
[1]*University of Science and Technology of China* [2]*The Chinese University of Hong Kong*
[3]*Anhui Province Key Laboratory of High Performance Computing, USTC*

Abstract

Modern distributed key-value (KV) stores adopt replication for fault tolerance by distributing replicas of KV pairs across nodes. However, existing distributed KV stores often manage all replicas in the same index structure, thereby leading to significant I/O costs beyond the replication redundancy. We propose a notion called *replica decoupling*, which decouples the storage management of the primary and redundant copies of replicas, so as to not only mitigate the I/O costs in indexing, but also provide tunable performance. In particular, we design a novel two-layer log that enables tunable ordering for the redundant copies to achieve balanced read/write performance. We implement a distributed KV store prototype, DEPART, atop Cassandra. Experiments show that DEPART outperforms Cassandra in all performance aspects under various consistency levels and parameter settings. Specifically, under the eventual consistency setting, DEPART achieves up to 1.43×, 2.43×, 2.68×, and 1.44× throughput for writes, reads, scans, and updates, respectively.

1 Introduction

Key-value (KV) stores serve as essential building blocks in the storage infrastructure of modern data-intensive applications, such as web search [14, 31], social networking [57], photo stores [10], and cloud storage [25,37]. To support large-scale usage, KV stores are often deployed in a *distributed* manner by storing the data objects (in the form of KV pairs) across multiple nodes. Examples of distributed KV stores include BigTable [14], HBase [3], Dynamo [25], HyperDex [28], Cassandra [37], TiKV [50], and Riak [54].

Failures become prevalent in any large-scale deployment, so providing fault tolerance for distributed KV stores is critical. Replication remains the commonly used fault tolerance mechanism in modern distributed KV stores (including the examples listed above [3, 14, 25, 28, 37, 50, 54]). Specifically, for each KV pair issued by a user write, replication makes multiple exact copies (called *replicas*) and distributes the replicas across different nodes, so as to tolerate any node failure.

One subtlety is that each node internally stores all replicas in the same index structure; we call such an approach *uniform indexing*. For example, we have examined the codebases of various open-source distributed KV stores, including HBase [3], HyperDex [28], Cassandra [37], TiKV [50], and ScyllaDB [60], and they all adopt uniform indexing for replica management. In particular, they keep all replicas originated from different nodes in a *log-structured-merged tree (LSM-tree)* [48], a multi-level tree structure that supports efficient reads and writes of KV pairs and maintains sorted KV pairs in each level for efficient scans (or range queries) to consecutive ranges of KV pairs. They either build on local LSM-tree KV stores (e.g., HyperDex uses HyperLevelDB [27] and TiKV uses RocksDB [29]), or implement their own LSM-tree structures (e.g., in HBase and Cassandra).

Uniform indexing is simple to implement for replica management, but it also significantly degrades both the write and read performance. First, the LSM-tree performs frequent compaction operations that rewrite the currently stored KV pairs to maintain their sorted order in each level. Storing all replicas in the same LSM-tree exacerbates the write amplification beyond the replication redundancy. For example, when replication is disabled, the write amplifications of Cassandra and TiKV are 6.5× and 13.8×, respectively; however, under triple replication, the write amplifications reach 25.7× and 50.9× in Cassandra and TiKV, respectively, incurring more than three times in write amplification (§3.1). Also, as reading a KV pair needs to search multiple levels in the LSM-tree, uniform indexing amplifies the search space and exacerbates the read amplification as well. For example, under triple replication, the read amplification of Cassandra reaches 34.6× (§3.1).

Our insight is that instead of putting all replicas in the same index structure, if we use different index structures for managing the storage of different types of replicas, we not only mitigate the read/write amplifications by reducing the size of the index structure for each type of replicas, but also enable flexible storage management to adapt to different design trade-offs. We make a case by proposing *replica decoupling*, which decouples the storage management of the replicas of each KV pair based on the *primary copy* (i.e., the main replica of the KV pair) and the *redundant copies* (i.e., the remaining replicas of the KV pair aside the primary copy). We use the LSM-tree to manage the primary copies only, so as to preserve the design features of the LSM-tree but in a more lightweight manner; meanwhile, we use simpler but tunable index structures for the redundant copies to balance the read and write performance depending on the performance requirements.

In this paper, we design replica decoupling in DEPART, a novel distributed KV store that decouples the storage management of primary and redundant copies for fault tolerance. DEPART builds on Cassandra [37]. It supports lightweight

differentiation of the primary and redundant copies of replicas on the critical I/O path, while keeping the existing data organization and configurable consistency features of Cassandra. While managing the primary copies in the LSM-tree, DEPART proposes a novel *two-layer log* to manage the redundant copies with tunable ordering for balanced read and write performance. Its idea is to issue batched writes for the redundant copies into an append-only *global log* for high write performance. It further splits the global log into multiple *local logs*. In particular, the ordering of KV pairs in each local log is tunable by a *single* parameter to balance the read and write performance for the redundant copies; for example, given a high read (or write) consistency level (i.e., the number of replicas to be read (or written) in a successful operation; see §2.3), the two-layer log can be tuned to favor for high read (or write) performance. The two-layer log also improves failure recovery performance, by organizing the KV pairs by different key ranges and limiting a recovery operation to access only the relevant range of KV pairs. Our contributions are summarized as follows.

- We analyze two state-of-the-art distributed KV stores, Cassandra and TiKV, and reveal their performance limitations due to uniform indexing for replicas.
- We design DEPART, which realizes replica decoupling and has several key design features: (i) lightweight differentiation of primary and redundant copies, (ii) a two-layer log design with tunable ordering of redundant copies, and (iii) a fast failure recovery implementation via parallelization.
- We implement DEPART atop Cassandra v3.11.4 [2]. Experiments show that DEPART outperforms Cassandra in various settings. For example, for the case of eventual consistency, DEPART achieves 1.43×, 2.43×, 2.68×, and 1.44× throughput gains over Cassandra in writes, reads, scans, and updates, respectively. DEPART also maintains its read and write performance gains under various consistency level configurations.

The source code of our DEPART prototype is available at: `https://github.com/ustcadsl/depart`.

2 Background

We use Cassandra [37] (which serves as the baseline for our DEPART design) as an example to describe the background of a distributed KV store, including its storage architecture, I/O workflows, and consistency management.

2.1 Storage Architecture

Data organization. A distributed KV store partitions KV pairs across a cluster of nodes. In Cassandra, the KV pairs are partitioned based on *consistent hashing* [33], which has also been adopted by other production distributed KV stores [25, 41, 54, 60]. Consistent hashing associates the locations of all nodes with a *hash ring* and maps each KV pair deterministically to a node. Specifically, we consider a distributed

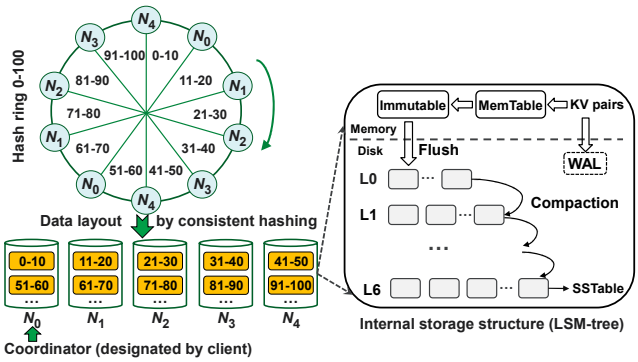


Figure 1: Storage architecture in Cassandra.

KV store with *n physical nodes*, each of which is associated with *v virtual nodes*. It divides the hash ring into $n \times v$ *ranges*, each of which covers one of the virtual nodes. For example, as shown in Figure 1, there are $n = 5$ physical nodes (i.e., N_0 to N_4) with $v = 2$ virtual nodes each. The hash ring contains $2 \times 5 = 10$ ranges, say $(0-10)$, $(11-20)$, $\cdots$, $(91-100)$. Each of the ranges is associated with the nearest virtual node in the clockwise direction in the hash ring and the corresponding physical node; for example, both ranges $(0-10)$ and $(51-60)$ are assigned to N_0. For each KV pair, consistent hashing hashes the key to a location in the hash ring (e.g., using MurmurHash [6] in Cassandra). The KV pair is then stored in the corresponding physical node that is associated with the range.

Replication is commonly used in modern distributed KV stores [3, 14, 25, 28, 37, 50] for fault tolerance, by distributing the replicas of each KV pair across different nodes to protect against node failures. In Cassandra, replicas are stored in a sequence of nodes along the clockwise direction in the hash ring denoted by $N_i, N_{(i+1) \bmod n}, N_{(i+2) \bmod n}, \cdots$, where $0 \le i \le n-1$ and N_i (i.e., the first node in the node sequence) is the node to which the KV pair is hashed based on consistent hashing. We refer to the replica that is stored in N_i as the *primary copy*, while referring to the remaining replicas that are stored in the successive physical nodes along the clockwise direction in the hash ring as the *redundant copies*.

Internal storage with the LSM-tree. Each node internally manages KV pairs with some index structure. In particular, the LSM-tree [48] is one of the most commonly used index structures in distributed KV stores, including Cassandra and others [3, 20, 28, 41, 50, 60]. An LSM-tree KV store organizes KV pairs in a multi-level tree and keeps KV pairs sorted by keys in each level, so as to support efficient reads, writes, and scans. As shown in Figure 1, the LSM-tree KV store maintains a tree-based index structure with multiple levels (denoted by $L_0, L_1, \cdots$) with an increasing capacity, in which each level stores the KV pairs in units of files called *SSTables*. It first appends the written KV pairs into an on-disk write-ahead log (WAL), and inserts them into an in-memory MemTable. When the MemTable is full, the LSM-tree KV store turns the MemTable to an immutable MemTable, which

is flushed to the lowest level L_0 as an SSTable. When a lower level reaches a capacity limit, the LSM-tree KV store merges the KV pairs at the lower level into the next higher level via *compaction*. To keep the KV pairs in each level sorted, a compaction operation first reads the KV pairs from both levels, merges the sorted KV pairs, and writes back the sorted KV pairs. Thus, compaction incurs extra I/Os during writes, leading to *write amplification*. Also, since KV pairs are not sorted across different levels, reading a KV pair needs to search from the lowest level L_0 to the higher levels, leading to *read amplification*. Both write and read amplifications are shown to cause significant performance degradations in LSM-tree KV stores [12, 43, 51].

2.2 I/O Workflows

Write workflow. Writing a KV pair in Cassandra works as follows. A client first randomly selects and connects to one of the nodes, called the *coordinator* and sends it the KV pair. The coordinator determines the nodes in which the primary and redundant copies are stored, based on consistent hashing. It then forwards the KV pairs to the nodes.

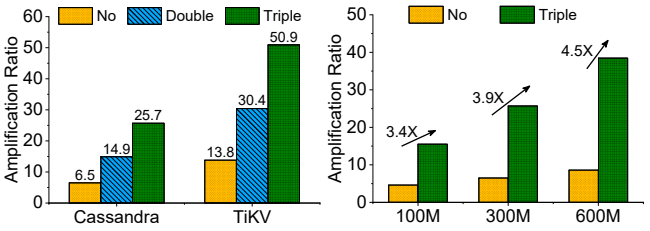
Read workflow. Reading a KV pair in Cassandra is similar to writing a KV pair and works as follows. The client first selects and contacts a coordinator. It issues the read request to the coordinator, which finds the nodes in which the replicas (regardless of primary and redundant copies) of the KV pair are stored. For load balancing, the coordinator prefers to read the KV pair from the nodes with low latencies, determined by the dynamic snitching module [5]. It then returns the KV pair to the client.

2.3 Consistency Management

Cassandra supports different consistency modes, e.g., strong consistency and eventual consistency. They are configured by tuning the *replication factor* (denoted by k), as well as the *read consistency level (RCL)* and the *write consistency level (WCL)*. The replication factor k is defined as the total number of replicas for fault tolerance. RCL and WCL are defined as the minimum numbers of replicas (regardless of primary or redundant copies) to be read and written by the coordinator to acknowledge the successful read and write operations, respectively. Both RCL and WCL are set as an integer from one to k. If WCL+RCL$> k$, then strong consistency is provided; if WCL+RCL $\leq k$, then eventual consistency is provided. By default, both WCL and RCL are set to one in Cassandra.

3 Replica Decoupling

To motivate replica decoupling, we describe the limitations of uniform indexing for managing all replicas in internal storage management (§3.1). We also describe the naïve replica decoupling designs to motivate our DEPART design (§3.2).



(a) Impact of replication in Cassandra and TiKV

(b) Impact of KV store size in Cassandra

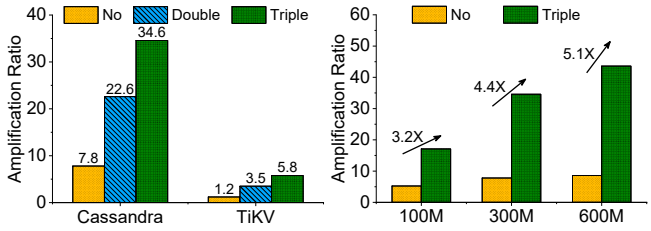
Figure 2: Write amplifications of no-replication ("No"), double replication ("double"), and triple replication ("triple") in Cassandra and TiKV.

3.1 Uniform Indexing and its Limitations

Recall from §1 that existing distributed KV stores (e.g., [3, 28, 37, 50]) mainly adopt uniform indexing, in which all replicas (including all primary and redundant copies) designated for each node are managed under the same index structure. We show that uniform indexing, rather than the extra writes from replication, is the main cause of significantly exacerbating both the write and read amplifications of the LSM-tree.

Limitation #1: Write amplification aggravation. With uniform indexing, each node treats all replicas as the regular KV pairs and stores them in the same LSM-tree without distinction (Figure 1). To show how it exacerbates the write amplification, we evaluate the write amplifications of two open-source distributed KV stores, Cassandra (v3.11.4) [2] and TiKV (release 4.0) [50]. Specifically, we deploy Cassandra and TiKV on a 5-node cluster with their default settings (detailed in §5). We configure a client machine to issue the writes of 300 M KV pairs of size 1 KiB each to the cluster that initially has empty storage. We consider no-replication ($k = 1$), double replication ($k = 2$), and triple replication ($k = 3$). Figure 2(a) shows that no-replication incurs a write amplification of 6.5× for Cassandra, due to the compaction overhead caused by the LSM-tree. However, for triple replication, the write amplification increases to 25.7×, which is around 4× the write amplification of no-replication. We also observe a similar trend for TiKV, where the write amplification increases from 13.8× to 50.9× (i.e., 3.7× increase).

Also, as the KV store size increases, the write amplification increases more significantly and shows a super-linear trend. The reason is that a larger KV store size increases the number of levels in the LSM-tree, leading to higher compaction overhead and a larger write amplification. We configure a client machine to issue the writes of 100 M, 300 M, and 600 M KV pairs of size 1 KiB each to the initially empty cluster. Here, we focus on Cassandra. Figure 2(b) shows the write amplifications of Cassandra for different KV store sizes. For a larger data store, the increase of the write amplification under triple replication compared to no replication also becomes larger. For example, triple replication has 3.4× write amplification compared to no replication under 100 M KV pairs, and becomes 4.5× under 600 M KV pairs. This super-linear trend



(a) Impact of replication in Cassandra and TiKV

(b) Impact of KV store size in Cassandra

Figure 3: Read amplifications of no-replication ("No"), double replication ("double"), and triple replication ("triple") in Cassandra and TiKV.

also implies that a larger KV store size can limit the scalability of uniform indexing.

Limitation #2: Read amplification aggravation. Uniform indexing also severely exacerbates the read amplification. The main reason is that all replicas are stored in the same LSM-tree, thereby enlarging the search space of KV pairs. We evaluate the read amplifications of Cassandra and TiKV as in the above settings, while a client machine issues 30 M reads to the existing 300 M KV pairs of size 1 KiB each that have already been stored. Figure 3(a) shows that for Cassandra, the read amplification increases from 7.8× in no-replication to 34.6× in triple replication (i.e., 4.4× increase). We observe a similar trend for TiKV.

In addition, we study the impact of the KV store size on the read amplification. Here, we focus on Cassandra. We first issue the writes of 100 M, 300 M, and 600 M KV pairs of size 1 KiB each to the initially empty cluster, followed by issuing 30 M reads to the existing KV pairs. Figure 3(b) shows a super-linear increase for the read amplification as the KV store size increases for Cassandra.

3.2 Motivation

Our analysis in §3.1 shows that uniform indexing exacerbates both write and read amplifications, as it is costly to manage all replicas within a single LSM-tree. This motivates us to explore the potentials of replica decoupling, which decouples the primary and redundant copies of replicas and manage them in separate index structures. We first consider two naïve replica decoupling approaches, and then motivate our design.

Naïve approaches. A simple replica decoupling approach is to deploy two LSM-trees, one for primary copies and one for all redundant copies. However, the LSM-tree for redundant copies still has a large size (especially for a large replication factor), while not all redundant copies are accessed in each I/O operation. For example, to recover a single-node failure under triple replication, only half of the redundant copies on average are accessed. Thus, there are extra I/Os for searching the whole LSM-tree for a subset of redundant copies.

Another simple replica decoupling approach is to manage k LSM-trees (k is the replication factor) for k replicas derived from each KV pair. For example, for Cassandra with triple replication, node N_i receives the redundant copies whose corresponding primary copies are stored in nodes $N_{(i-1) \bmod n}$ and $N_{(i-2) \bmod n}$ (where $0 \leq i \leq n-1$ and n is the number of physical nodes). Then we use three LSM-trees in node N_i, one of which stores the primary copies and the other two store the redundant copies from nodes $N_{(i-1) \bmod n}$ and $N_{(i-2) \bmod n}$, respectively.

However, maintaining multiple LSM-trees incurs both significant memory and I/O overheads. Since each LSM-tree has its own MemTable and immutable MemTable, the memory overhead amplifies by k times for the replication factor k. Specifically, if the MemTable size is m MiB and the cluster size is n, the memory cost of Cassandra is $m \times n$ MiB as each node maintains a single LSM-tree. However, when using k LSM-trees in each node, the memory cost becomes $k \times m \times n$ MiB, which is k times that in Cassandra. Note that if we reduce the MemTable size for each LSM-tree to limit the memory overhead, it degrades the efficiency of flushing the MemTable to disk, thereby degrading the user write performance [7, 8].

Also, each LSM-tree incurs its own compaction overhead for maintaining the fully-sorted ordering in each level. Thus, the compaction overhead is still significant and the compaction operations of multiple LSM-trees in the same node compete for the disk bandwidth, and hence the overall I/O performance is compromised. Our evaluation (Exp#1 in §5.2) shows that replica decoupling with multiple LSM-trees only brings limited performance gains over uniform indexing, even though the replicas are managed by different LSM-trees.

Our approach. Recall that the LSM-tree always maintains the fully-sorted ordering in each level. Using a single LSM-tree for all replicas in uniform indexing, or using multiple LSM-trees for replica decoupling, may favor high read performance, but both of them incur substantial high compaction overhead that degrades write performance. In particular, different consistency levels imply different performance requirements for the reads and writes issued to the replicas, such that a high read (or write) consistency level requires high read (or write) performance for the replicas. This motivates us to design a new storage management solution that supports *tunable ordering* for replica decoupling, so as to balance the read and write performance.

4 DEPART Design

We present DEPART, a distributed KV store that builds on Cassandra to realize replica decoupling by separating the storage management of primary and redundant copies. We introduce its architecture (§4.1) and elaborate its design techniques (§4.2-§4.5).

4.1 Overall Architecture

DEPART decouples the storage management of primary and redundant copies to achieve high performance. It manages the primary copies in the LSM-tree, while managing the re-

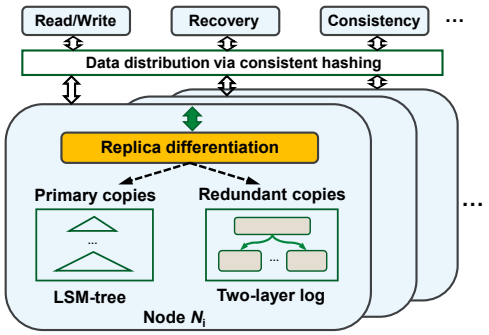


Figure 4: DEPART architecture.

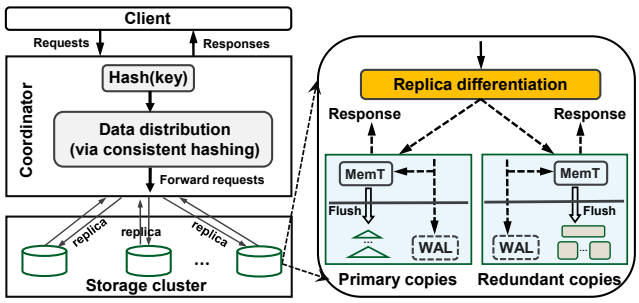


Figure 5: Replica differentiation in DEPART.

dundant copies in a novel *two-layer log*, whose ordering of redundant copies is tunable depending on the performance requirements. Keeping only the primary copies in the LSM-tree maintains the design features of the LSM-tree for reads, writes, and scans, but in a more lightweight manner as the LSM-tree size is now significantly smaller without the redundant copies. Also, the tunable ordering of the two-layer log allows balanced read and write performance for different settings of consistency levels.

Figure 4 depicts the architecture of DEPART. Note that DEPART only modifies the internal storage module of each Cassandra node, but preserves the inter-node management in Cassandra (e.g., consistent hashing for data organization and consistency management). In summary, DEPART addresses several design challenges via a number of techniques.

- **Lightweight replica differentiation.** DEPART differentiates the primary and redundant copies in the storage module of each node for separate management. Its replica differentiation is lightweight based on simple hash computations, and incurs limited overhead on the critical I/O path (§4.2).
- **Two-layer log design.** DEPART manages the redundant copies with a two-layer log, so as to achieve fast writes and efficient recovery. It first appends redundant copies to a *global log* as sequential batched writes. It then splits the global log into multiple *local logs* in background (§4.3).
- **Tunable ordering.** DEPART further provides a tunable ordering scheme for the two-layer log design to adjust the degree of ordering of the redundant copies with a single parameter, so as to balance the read and write performance for accessing the redundant copies (§4.4).
- **Parallel recovery.** DEPART uses a parallel recovery scheme that reads and writes the primary and redundant copies in parallel during recovery, so as to achieve high recovery performance (§4.5).

4.2 Replica Differentiation

DEPART differentiates the written KV pairs in the storage module of each node as primary or redundant copies. Figure 5 depicts the replica differentiation workflow. Recall that the coordinator forwards k replicas of a KV pair to a sequence of k nodes along the clockwise direction in the hash ring, where the key of the KV pair is hashed to the first node in the node sequence (§2). When a node, say N, receives one of the replicas of the KV pair from the coordinator, it performs the same hash computation (i.e., MurmurHash [6] in Cassandra) on the key of the replica and determines the node to which the key is hashed. If the resulting node is the same as N itself, then N is the first node in the node sequence and we refer to the replica as a primary copy; otherwise, we refer to the replica as a redundant copy.

Each node maintains a write-ahead log (WAL) and a MemTable for the LSM-tree (for primary copies) and the two-layer log (for redundant copies). After a node differentiates whether the KV pair is a primary copy or a redundant copy, it writes the KV pair to the corresponding WAL and MemTable and acknowledges the coordinator. When the MemTable is full and becomes immutable, the node flushes the immutable MemTable to either the LSM-tree or the two-layer log.

The logic of replica differentiation is lightweight, as it requires one extra hash computation in each storage node in the critical I/O path (and k extra computations in total for the replication factor k). Our experiments show that the differentiation time is less than 0.4% of the total write time (Exp#5 in §5.2).

4.3 Two-layer Log Design

Each node maintains a *two-layer log*, which is designed for the management of redundant copies with the following design features. First, it supports fast writes for the redundant copies, even though the number of redundant copies is much larger than that of primary copies and increases with the replication factor. Second, it supports tunable ordering to adapt to different consistency levels (§4.4). Third, it supports efficient parallel recovery of any failed nodes by allowing fast reads to the redundant copies in parallel.

Figure 6 shows the architecture of the two-layer log in each node. Upon receiving the replicas, a node first issues sequential batched writes for the redundant copies into a *global log*. A background thread continuously retrieves the redundant copies from the global log and splits them into multiple *local logs*. We elaborate the global log and local log designs below.

Append-only global log. To enable fast writes, each node writes all redundant copies of KV pairs (flushed from the

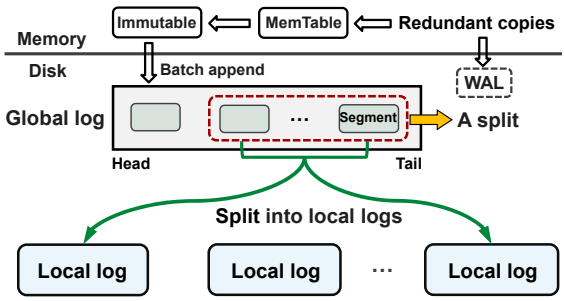


Figure 6: Architecture of the two-layer log design.

immutable MemTable) to an append-only global log. All redundant copies are grouped in units of *segments* and appended to the head of the global log as sequential batched writes. Note that the global log only stores all redundant copies without maintaining any extra index structure. Thus, it achieves high write performance for the redundant copies.

Keeping all redundant copies in the global log achieves high write performance, but poses two issues. First, the recovery performance degrades. For the redundant copies in the global log, their corresponding primary copies may reside in different nodes. When a node failure happens, only part of the redundant copies in the global log (i.e., the redundant copies whose corresponding primary copies reside in the failed node) are needed for recovery. Thus, recovery incurs only partial access to the global log, thereby incurring lots of random I/Os. Second, the garbage collection cost increases. As new KV pairs are appended to the log head, invalid (or stale) KV pairs cannot be overwritten and hence they occupy lots of space. This incurs large storage overhead, especially in update-intensive workloads. Garbage collection can be used to reduce the storage cost by continuously reclaiming the free space of invalid KV pairs from the log tail, but it inevitably introduces large amount of extra I/Os to read segments from the log tail and write back the valid KV pairs to the log head.

Splitting into local logs. To enable fast recovery, DEPART maintains a background thread to continuously split the global log into multiple local logs, each of which keeps only the redundant copies whose corresponding primary copies are stored in the same node. This allows the recovery of any failed node to access only the local log associated with the failed node. Note that each node only needs to maintain $k-1$ local logs (recall that k is the replication factor), since consistent hashing distributes the replicas in a sequence of nodes along the clockwise direction in the hash ring and each node only stores a redundant copy from up to $k-1$ nodes.

The splitting operation works as follows. It first retrieves a configurable number of segments, collectively called a *split*, from the tail of the global log. It then reorganizes a split of redundant copies into multiple sub-splits, each of which contains only the redundant copies whose corresponding primary copies reside in the same node. It finally writes back each sub-split into a separate local log in an append-only manner,

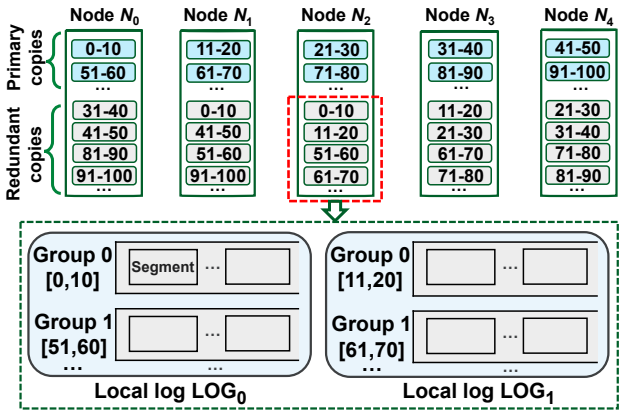


Figure 7: Range-based grouping within local logs.

and issues the writes to different local logs in parallel.

During splitting, DEPART also discards any invalid KV pairs in the selected segments. Thus, it does not trigger garbage collection explicitly; instead, it realizes garbage collection in the splitting operation to save the extra I/Os.

For each redundant copy, each node needs to determine the node in which its corresponding primary copy resides. It can be feasibly done locally within a node based on replica differentiation (§4.2).

Range-based grouping within local logs. While splitting the global log into multiple local logs alleviates the recovery and garbage collection overhead, the benefit remains limited since the ranges of a hash ring stored in each node are not necessarily contiguous (e.g., in Figure 1, Node N_0 stores ranges [0,10] and [51,60]). Recovering any range of KV pairs only needs to access the redundant copies for the range, so it still causes partial accesses to a local log and issues random I/Os.

We enhance each local log by managing KV pairs with range-based grouping. Figure 7 shows the idea of range-based grouping. Each local log is further divided into multiple *range groups*, each of which corresponds to a range in the hash ring. Note that different range groups within each local log have no overlaps in keys, so they can be managed independently. For example, for Node N_2 in Figure 7, the local log LOG_0 stores the redundant copies whose corresponding primary copies reside in Node N_0. As Node N_0 has two ranges, [0,10] and [51,60], LOG_0 now contains two range groups, each of which holds the redundant copies for [0,10] and [51,60], respectively. Range-based grouping can be realized by comparing the keys (or their hashes) with the boundary of each range in the hash ring based on consistent hashing (§2.1). It still ensures that the writes to each range group in a local log are performed in an append-only, batched manner. The number of range groups in a local log, and hence the number of ranges in Cassandra, are configurable by the parameter `num_tokens` [2]. Range-based grouping improves the recovery performance by accessing only the KV pairs in the corresponding range groups without accessing all KV pairs in the whole local log.

Under range-based grouping, when writing KV pairs from

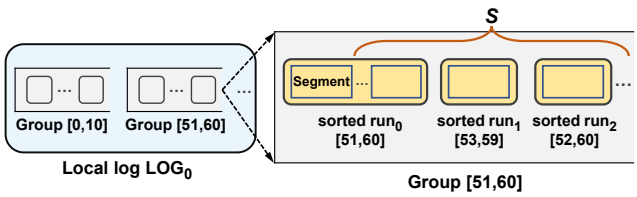


Figure 8: Tunable ordering.

the global log to the local logs during a splitting operation, DEPART further sorts all KV pairs by keys for each range before storing the KV pairs in a range group in the local logs; we call the sorted KV pairs for a range in a splitting operation a *sorted run*. Thus, each range group may store multiple sorted runs, while different sorted runs in the same range group may have overlaps in keys. Managing the range groups by sorted runs makes tunable ordering feasible (§4.4).

Reads from the two-layer log. To read a KV pair from the two-layer log, DEPART first checks the segments in the global log one by one, starting with the latest one. Note that the internal structure of each segment is similar to that of SSTables in the LSM-tree, so DEPART first reads the metadata from the segment and reads the corresponding KV pair according to the offset in the metadata. If the KV pair is not found in the global log, then DEPART searches the corresponding range group, located by comparing the key with the boundary keys of the range groups. Since each range group contains multiple sorted runs and KV pairs within the sorted run are fully sorted, DEPART searches from the latest to the oldest sorted run, and uses binary search to find the key within a sorted run.

4.4 Tunable Ordering

Recall that each range group in a local log may contain multiple sorted runs, and the KV pairs across the sorted runs within a range group are not fully sorted. If a range group contains too many sorted runs, the read performance for the redundant copies will degrade, especially for high read consistency levels where both primary and redundant copies are accessed in a read operation (§2.3). Thus, we extend the two-layer log with a *tunable ordering* scheme, in which users can configure a single parameter to adjust the degree of ordering of each range group for different consistency requirements.

DEPART adjusts the degree of ordering across multiple sorted runs with a user-configurable threshold S, which is a positive integer that controls the maximum number of sorted runs being allowed to exist in each range group. Figure 8 shows the idea of the tunable ordering scheme. For each new sorted run generated from a splitting operation, DEPART first checks if the existing number of sorted runs in a range group reaches the threshold. If not, it appends the new sorted run from the splitting operation directly to the range group; otherwise, it merge-sorts the new sorted run with the existing ones into a single sorted run. The merge-sort operation is similar to the compaction operation in the LSM-tree, including three

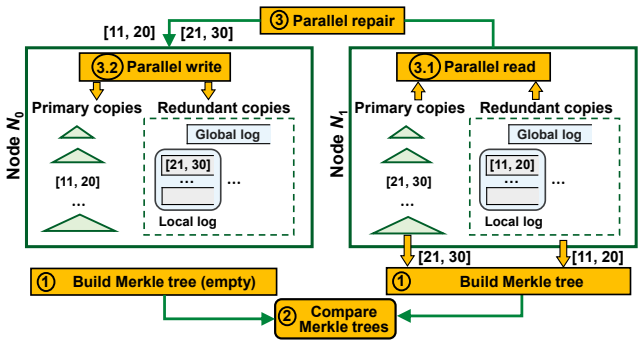


Figure 9: Parallel recovery in DEPART.

steps: (i) it reads all existing sorted runs from the range group; (ii) it merges all KV pairs in the new sorted run and the existing sorted runs, and if there exist multiple KV pairs with the same key in different sorted runs, it keeps only the KV pair in the latest sorted run and discards the older ones; and (iii) it writes back all merged KV pairs into the range group. We consider two special cases for different values of S.

- Case 1: $S = 1$. In this case, each range group always sorts the incoming sorted run with the currently stored sorted run, and hence all KV pairs are sorted. Thus, each local log resembles a single-level LSM-tree.
- Case 2: S approaches infinity. In this case, DEPART always appends the new sorted run into a range group without any merge-sorting with any existing sorted runs.

To set an appropriate value of S, we note that S determines the trade-off between the read and write performance. A small S favors the read performance by keeping a small number of sorted runs in a range group. It also maintains the storage efficiency by discarding the invalid KV pairs in merge-sort operations. However, it incurs a large merge-sort overhead that degrades the write performance. Thus, to support a system setting with the high read consistency level under read-dominant workloads, we should set a high degree of ordering with a small S to benefit reads; otherwise, we should decrease the degree of ordering by increasing the value of S to benefit writes. We also evaluate the impact of different values of S via experiments, and we recommend a default setting, $S = 20$, that can effectively balance the read and write performance under different consistency configurations (see Exp#8 in §5.2).

4.5 Parallel Recovery

For fast recovery of any failed node, DEPART proposes a *parallel recovery* scheme that exploits the benefit of decoupling the storage management of primary and redundant copies. We first review the recovery process in the current Cassandra implementation. Cassandra currently does not have a centralized node to monitor data loss and coordinate data recovery. Instead, it maintains a Merkle tree [4, 47] in each node to detect data inconsistency among multiple copies. Note that Merkle trees are also used by other consistent-hashing-based distributed KV stores, such as Dynamo [25] and Riak [54].

A Merkle tree is a binary hash tree, in which each leaf node stores the hash value of a range of KV pairs, while each non-leaf node stores the hash value of its child nodes. If a KV pair is lost, the replicas of the KV pair become inconsistent as detected by the Merkle trees across the nodes that store the replicas, so Cassandra triggers a recovery process. Specifically, the recovery process has three steps: (i) building a Merkle tree for each range of KV pairs in each node; (ii) comparing the Merkle trees of the same range of KV pairs in different nodes to identify any inconsistent range of KV pairs (which implies data loss); and (iii) reconstructing any inconsistent range by retrieving the range of KV pairs from a non-failed node and sending the range of KV pairs to the recovered node.

DEPART parallelizes the read and write processes for recovering multiple ranges of KV pairs for fast recovery. Its parallel recovery process is based on the recovery workflow in Cassandra, as shown in Figure 9. Suppose that we recover the lost data of a failed node at node N_0. First, each node in DEPART retrieves the KV pairs from the LSM-tree and the two-layer log, and builds its own Merkle tree (note that the Merkle tree in N_0 is initially empty) (Step 1). N_0 compares the Merkle trees and identifies the missing KV pairs (Step 2). To recover the lost KV pairs, each surviving node (e.g., node N_1 in Figure 9) issues parallel reads to the primary and redundant copies with two threads, and similarly the new node (i.e., N_0) retrieves the KV pairs from other surviving nodes and issues parallel writes for the primary and redundant copies with two threads. Such multi-threading is feasible as the primary and redundant copies are stored in different index structures.

5 Evaluation

DEPART builds on the codebase of Cassandra v3.11.4 [2] by implementing replica decoupling in the storage module of each node. Our DEPART prototype itself contains 6.9 K LoC, while the modification to Cassandra contains 1.9 K LoC. Note that Cassandra v3.11.4 contains about 206.2 K LoC. To demonstrate the benefits of the two-layer log design in DEPART, we also implement the naïve replica decoupling approach that simply stores replicas in multiple LSM-trees, which we refer to as *mLSM* (§3.2).

We conduct testbed experiments to demonstrate the efficiency of DEPART. We compare our DEPART prototype with Cassandra (v3.11.4), which performs uniform indexing for all replicas, mLSM. We address the following questions.

- How is the overall performance of DEPART compared with Cassandra and mLSM under different settings, e.g., the microbenchmark performance in different types of KV operations, the performance under different consistency configurations and different replication factors, as well as the performance under YCSB core workloads [21, 22]? (Experiments 1-4)
- What are the performance breakdowns of DEPART and Cassandra? (Experiment 5)
- What is the performance of DEPART when a node failure occurs? (Experiments 6-7)
- How does the performance of DEPART vary across parameter settings, including the ordering degree S, the store sizes, and the numbers of storage nodes? (Experiments 8-10)

5.1 Setup

Testbed. We conduct all experiments on a local cluster of multiple machines, each of which has two 12-core Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20 GHz, 32 GiB RAM, and a 500 GiB Samsung 860 EVO SATA SSD. All machines are interconnected via a 10 Gb/s Ethernet switch. Each machine runs CentOS 7.6.1810, with the 64-bit Linux kernel 3.10.0 and the Ext4 file system. We use one machine to simulate multiple clients via a thread pool, while the remaining machines serve as storage nodes.

Workloads. We generate workloads with YCSB [21, 22], a general-purpose cloud system benchmark tool. By default, we focus on 1 KiB KV pairs with 24-byte keys, and generate requests based on the Zipf distribution with the default Zipfian constant 0.99. We deploy YCSB on the client machine and set the number of client threads as 50, while each client thread issues a workload from YCSB.

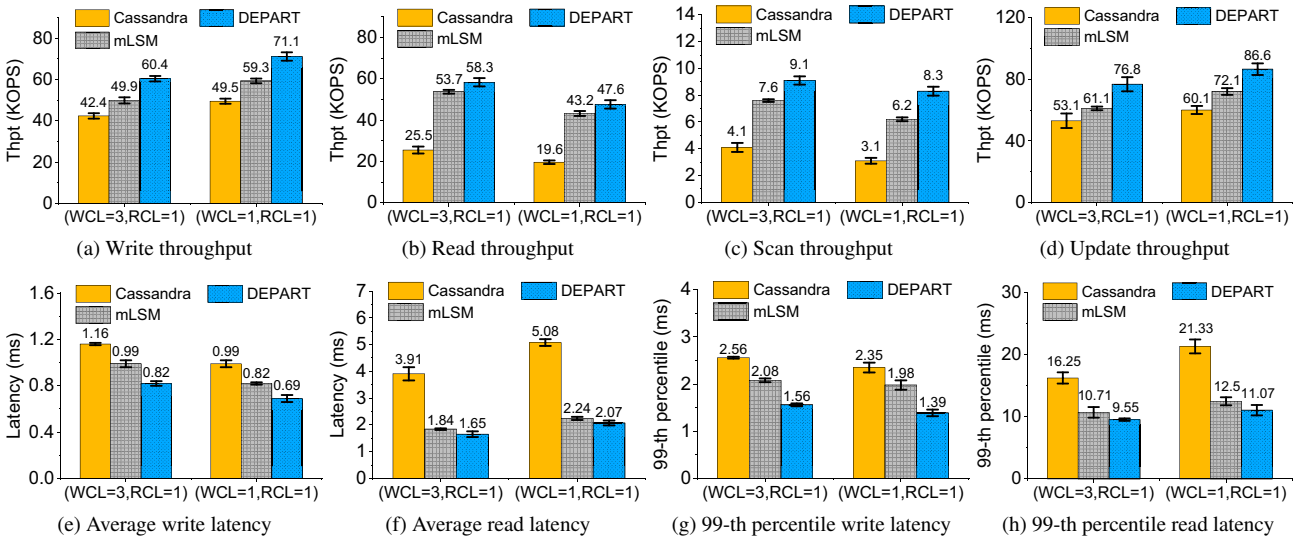
Default settings. We configure five storage nodes in the cluster and triple replication to deploy Cassandra and DEPART. Before each experiment, the cluster has empty storage. By default, we set (WCL=1, RCL=1) (i.e., the default setting in Cassandra), which corresponds to eventual consistency. We also study the impact of different consistency levels (Experiments 1 and 2). Both Cassandra and DEPART use the default `dynamic snitching` module [5] to choose the fastest nodes for serving reads, so as to load-balance reads across different replicas. For the parameter `num_tokens` [2], which determines the number of range groups, we use the default value 256 as in Cassandra.

For DEPART, we set the MemTable size to be the same as that of Cassandra (160 MiB by default), and the segment size in the global log to be the same as the MemTable size. Since DEPART keeps an extra MemTable for the two-layer log, we increase the `row_cache` size of Cassandra by 160 MiB for fair comparisons. For the two-layer log, we set the data size of each split operation as 20 segments (around 3 GiB) and set S as 20 to achieve balanced read and write performance. We keep the other parameter settings in Cassandra unchanged.

We plot the average results over five runs, with error bars showing the standard deviation.

5.2 Results

Experiment 1 (Performance in KV operations). We first compare the performance of Cassandra, mLSM, and DEPART in different KV operations, including writes (i.e., writing new KV pairs), reads (i.e., reading existing KV pairs), scans (i.e., reading existing consecutive KV pairs), and updates (i.e., updating existing KV pairs). We configure the client machine



(a) Write throughput (b) Read throughput (c) Scan throughput (d) Update throughput

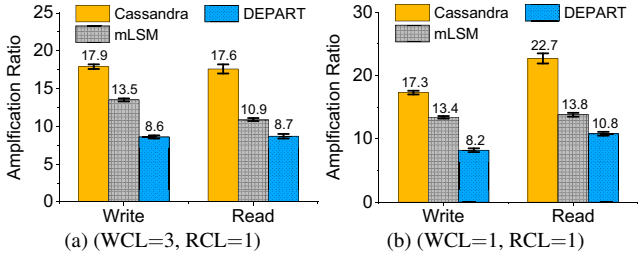
(e) Average write latency (f) Average read latency (g) 99-th percentile write latency (h) 99-th percentile read latency

Figure 10: Exp#1 (Performance in KV operations).

to first randomly write 200 M KV pairs. It then issues the following requests in order: (i) 20 M reads, (ii) 2 M scans (each scan contains one `seek()` to locate the first key and then iterates with 100 `next()`'s), and (iii) 200 M updates. We also consider two settings of consistency levels: (i) (WCL=3, RCL=1) (i.e., strong consistency) and (ii) (WCL=1, RCL=1) (i.e., eventual consistency).

Figure 10 shows the throughput and latency results. First, DEPART improves the overall performance over Cassandra in all cases. For (WCL=3, RCL=1), DEPART increases the throughput of writes, reads, scans, and updates to 1.42×, 2.29×, 2.22×, and 1.45×, respectively; it reduces the average write latency, average read latency, 99-th percentile write latency, and 99-th percentile read latency by 29%, 58%, 39%, and 41%, respectively. For (WCL=1, RCL=1), DEPART improves the throughput of writes, reads, scans, and updates to 1.43×, 2.43×, 2.68×, and 1.44×, respectively; it reduces the average write latency, average read latency, 99-th percentile write latency, and 99-th percentile read latency by 30%, 59%, 41%, and 48%, respectively. The latency results for scans and updates are similar and we omit the results here. The main reasons of the performance improvements of DEPART are two-fold. First, for reads, DEPART only searches the LSM-tree or the specific range group in the two-layer log within a node, thereby greatly reducing the search space. Second, for writes, DEPART mitigates the compaction overhead in the LSM-tree, which now keeps the primary copies only. The two-layer log also has limited merge-sort overhead by having a large value of the ordering degree S.

Second, mLSM notably improves the read performance, but only shows marginal improvements on writes. Specifically, compared with Cassandra, it increases the throughput of writes, reads, scans, and updates to 1.18-1.20×, 2.11-2.20×, 1.85-2.0×, and 1.15-1.20×, respectively. It also reduces the average write latency, average read latency, 99-th percentile

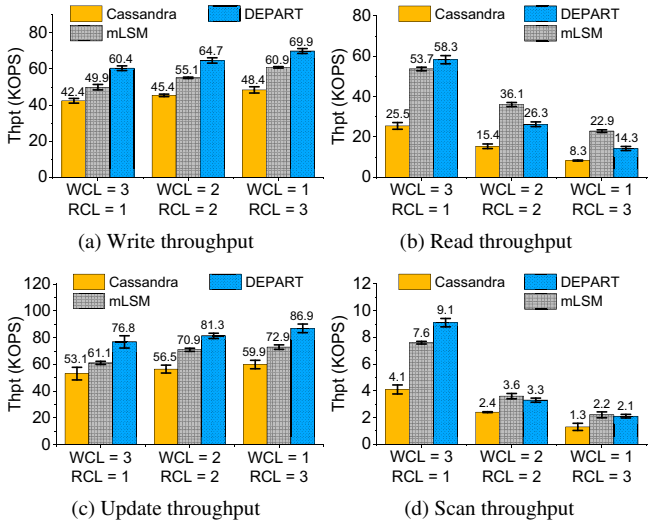


(a) (WCL=3, RCL=1) (b) (WCL=1, RCL=1)

Figure 11: Exp#1 (Read and write amplifications).

write latency, and 99-th percentile read latency by 15-17%, 53-56%, 16-18%, and 34-41%, respectively. The reason is that mLSM still triggers frequent compaction operations, which compete for disk bandwidth and degrade the write performance. For example, the total compaction sizes of Cassandra and mLSM are 3.46 TiB and 2.72 TiB, respectively, and the total compaction and merge-sort size of DEPART is 1.65 TiB (i.e., compared with Cassandra, mLSM only reduces the total compaction size by 21%, but DEPART reduces it by 52%).

We next compare the storage and memory costs of Cassandra, mLSM, and DEPART. After the end of the update phase, the KV store sizes are 613.5 GiB for Cassandra, 611.3 GiB for mLSM, and 654.8 GiB for DEPART. DEPART incurs 6.7% additional storage overhead compared with Cassandra, since each range group allows at most $S = 20$ sorted runs in our default setting and contains invalid KV pairs before being merge-sorted. To measure the memory overhead, we note that the MemTable size varies over time (up to the 160 MiB limit) as the KV pairs are continuously inserted into a MemTable and flushed to disk when the MemTable is full. Thus, we measure the total memory usage of the MemTables every five seconds and obtain the average results. The total memory usage of mLSM is 335.7 MiB, which is 3.7× that of Cassandra (90.4 MiB), as each LSM-tree maintains a MemTable. However, DEPART only costs 183.9 MiB, which is 2.0× that of Cassandra, since DEPART only maintains one extra



(a) Write throughput (b) Read throughput

(c) Update throughput (d) Scan throughput

Figure 12: Exp#2 (Performance under different consistency configurations).

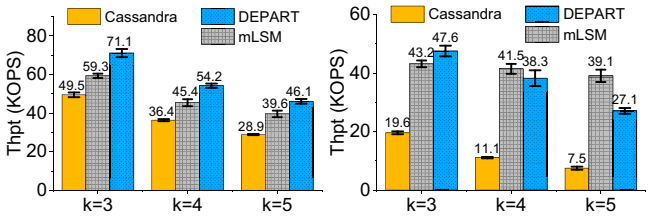
MemTable for the two-layer log. Note that the total memory usage of mLSM increases with the replication factor, while that of DEPART remains unaffected.

Finally, we compare the read/write amplifications (i.e., the ratios between the amounts of system reads/writes and the amounts of the user reads/writes) of Cassandra, mLSM, and DEPART. Figure 11 shows the results. Compared with Cassandra, mLSM reduces the read and write amplifications by up to 40% and 24%, respectively, while DEPART reduces the read and write amplifications by up to 53% and 52%, respectively. Note that the performance gain of DEPART is similar under both consistency levels, so we focus on (WCL=1, RCL=1) in the following experiments (except Exp#2 and 8).

Experiment 2 (Performance under different consistency configurations). We evaluate the performance under different consistency configurations. In particular, for strong consistency, we consider additional configurations for WCL and RCL under triple replication that satisfy the condition WCL+RCL>3, including (WCL=2, RCL=2) and (WCL=1, RCL=3).

Figure 12 shows the results. DEPART consistently improves the throughput of writes, reads, scans, and updates over Cassandra under different consistency configurations. Specifically, for (WCL=2, RCL=2), DEPART increases the throughput of writes, reads, scans, and updates to 1.43×, 1.70×, 1.38×, and 1.44×, respectively. For (WCL=1, RCL=3), DEPART increases the throughput of writes, reads, scans, and updates to 1.44×, 1.72×, 1.62×, 1.45×, respectively. DEPART also consistently improves the throughput of writes and updates over mLSM. Note that the write performance gains of DEPART over Cassandra stay nearly the same under different consistency configurations, since the index structures of both Cassandra and DEPART remain unchanged under triple replication.

However, the read performance gains of DEPART over



(a) Write throughput (b) Read throughput

Figure 13: Exp#3 (Performance under different replication factors).

Cassandra become smaller, and DEPART's read performance is worse than mLSM for RCL≥2. In this case, each read request needs to access at least two replicas successfully, so the redundant copies in the two-layer log must be searched. As the redundant copies in the two-layer log are not fully sorted, the performance is slower than reading the primary copies in the LSM-tree. Nevertheless, DEPART still achieves faster reads than Cassandra, as it searches for less data than Cassandra. Also, mLSM keeps redundant copies being fully sorted in each level, so it achieves higher read performance than DEPART. On the other hand, for RCL=1, each read only needs to access one replica for a successful operation. Most of the reads are routed to their primary copies, whose read latency is smaller than that of the redundant copies as determined by the dynamic snitching module (§2.2). Thus, the read performance gains under RCL=1 are higher than those under RCL≥2 in general.

Experiment 3 (Performance under different replication factors). We evaluate the performance of DEPART by varying the replication factor k from 3 to 5. We configure the client machine to first randomly write 200 M KV pairs and then issue 20 M reads.

Figure 13 shows the throughput results of writes and reads versus the replication factor. Compared with Cassandra, DEPART increases the throughput of writes and reads to 1.43-1.59× and 2.43-3.61×, respectively. Also, DEPART achieves a higher throughput gain for a larger replication factor. The main reasons are two-fold. First, for reads, DEPART either reads primary copies from the LSM-tree or reads redundant copies from the two-layer log; for the latter, it only searches the global log and the corresponding range group in the two-layer log. Thus, the read performance of DEPART is less affected by the number of replicas. However, Cassandra stores all replicas in a single LSM-tree and its reads need to traverse the whole LSM-tree. Its read performance drops significantly as the replication factor increases. Second, for writes, DEPART implements replica decoupling and manages the redundant copies in range groups. When the number of replicas increases, the compaction cost in the LSM-tree remains unchanged and the merge-sort cost in the two-layer log increases only slightly. However, Cassandra stores all replicas in the single LSM-tree and the compaction cost increases significantly as the number of replicas increases. Combining both reasons, the performance gain of DEPART becomes larger

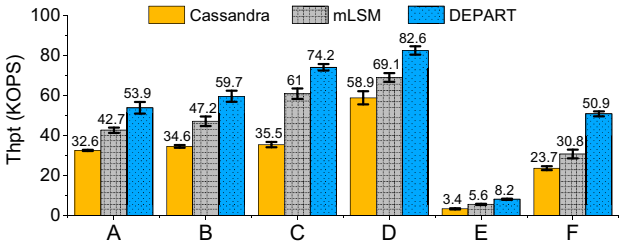
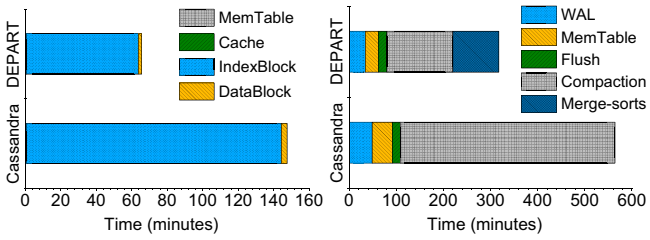
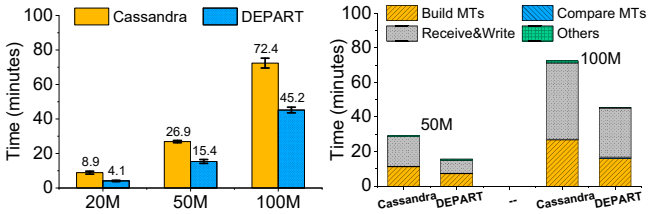


Figure 14: Exp#4 (YCSB performance).



(a) Read time breakdown (b) Write time breakdown

Figure 15: Exp#5 (Time breakdown for reads/writes).



(a) Recovery time (b) Time breakdown

Figure 16: Exp#6 (Recovery performance).

for a higher replication factor.

Similar to DEPART, mLSM also consistently improves the throughput of writes and reads over Cassandra under different replication factors. When the replication factor increases to $k = 4$ and $k = 5$, its read performance is even better than DEPART. The main reason is that mLSM only searches the corresponding LSM-tree that is fully sorted in each level regardless of the replication factor, but DEPART keeps redundant copies in the two-layer log that is not fully sorted under the default setting. Note that we can tune the degree of ordering of the two-layer log to further increase the read performance. Furthermore, the memory usage of DEPART remains 2× that of Cassandra, but that of mLSM increases to 5× when the replication factor increases to $k = 5$.

Experiment 4 (YCSB performance). We compare Cassandra, mLSM, and DEPART using the six YCSB core workloads [21, 22], namely A (50% reads, 50% writes), B (95% reads, 5% writes), C (100% reads), D (95% reads, 5% writes), E (95% scans, 5% writes), and F (50% reads, 50% read-modify-writes). The client machine first randomly writes 200 M KV pairs to the cluster before running each of the six YCSB core workloads. Each workload consists of 100 M operations, except for Workload E, which contains 10 M operations with each scan involving 100 `next()`'s.

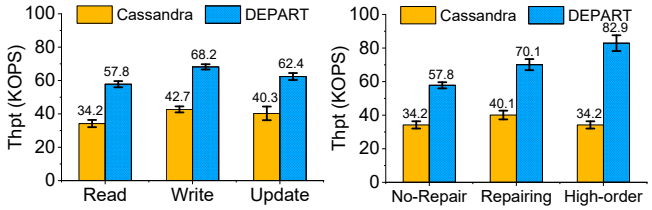
Figure 14 shows the results. DEPART outperforms Cassandra under all workloads. Specifically, it increases the throughput to 1.4-2.1× under read-dominant Workloads B-D, 1.6-2.2× under write-dominant Workloads A and F, and 2.4× under scan-dominant Workload E. With replica decoupling and the two-layer log design, DEPART reduces the compaction overhead of the LSM-tree during writes and reduces the search space during reads, so it improves both read and write performance simultaneously. On the other hand, mLSM also outperforms Cassandra under all workloads due to replica decoupling. However, DEPART further improves the performance of mLSM, as the latter incurs large compaction overhead.

Experiment 5 (Time breakdown for reads/writes). We show the time breakdown for both read and write processes in Cassandra and DEPART. We configure the client machine to first load 200 M KV pairs, followed by issuing 20 M reads. The read process comprises the reads to the MemTable, the cache (including the `row_cache` and `key_cache`), the index block of an SSTable (e.g., the Bloom filters and offsets), and the data block in the SSTable. Note that each segment in the two-layer log is treated as an SSTable here. The write process comprises writes to the WAL, writes to the MemTable, flushing the MemTable, the compaction of the LSM-tree, and the merge-sorts of the two-layer log (in DEPART only).

Figure 15 shows the time breakdown for reads and writes. For reads, most of the read time is for reading the index blocks of SSTables in both Cassandra and DEPART, since reading a KV pair needs to check the Bloom filter in the index block in each LSM-tree level to determine if the KV pair exists, and reads the data block from the SSTable according to the offset in the index block only if it does. Overall, DEPART reduces the time costs of reading the index blocks and the data blocks of SSTables by 56% and 45%, respectively. The reasons are two-fold. First, DEPART stores only the primary copies in the LSM-tree, so the number of SSTables in the LSM-tree greatly decreases. Also, DEPART manages the redundant copies in range groups, so the number of reads for locating a KV pair decreases as well.

For writes, DEPART reduces the time costs of writes to the WAL and writes to the MemTable by 28% and 37%, respectively, as DEPART writes primary and redundant copies in parallel. DEPART also greatly reduces the compaction overhead by reducing the LSM-tree size; for example, its compaction time is only 30.7% of Cassandra's. Furthermore, the merge-sort time of the two-layer log in DEPART is only 21.4% of the compaction time in Cassandra. Thus, the total time of compaction and merge-sorts in DEPART is only 52.1% of the compaction time in Cassandra.

Experiment 6 (Recovery performance). We evaluate the recovery performance on recovering a failed node. We consider different write sizes, by configuring the client machine to randomly write 20 M, 50 M, and 100 M KV pairs to the cluster. We then crash one node, by killing the KV store process with



(a) Throughput under failure (b) Degraded read

Figure 17: Exp#7 (Performance when a node crashes).

the "`kill -s processID`" command and removing all its data with the "`rm -r data`" command. Finally, we restart the KV store process on the same node and call the "`nodetool repair -full keyspacename`" command for recovery.

Figure 16(a) shows the total recovery time. Cassandra takes 8.9, 26.9, and 72.4 minutes to recover 20 M, 50 M, and 100 M KV pairs, respectively, while DEPART only takes 4.1, 15.4, and 45.2 minutes, respectively. Overall, DEPART reduces the recovery time of Cassandra by 38-54%. The main reason is that DEPART repairs primary and redundant copies in parallel, and scans much less data during recovery due to replica decoupling.

Figure 16(b) also shows the breakdown results of the recovery time for repairing 50 M and 100 M KV pairs. We consider different steps: Build MTs (i.e., building Merkle trees for all nodes), Compare MTs (i.e., comparing all Merkle trees), Receive&Write (i.e., receiving repaired data from other nodes and writing data to disk), and Others (i.e., other operations in recovery). DEPART reduces the time costs of Build MTs and Receive&Write by nearly a half compared to Cassandra through parallelizing the read/write processes to the primary and redundant copies.

Experiment 7 (Performance when a node crashes). We evaluate the read, write and update performance when a node crashes and before it is repaired. The client machine first randomly writes 100 M KV pairs to the cluster and we manually crash one node as in Experiment 6. We then issue 20 M reads, 100 M writes, and 100 M updates.

Figure 17(a) shows the throughput under a node failure. Compared with Cassandra, DEPART increases the throughput of reads, writes, and updates to 1.69×, 1.59×, and 1.55×, respectively. The main reason is that DEPART always searches much less data than Cassandra, even though it reads the redundant copies from the two-layer log. Also, DEPART always improves write performance under both normal and failure modes.

We also evaluate the degraded read performance in different cases: (i) the node repair is not yet triggered, (ii) the node repair is in progress, and (iii) the two-layer log for redundant copies in each node supports a higher degree of ordering by decreasing the threshold S (§4.4) from the default value 20 to 5. Figure 17(b) shows the degraded read throughput. When node repair is not triggered or is in progress, DEPART improves the throughput of degraded reads to 1.69× and 1.75×, respectively, since DEPART always searches much less data compared to uniform indexing in Cassandra. Also, when the two-layer log has a higher degree of ordering (e.g., with a smaller threshold S), the degraded read performance gains of DEPART become larger, because it is more efficient to read the KV pairs in the two-layer log that with a high degree of ordering.

S	Write thpt (KOPS)	Read thpt (KOPS)
1	37.2	42.3
10	57.2	31.5
20	64.7	23.1
$\rightarrow \infty$	78.4	7.6
Cassandra	45.4	15.4

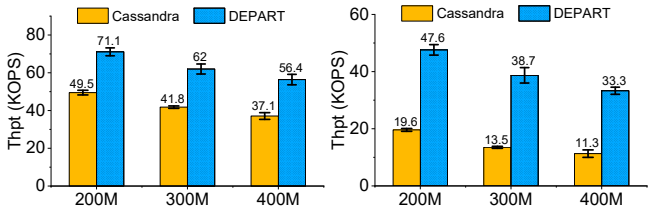
Table 1: Exp#8 (Impact of the ordering degree S).

Experiment 8 (Impact of the ordering degree S). We evaluate the write and read performance under different settings of the ordering degree S in DEPART, so as to show how DEPART can balance the read and write performance gains for the redundant copies by tuning the value of S. The client machine first randomly writes 200 M KV pairs to the cluster, followed by issuing 20 M reads. Here, we use the consistency configuration (WCL=2, RCL=2), so that the redundant copies must be accessed for each successful read.

Table 1 shows the results. For DEPART, if $S = 1$, the two-layer log reduces to a two-level LSM-tree, so it achieves the highest read throughput as the KV pairs are fully sorted, but the write throughput is the least due to the frequent merge-sorts for maintaining a single sorted run in each range group in the local logs. As we increase S (e.g., S is 10 or 20), the ordering of the two-layer log is relaxed and hence the merge-sort overhead becomes smaller, so the write throughput increases. As we set S to be a sufficiently large value, the two-layer log reduces to the append-only log, so the write throughput is the highest, but the read throughput is the least. Note that when S is 1, 10, or 20, DEPART still maintains higher throughput in both writes and reads than Cassandra, even though there exists a performance trade-off between writes and reads in DEPART.

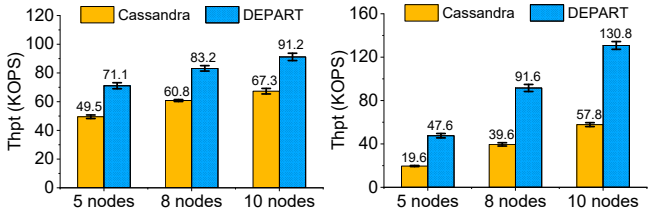
Experiment 9 (Impact of different KV store sizes). We now evaluate the impact of different KV store sizes. We vary the data size written by the client from 200 M to 400 M KV pairs (i.e., the total amount of primary and redundant copies increases from 600 GiB to 1200 GiB under triple replication). Figure 18 shows the throughput of writes and reads under different KV store sizes. Compared with Cassandra, DEPART improves the write and read throughput by 1.43-1.52× and 2.43-2.95×, respectively. Also, the performance gains of DEPART increase as the KV store size increases, as DEPART alleviates the write and read amplifications via replica decoupling, but Cassandra aggravates the write and read amplifications via uniform indexing.

To better show the scalability of the two-layer log design under different KV store sizes, we also evaluate the com-



(a) Write throughput (b) Read throughput

Figure 18: Exp#9 (Impact of different KV store sizes).



(a) Write throughput (b) Read throughput

Figure 19: Exp#10 (Impact of different numbers of nodes).

paction time of the LSM-tree and the merge-sort time of the two-layer log in DEPART, and compare them with the compaction time in Cassandra. When the KV store size increases from 200 M to 400 M KV pairs, the compaction time in Cassandra increases 2.3×, while the total time of compaction and merge-sort operations in DEPART only increases 1.9×. In particular, the ratio of the merge-sort time in DEPART to the compaction time in Cassandra drops from 21.4% to 18.7%, and the ratio of the compaction time in DEPART to the compaction time in Cassandra drops from 30.7% to 25.2%. Thus, DEPART scales well as the KV store grows.

Experiment 10 (Impact of different numbers of nodes). We evaluate the performance of DEPART when the cluster contains more nodes. We vary the number of nodes as 5, 8, and 10. We configure the client machine to issue the writes of 200 M, 320 M, and 400 M KV pairs, respectively, so that each node contains the same amount of data. After the writes, we configure the client machine to issue 20 M, 32 M, and 40 M reads, respectively.

Figure 19 shows the throughput results of write and read operations. Compared with Cassandra, DEPART increases the throughput of writes and reads to 1.35-1.43× and 2.26-2.43×, respectively. DEPART maintains its performance gains over Cassandra via replica decoupling, regardless of the cluster size. Thus, DEPART achieves good scalability as the cluster size increases.

6 Related Work

Local LSM-tree KV stores. A number of studies optimize the read and write performance of local LSM-tree KV stores that run on single machines. Read performance can be improved by Bloom filter optimization [23,38], adaptive caching [65], and scan optimization with succinct tries [68], while write performance can be improved by compaction optimization [24, 32, 55, 56], the fragmented LSM-tree [51], KV separation [12, 36, 43], I/O scheduling optimization [8], memory structure optimization [9], and a mix of optimization techniques for memory-disk-log components [7]. Our work focuses on the replica management in distributed KV stores, and is compatible with the above optimization techniques for local LSM-tree KV stores in individual nodes.

Distributed KV stores. Distributed KV stores can be classified into in-memory KV caches [42, 53] and persistent stores [1–3, 41, 50]. Optimization efforts for in-memory KV stores include lock-free and cache-friendly designs for high concurrency and throughput [13, 30, 40], erasure coding designs for memory efficiency [66, 67], self-tuning data placement [49], size-aware sharding for tail latency reduction [26], adaptive load balancing [16], secondary indexing [34], stretched Reed-Solomon coding [35], as well as hot spot optimization [15]. For distributed persistent KV stores, prior studies propose offline index construction for bulk loading [57], adaptive replica selection [52], multi-get scheduling [58], auto-tuning of tail latency optimization [39], load balancing [11], performance optimization via cost-benefit analysis with workload prediction [45], and optimizations of data placement and controlled migration [63, 64]. Persistent KV stores mostly adopt the LSM-tree in the storage layer to store all KV pairs. In contrast, DEPART proposes replica decoupling in distributed LSM-tree KV stores for efficient replica management.

Replica management. Prior studies improve the replication of distributed KV stores via efficient replica placement. Early studies include chain replication [61, 62] and its extension [44], low-cost wide-area replication [17], and dynamic hierarchical replication for data grids [46]. Copyset [19] and tiered replication [18] focus on maintaining high storage reliability. Replex [59] supports efficient queries on multiple keys. Our work focuses on the replica management within each storage node, while being compatible with the upper-layer replica placement policies.

7 Conclusion

We propose DEPART, which builds on a novel replica management scheme, replica decoupling, for distributed KV stores. DEPART uses a novel two-layer log design with tunable ordering to efficiently manage the redundant copies for different read and write performance requirements. Our DEPART prototype significantly outperforms Cassandra in different types of KV operations and maintains its performance gains for different consistency levels and parameter settings.

Acknowledgments

We thank our shepherd, Abutalib Aghayev, and the anonymous reviewers for their comments. This work is supported in part by NSFC (61772484, 61832011, 61772486), Youth Innovation Promotion Association CAS (No. 2019445), and USTC Research Funds of the Double First-Class Initiative (YD2150002003). Yongkun Li is USTC Tang Scholar, and he is the corresponding author.

References

[1] Amazon. DynamoDB. https://aws.amazon.com/dynamodb.

[2] Apache. Cassandra-3.11.4. https://github.com/apache/cassandra/tree/cassandra-3.11.4.

[3] Apache. HBase. https://hbase.apache.org/.

[4] Apache. Manual repair in Cassandra. https://docs.datastax.com/en/archived/cassandra/3.0/cassandra/operations/opsRepairNodesManualRepair.html.

[5] Apache. Dynamic snitching. https://www.datastax.com/blog/dynamic-snitching-cassandra-past-present_-and-future, 2021.

[6] A. Appleby. Murmurhash. https://sites.google.com/site/murmurhash/.

[7] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proc. of USENIX ATC*, 2017.

[8] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *Proc. of USENIX ATC*, 2019.

[9] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. FloDB: Unlocking memory in persistent key-value stores. In *Proc. of ACM EuroSys*, 2017.

[10] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *Proc. of USENIX OSDI*, 2010.

[11] K. L. Bogdanov, W. Reda, G. Q. Maguire, D. Kostić, and M. Canini. Fast and accurate load balancing for geo-distributed storage systems. In *Proc. of ACM SoCC*, 2018.

[12] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu. HashKV: Enabling efficient updates in KV storage via hashing. In *Proc. of USENIX ATC*, 2018.

[13] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proc. of ACM SIGMOD*, 2018.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of USENIX OSDI*, 2006.

[15] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li. HotRing: A hotspot-aware in-memory key-value store. In *Proc. of USENIX FAST*, 2020.

[16] Y. Cheng, A. Gupta, and A. R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proc. of ACM EuroSys*, 2015.

[17] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. of USENIX NSDI*, 2006.

[18] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *Proc. of USENIX ATC*, 2015.

[19] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Proc. of USENIX ATC*, 2013.

[20] F. Community. Fauna. https://docs.fauna.com/fauna/current/.

[21] B. F. Cooper. YCSB-0.15.0. https://github.com/brianfrankcooper/YCSB.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SoCC*, 2010.

[23] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *Proc. of ACM SIGMOD*, 2017.

[24] N. Dayan and S. Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proc. of ACM SIGMOD*, 2018.

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of of ACM SOSP*, 2007.

[26] D. Didona and W. Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *Proc. of USENIX NSDI*, 2019.

[27] R. Escriva. HyperLevelDB. https://github.com/rescrv/HyperLevelDB/.

[28] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store. In *Proc. of ACM SIGCOMM*, 2012.

[29] Facebook. RocksDB. https://rocksdb.org.

[30] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. of USENIX NSDI*, 2013.

[31] S. Ghemawat and J. Dean. LevelDB. https://leveldb.org.

[32] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. In *Proc. of ACM EuroSys*, 2015.

[33] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM STOC*, 1997.

[34] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. SLIK: Scalable low-latency indexes for a key-value store. In *Proc. of USENIX ATC*, 2016.

[35] T. H. Konstantin Taranov, Gustavo Alonso. Fast and strongly-consistent per-item resilience in key-value stores. In *Proc. of ACM EuroSys*, 2018.

[36] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's key-value storage system for cloud data. In *Proc. of IEEE MSST*, 2015.

[37] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[38] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu. ElasticBF: Elastic bloom filter with hotness awareness for boosting read performance in large key-value stores. In *Proc. of USENIX ATC*, 2019.

[39] Z. L. Li, C.-J. M. Liang, W. He, L. Zhu, W. Dai, J. Jiang, and G. Sun. Metis: Robustly tuning tail latencies of cloud systems. In *Proc. of USENIX ATC*, 2018.

[40] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. of USENIX NSDI*, 2014.

[41] LinkedIn. Voldemort. `http://project-voldemort.com/`.

[42] LiveJournal. Memcached. `https://memcached.org/`.

[43] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *Proc. of USENIX FAST*, 2016.

[44] J. MacCormick, N. Murphy, V. Ramasubramanian, U. Wieder, J. Yang, and L. Zhou. Kinesis: A new approach to replica placement in distributed storage systems. *ACM Trans. Storage*, 4(4):11:1–11:28, Feb. 2009.

[45] A. Mahgoub, P. Wood, A. Medoff, S. Mitra, F. Meyer, S. Chaterji, and S. Bagchi. SOPHIA: Online reconfiguration of clustered NoSQL databases for time-varying workloads. In *Proc. of USENIX ATC*, 2019.

[46] N. Mansouri and G. H. Dastghaibyfard. A dynamic replica management strategy in data grid. *Journal of Network and Computer Applications*, 35(4):1297–1303, 2012.

[47] R. C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques*, 1987.

[48] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree. *Acta Informatica*, 33(4):351–385, 1996.

[49] J. a. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. AutoPlacer: Scalable self-tuning data placement in distributed key-value stores. *ACM Trans. Auton. Adapt. Syst.*, 9(4):19.1–19.30, Dec. 2015.

[50] PingCAP. TiKV. `https://tikv.org`.

[51] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proc. of ACM SOSP*, 2017.

[52] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proc. of ACM EuroSys*, 2017.

[53] Redislab. Redis. `https://redis.io`, 2017.

[54] Riak Community. Riak. `https://riak.com`.

[55] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *Proc. of ACM SIGMOD*, 2012.

[56] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building workload-independent storage with VT-Trees. In *Proc. of USENIX FAST*, 2013.

[57] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project Voldemort. In *Proc. of USENIX FAST*, 2012.

[58] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proc. of USENIX NSDI*, 2015.

[59] A. Tai, M. Wei, M. J. Freedman, I. Abraham, and D. Malkhi. Replex: A scalable, highly available multi-index data store. In *Proc. of USENIX ATC*, 2016.

[60] S. Team. Scylladb. `https://www.scylladb.com`.

[61] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proc. of USENIX ATC*, 2009.

[62] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. of USENIX OSDI*, 2004.

[63] R. Vilaça, R. Oliveira, and J. Pereira. *A Correlation-Aware Data Placement Strategy for Key-Value Stores*, pages 214–227. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[64] L. Wang, Y. Zhang, J. Xu, and G. Xue. MAPX: Controlled data migration in the expansion of decentralized object-based storage systems. In *Proc. of USENIX FAST*, 2020.

[65] F. Wu, M.-H. Yang, B. Zhang, and D. H. Du. AC-Key: Adaptive caching for LSM-based key-value stores. In *Proc. of USENIX ATC*, 2020.

[66] M. M. Yiu, H. H. Chan, and P. P. Lee. Erasure coding for small objects in in-memory KV storage. In *Proc. of ACM SYSTOR*, page 14. ACM, 2017.

[67] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proc. of USENIX FAST*, 2016.

[68] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical range query filtering with fast succinct tries. In *Proc. of ACM SIGMOD*, 2018.

PAIO: General, Portable I/O Optimizations With Minor Application Modifications

Ricardo Macedo, Yusuke Tanimura[†], Jason Haga[†], Vijay Chidambaram[‡], José Pereira, João Paulo
INESC TEC and University of Minho [†]*AIST* [‡]*UT Austin and VMware Research*

Abstract

We present PAIO, a framework that allows developers to implement portable I/O policies and optimizations for different applications with minor modifications to their original code base. The chief insight behind PAIO is that if we are able to intercept and differentiate requests as they flow through different layers of the I/O stack, we can enforce complex storage policies without significantly changing the layers themselves. PAIO adopts ideas from the Software-Defined Storage community, building data plane stages that mediate and optimize I/O requests across layers and a control plane that coordinates and fine-tunes stages according to different storage policies. We demonstrate the performance and applicability of PAIO with two use cases. The first improves 99^{th} percentile latency by 4× in industry-standard LSM-based key-value stores. The second ensures dynamic per-application bandwidth guarantees under shared storage environments.

1 Introduction

Data-centric systems such as databases, key-value stores (KVS), and machine learning engines have become an integral part of modern I/O stacks [12, 19, 32, 43, 53, 55]. Good performance for these systems often requires storage optimizations such as I/O scheduling, differentiation, and caching. However, these optimizations are implemented in a sub-optimal manner, as these are *tightly coupled to the system implementation*, and can *interfere with each other due to lack of global context*. For example, optimizations such as differentiating foreground and background I/O to reduce tail latency are broadly applicable; however, the way they are implemented in KVS today (*e.g.,* SILK [16]) requires a deep understanding of the system, and are not portable across other KVS. Similarly, optimizations from applications deployed at shared infrastructures may conflict due to not being aware of each other [27, 51, 61, 62].

In this paper, we argue that there is a better way to implement such storage optimizations. We present PAIO, a user-level framework that enables building portable and generally applicable storage optimizations by adopting ideas from the Software-Defined Storage (SDS) community [38]. The key idea is to implement the optimizations *outside* the applications, as *data plane stages*, by intercepting and handling the I/O performed by these. These optimizations are then controlled by a logically centralized manager, the *control plane*, that has the global context necessary to prevent interference among them. PAIO does not require any modifications to the kernel (critical for deployment). Using PAIO, one can decouple complex storage optimizations from current systems, such as I/O differentiation and scheduling, while achieving results similar to or better than tightly coupled optimizations.

Building PAIO is not trivial, as it requires addressing multiple challenges that are not supported by current solutions. To perform complex I/O optimizations outside the application, PAIO needs to *propagate context* down the I/O stack, from high-level APIs down to the lower layers that perform I/O in smaller granularities.[1] It achieves this by combining ideas from *context propagation* [36], enabling application-level information to be propagated to data plane stages with minor code changes and without modifying existing APIs.

PAIO requires the design of new abstractions that allow differentiating and mediating I/O requests between user-space I/O layers. These abstractions must promote the implementation and portability of a variety of storage optimizations. PAIO achieves this with four main abstractions. The *enforcement object* is a programmable component that applies a single user-defined policy, such as rate limiting or scheduling, to incoming I/O requests. PAIO characterizes and differentiates requests using *context objects*, and connects I/O requests, enforcement objects and context objects through *channels*. To ensure coordination (*e.g.,* fairness, prioritization) across independent storage optimizations, the control plane, with global visibility, fine-tunes the enforcement objects by using *rules*.

With these new features and abstractions, system designers can use PAIO to develop custom-made SDS data plane stages. To demonstrate this, we validate PAIO under two use cases. First, we implement a stage in RocksDB [9] and demonstrate how to prevent latency spikes by orchestrating foreground and background tasks. Results show that a PAIO-enabled RocksDB improves 99^{th} percentile latency by 4× under different workloads and testing scenarios (*e.g.,* different storage devices, with and without I/O bandwidth restrictions) when compared to baseline RocksDB, and achieves similar tail latency performance when compared to SILK [16]. Our approach demonstrates that complex I/O optimizations, such as SILK's I/O scheduler, can be decoupled from the original layer to a self-contained, easier to maintain, and portable stage. Second, we apply PAIO to TensorFlow [11] and show how to achieve dynamic per-application bandwidth guarantees under a real shared-storage scenario at the ABCI supercomputer [1]. Results show that all PAIO-enabled TensorFlow instances are

[1] We refer to the term *"layer"* as a component of a given I/O stack that handles I/O requests (*e.g.,* application, KVS, file system, device driver).

provisioned with their bandwidth goals. This shows that PAIO enables enforcing storage policies with system-wide visibility and holistic control.

In summary, the paper makes the following contributions:

- PAIO, a user-level framework for building programmable and dynamically adaptable data plane stages (§3-§7). PAIO is publicly available at https://github.com/dsrhaslab/paio.
- Implementation of two stages to (1) reduce latency spikes in an LSM KVS; and (2) achieve per-application bandwidth guarantees under shared storage settings (§8).
- Experimental results demonstrating PAIO's performance and applicability under synthetic and real scenarios (§9).

2 Motivation and Challenges

We now describe the problems of system-specific I/O optimizations and how these drive the proposal of PAIO.

Problem 1: tightly coupled optimizations. Most I/O optimizations are single-purposed as they are tightly integrated within the core of each system [16, 29, 50]. Implementing these optimizations requires deep understanding of the system's internal operation model and profound code refactoring, limiting their maintainability and portability across systems that would equally benefit from them. For instance, to reduce tail latency spikes at RocksDB, an industry-standard LSM-based KVS, SILK proposes an I/O scheduler to control the interference between foreground and background tasks. However, applying this optimization over RocksDB required changing several core modules made of thousands of LoC, including *background operation handlers*, *internal queuing logic*, and *thread pools* [5, 15]. Further, porting this optimization to other KVS (*e.g.,* LevelDB [21], PebblesDB [47]) is not trivial, as even though they share the same high-level design, the internal I/O logic differs across implementations (*e.g.,* data structures [20, 47], compaction algorithms [34, 47]).

Solution: decouple optimizations. I/O optimizations should be disaggregated from the system's internal logic and moved to a dedicated layer, becoming generally applicable and portable across different scenarios.

Resulting challenge: rigid interfaces. Decoupling optimizations comes with a cost, as we lose the granularity and internal application knowledge present in system-specific optimizations. Specifically, the operation model of conventional I/O stacks requires layers to communicate through rigid interfaces that cannot be easily extended, discarding information that could be used to classify and differentiate requests at different levels of granularity [13]. For instance, let us consider the I/O stack depicted in Fig. 1 made of an *Application*, a *KVS*, and a POSIX-compliant *File System*. POSIX operations submitted from the *KVS* can be originated from different workflows, including foreground (a) and background flows *i.e.,* flushes (b) and compactions (c). The *File System* however, can only observe the request's size and type (*i.e.*, read and write), making it impossible to infer its origin. Implementing SILK's I/O scheduler at a lower layer (*e.g., File System*, layer between the *KVS* and the *File System*), would make the optimization portable to other KVS solutions. However, it would be ineffective since it could not differentiate between foreground and background operations.

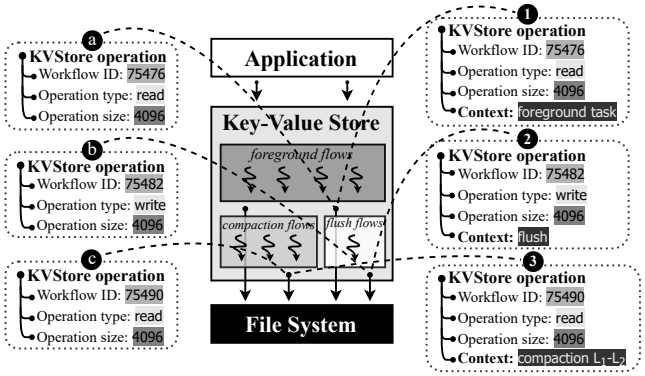


Figure 1: ***Operations submitted from different workflows.*** *Example of the operation flow of a multi-layered I/O stack. Left side depicts the regular information that can be extracted from operations between the KVS and File System, while the right side propagates additional request information throughout layers.*

Solution: information propagation. Application-level information must be propagated throughout layers to ensure that decoupled optimizations can provide the same level of control and performance as system-specific ones.

Resulting challenge: kernel-level layers. While implementing SILK's I/O scheduler at the kernel (*e.g.,* file system, block layer) would promote its applicability across other KVS solutions, it poses several disadvantages. First, for application-level information to be propagated to these layers, it requires breaking user-to-kernel (*i.e.,* POSIX) and kernel-internal interfaces (*e.g.,* VFS, block layer, page cache), decreasing portability and compatibility [13]. Further, kernel-level development is more restricted and error prone than in user-level [42, 56]. Finally, these optimizations would be ineffective under kernel-bypass storage stacks (*e.g.,* SPDK [10], PMDK [8]), since I/O requests are submitted directly from the application (user-space) to the storage device.

Solution: actuate at user-level. I/O optimizations should be implemented at a dedicated user-level layer, promoting portability across different systems and scenarios, and easing information propagation throughout layers.

Problem 2: partial visibility. Optimizations implemented in isolation are oblivious of other systems that compete for the same storage resources. Under shared infrastructures (*e.g.,* cloud, HPC), this lack of coordination can lead to conflicting optimizations [27, 62], I/O contention, and performance variation for both applications and storage backends [51, 61].

Solution: global control. Optimizations should be aware of the surrounding environment and operate in coordination to ensure holistic control of I/O workflows and shared resources.

3 PAIO in a Nutshell

PAIO is a framework that enables system designers to build custom-made SDS data plane stages. A data plane stage built with PAIO targets the workflows of a given user-level layer, enabling the classification and differentiation of requests and the enforcement of different storage mechanisms according to user-defined storage policies. Examples of such policies can be as simple as rate limiting greedy tenants to achieve resource fairness, to more complex ones as coordinating workflows with different priorities to ensure sustained tail latency. PAIO's design is built over five core principles.

General applicability. To ensure applicability across different I/O layers, PAIO stages are disaggregated from the internal system logic, contrary to tightly coupled solutions.

Programmable building blocks. PAIO follows a decoupled design that separates the I/O mechanisms from the policies that govern them, and provides the necessary abstractions for building new storage optimizations to employ over requests.

Fine-grained I/O control. PAIO classifies, differentiates, and enforces I/O requests with different levels of granularity, enabling a broad set of policies to be applied over the I/O stack.

Stage coordination. To ensure stages have coordinated access to resources, PAIO exposes a control interface that enables the control plane to dynamically adapt each stage to new policies and workload variations.

Low intrusiveness. Porting I/O layers to use PAIO requires none to minor code changes.

3.1 Abstractions in PAIO

PAIO uses four main abstractions, namely *enforcement objects*, *channels*, *context*, and *rules*.

Enforcement object. An enforcement object is a self-contained, single-purposed mechanism that applies custom I/O logic over incoming I/O requests. Examples of such mechanisms can range from *performance control* and *resource management* such as token-buckets and caches, *data transformations* as compression and encryption, to *data management* (*e.g.,* data prefetching, tiering). This abstraction provides to system designers the flexibility and extensibility for developing new mechanisms tailored for enforcing specific storage policies.

Channel. A channel is a stream-like abstraction through which requests flow. Each channel contains one or more enforcement objects (*e.g.,* to apply different mechanisms over the same set of requests) and a *differentiation rule* that maps requests to the respective enforcement object to be enforced.

Context object. A context object contains metadata that characterizes a request. It includes a set of elements (or *classifiers*), such as the *workflow id* (*e.g.,* thread-ID), *request type* (*e.g.,* `read`, `open`, `put`, `get`), *request size*, and the *request context*, which is used to express additional information of a given request, such as determining its origin, context, and more. For each request, PAIO generates the corresponding *Context* object that is used for classifying, differentiating, and enforcing the request over the respective I/O mechanisms.

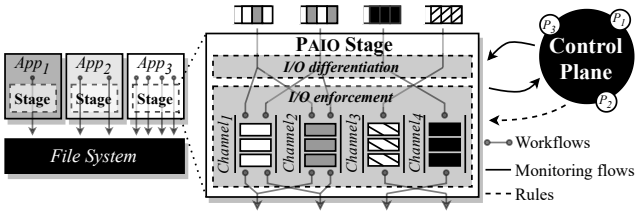


Figure 2: PAIO *overview.* PAIO *is a user-level framework that allows implementing programmable and adaptable data plane stages.*

Rule. In PAIO, a rule represents an action that controls the state of a data plane stage. Rules are submitted by the control plane and are organized in three types: *housekeeping rules* manage the internal stage organization, *differentiation rules* classify and differentiate I/O requests, *enforcement rules* adjust enforcement objects upon workload variations.

3.2 High-level Architecture

Fig. 2 outlines PAIO's high-level architecture. It follows a decoupled design that separates policies, implemented at an external control plane, from the mechanisms that enforce them, implemented at the data plane stage. PAIO targets I/O layers at the user-level. Stages are embedded within layers, intercepting all I/O requests and enforcing user-defined policies. To achieve this, PAIO is organized in four main components.

Stage interface. Applications access stages through a stage interface (§6.1) that routes all requests to PAIO before being submitted to the next I/O layer (*i.e., App*$_3$ $\rightarrow$PAIO $\rightarrow$*File System*). For each request, it generates a *Context* object with the corresponding I/O classifiers.

Differentiation module. The differentiation module (§4) classifies and differentiates requests based on their *Context* object. To ensure requests are differentiated with fine-granularity, we combine ideas from *context propagation* [36] to enable application-level information, only accessible to the layer itself, to be propagated to PAIO, broadening the set of policies that can be enforced.

Enforcement module. The enforcement module (§5) is responsible for applying the actual I/O mechanisms over requests. It is organized with channels and enforcement objects. For each request, the module selects the channel and enforcement object that should handle it. After being enforced, requests are returned to the original data path and submitted to the next I/O layer (*File System*).

Control interface. PAIO exposes a control interface (§6.1) that enables the control plane to (1) orchestrate the stage lifecycle by creating channels, enforcement objects, and differentiation rules, and (2) ensure all policies are met by continuously monitoring and fine-tuning the stage. The control plane provides global visibility, ensuring that stages are controlled holisti-

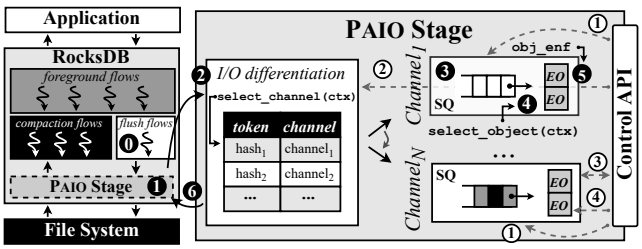


Figure 3: PAIO *operation flow.* *Black circles depict the execution flow of a request in the* PAIO *stage. White circles depict the control flow between the SDS control plane and the stage.*

cally. Exposing this interface allows stages to be managed by existing control planes [22, 35, 54].

3.3 A Day in the Life of a Request

Before delving into PAIO's internal modules, we first illustrate how it orchestrates the workflows of a given layer. We consider the I/O stack depicted on Fig. 3, which is made of an *Application*, *RocksDB*, a PAIO stage, and a POSIX-compliant *File System*; and the enforcement of the following policy: *"limit the rate of RocksDB's flush operations to X MiB/s"*. *RocksDB*'s background workflows generate flush and compaction jobs, which are translated in multiple POSIX operations that are submitted to the *File System*. Flushes are translated in `writes`, while compactions in `reads` and `writes`.

At startup time, *RocksDB* initializes the PAIO stage, which connects to an already deployed control plane. The control plane submits housekeeping rules to create a channel and an enforcement object that rate limits requests at *X* MiB/s (①). It also submits differentiation rules (②) to determine which requests should be handled by the stage, namely flush-based `writes`. Details on how the *differentiation* and *enforcement* processes work are given in §4 and §5, respectively.

At execution time, *RocksDB* propagates the context at which a given operation is created (❼) and redirects all `write` operations to PAIO (❶). Through ❶, we ensure that only `write` operations are enforced at PAIO, while with ❼, we differentiate flush-marked `writes` from others that can be triggered by compactions jobs. Upon a flush-based `write`, a *Context* object is created with its *request type* (`write`), *context* (`flush`), and *size*, and submitted, along the request, to the stage (❶). Then, the stage selects the channel (❷) to be used, enqueues the request (❸), and selects the enforcement object to service the request (❹), which in turn rate limits the request at *X* MiB/s (❺). After enforcing the request (❻), the original `write` operation is submitted to the *File System* .

The control plane continuously monitors and fine-tunes the data plane stage. Periodically, it collects from the stage the throughput at which requests are being serviced (③). Based on this metric, the control plane may adjust the enforcement object to ensure flush operations flow at *X* MiB/s, generating enforcement rules with new configurations (④).

Table 1: *Examples of the type of requests a channel receives.*

Channel	Workflow ID	Request context	Request type
$channel_1$	$flow_1$	—	—
$channel_2$	—	*background tasks*	`read`
$channel_3$	$flow_5$	*compaction*	`write`

4 I/O Differentiation

PAIO's differentiation module provides the means to classify and differentiate requests at different levels of granularity, namely *per-workflow*, *request type*, and *request context*. The process for differentiating requests is achieved in three phases.

Startup time. At startup time, the user defines *how* requests are differentiated and *who* should handle each request. First, it defines the granularity of the differentiation, by specifying which I/O classifiers should be used to differentiate requests. For example, to provide per-workflow differentiation PAIO only considers the *Context's workflow id* classifier, while to differentiate requests based on their context and type, it uses both *request context* and *request type* classifiers. Second, the user attributes specific I/O classifiers to each channel to determine the set of requests that a given channel receives. Table 1 provides examples of this specification: $channel_1$ only receives requests from $flow_1$, while $channel_2$ only handles `read` requests originated from *background tasks*; $channel_3$ receives compaction-based `writes` from $flow_5$. To generate a unique identifier that maps requests to channels, the classifiers can be concatenated into a string or hashed into a fixed-size token (§7). Further, this process can be set by the control plane (*i.e.,* differentiation rules) or configured at stage creation.

Execution time. The second phase differentiates the I/O requests submitted to the stage and routes them to the respective channel to be enforced. This is achieved in two steps.

Channel selection. For each incoming request, which is accompanied by its *Context* object, PAIO selects the channel that must service it (Fig. 3, ❷). PAIO verifies the *Context*'s I/O classifiers and maps the request to the respective channel to be enforced. This mapping is done as described in the first phase of the differentiation process.

Enforcement object selection. As each channel can contain multiple enforcement objects, analogously to channel selection, PAIO selects the correct object to service the request (Fig. 3, ❹). For each request, the channel verifies the *Context*'s classifiers and maps the request to the respective enforcement object, which will then employ its I/O mechanism (§5).

Context propagation. Several I/O classifiers, such as *workflow id*, *request type*, and *size*, are accessible from observing raw I/O requests. However, application-level information, that is only accessible to the layer that submits the I/O requests, could be used to expand the policies to be enforced over the I/O stack. An example of such information, as depicted in Fig.1, is the *operation context*, which allows to determine the origin or context of a given request, *i.e.,* if it comes from a foreground or background task, flush or compaction, or other.

As such, PAIO enables the propagation of additional information from the targeted layer to the stage. It combines ideas from *context propagation*, a commonly used technique that enables a system to forward context along its execution path [36, 37, 41, 62], and applies them to ensure fine-grained control over requests. To achieve this, system designers instrument the data path of the targeted layer where the information can be accessed, and make it available to the stage through the process's address space, shared memory, or thread-local variables. The information is included at the creation of the *Context* object as the *request context* classifier. Propagating the context without this method would require changing all core modules and function signatures between where the information can be found and its submission to the stage.

As an example, consider the I/O stack of Fig. 3. To determine the origin of POSIX operations submitted by *RocksDB*'s background workflows, system designers instrument the *RocksDB*'s critical path responsible for managing flush or compaction jobs (❶) to capture their context. This information is then propagated to the *stage interface*, where the *Context* object is created with all I/O classifiers, including the *request context*, and submitted to the stage (❶).

Note that *this step is optional*, as it can be skipped for policies that do not require additional information to be enforced.

5 I/O Enforcement

The enforcement module provides the building block for developing the actual I/O mechanisms that will be employed over requests. It is composed of several channels, each containing one or more enforcement objects.

As depicted in Fig. 3, requests are moved to the selected channel and placed in a *submission queue* (❸). For each dequeued request, PAIO selects the correct enforcement object (❹) and applies its I/O mechanism (❺). Examples of these mechanisms include token-buckets, caches, encryption schemes, and more; we discuss how to build enforcement objects in §6.3 Since several mechanisms can change the original request's state, such as data transformations (*e.g.,* encryption, compression), during this phase, the enforcement object generates a *Result* that encapsulates the updated version of the request, including its content and size. The *Result* object is then returned to the stage interface, that unmarshalls it, inspects it, and routes it to the original data path (❻). After this process, PAIO ensured that the request has met the objectives of the specified policy.

Optimizations. Depending on the policies and mechanisms to be employed, PAIO can enforce requests using only their I/O classifiers. While data transformations are directly applicable over the request's content, performance-driven mechanisms such as token-buckets and schedulers, only require specific request metadata to be enforced (*e.g.,* type, size, priority, storage path). As such, to avoid adding overhead to the system execution, PAIO allows for the request's content to be copied to the stage's execution path only when necessary.

Table 2: Interface definitions of PAIO.

1†	`paio_init()`	Initialization of PAIO stage
	`enforce(ctx,r)`	Enforce context `ctx` and request `r`
2*	`obj_init(s)`	Initialize enforcement object with state `s`
	`obj_enf(ctx,r)`	Enforce I/O mechanism over `ctx` and `r`
	`obj_config(s)`	Configure enforcement object with state `s`
3*	`stage_info()`	Get data plane stage information
	`hsk_rule(t)`	Housekeeping rule with tuple `t`
	`dif_rule(t)`	Differentiation rule with tuple `t`
	`enf_rule(id,s)`	Enf. rule over enf. object `id` with state `s`
	`collect()`	Collect statistics from data plane stage

†**Stage API**; *__Enforcement object API__; *__Control API__.

6 PAIO Interfaces and Usage

We now detail how PAIO interacts with I/O layers and control planes, how to integrate PAIO in user-level layers, and how to build enforcement objects.

6.1 Interfaces

Stage interface. PAIO provides an application programming interface to establish the connection between an I/O layer and PAIO's internal mechanisms. As depicted in Table 2, it presents two functions: `paio_init` initializes a stage, which connects to the control plane for internal stage management and defining how workflows should be handled; `enforce` intercepts requests from the layer and routes them, along the associated *Context* object, to the stage (§6.2 details how requests should be intercepted and submitted to PAIO). After enforcing the request, the stage outputs the enforcement result and the layer resumes the original execution path.

Control interface. Communication between stages and the control plane is achieved through five calls, as depicted in Table 2. A `stage_info` call lists information about the stage, including the *stage identifier* and *process identifier* (PID). Rule-based calls are used for managing and tuning the data plane stage. *Housekeeping rules* (`hsk_rule`) manage the stage lifecycle (*e.g.,* create channels and enforcement objects), *differentiation rules* (`dif_rule`) map requests to channels and enforcement objects, and *enforcement rules* (`enf_rule`) dynamically adjust the internal state (`s`) of a given enforcement object (`id`) upon workload and policy variations. The control plane also monitors stages though a `collect` call, that gathers key performance metrics of all workflows (*e.g.,* IOPS, bandwidth) and can be used to tune the data plane stage.

This interface enables the control plane to define how PAIO stages handle I/O requests. Nonetheless, concerns related to the dependability of data plane stages, as well as the resolution of conflicting policies are responsibility of the control plane [38], and are thus orthogonal to this paper.

6.2 Integrating PAIO in User-level Layers

Porting I/O layers to use PAIO stages can require a few steps.

Using PAIO with context propagation. To integrate a stage within a layer, the system designer typically needs to:

1. Create the stage in the targeted layer, using `paio_init`.
2. Instrument the critical data path, where the layer-level information is accessible, and propagate it to the stage upon the *Context* object creation. This might entail creating additional data structures.
3. Create the *Context* object that will be submitted, alongside the request, to the stage. It can include the *workflow id*, *request type* and *size*, and the propagated information.
4. Add an `enforce` call to the I/O operations that need to be enforced at the stage before being submitted to the next layer. For example, to enforce the POSIX `read` operations of a given layer, all `read` calls need to be first routed to PAIO before being submitted to the file system.
5. Verify if the request was successfully enforced by inspecting the *Result* object, returned from `enforce`, and resume the execution path.

Using PAIO transparently. When context propagation is not required, PAIO stages can be used transparently between I/O layers, such as applications and file systems. PAIO exposes layer-oriented interfaces (*e.g.,* POSIX) and uses `LD_PRELOAD` to replace the original interface calls at the top layer (*e.g.,* `read` and `write` calls invoked by applications) for ones that are first submitted to PAIO before being submitted to the bottom layer (*e.g.,* file system) [7]. Each supported call defines the logic to create the *Context* object, submits the request to the stage, verifies the *Result*, and invokes the original I/O call. This enables layers to use PAIO without changing any line of code.

6.3 Building Enforcement Objects

PAIO exposes to system designers a simple API to build enforcement objects, as depicted in Table 2.

- **`obj_init`.** Create an enforcement object with initial state `s`, which includes its type and initial configuration.
- **`obj_config`.** Provides the tuning knobs to update the enforcement object's internal settings with a new state `s`. This enables the control plane to dynamically adapt it to workload variations and new policies.
- **`obj_enf`.** Implements the actual I/O logic to be applied over requests. It returns a *Result* that contains the updated version of the request (`r`), after applying its logic. It also receives a *Context* object (`ctx`) that is used to employ different actions over the I/O request.

By default, PAIO preserves the operation logic of the targeted system (*e.g.,* ordering, error handling), as both enforcement objects and operations submitted to PAIO follow a synchronous model. While developing asynchronous enforcement objects is feasible, one needs to ensure that both correctness and fault tolerance guarantees are preserved.

7 Implementation

We have implemented PAIO prototype with 9K lines of C++ code. It targets layers at the user-level, enabling the construction of new stage implementations and simple integration, requiring none or minor code changes.

Enforcement objects. We implemented two enforcement objects. `Noop` implements a pass-through mechanism that copies the request's content to the *Result* object, without additional data processing. *Dynamic rate limiter* (`DRL`) implements a token-bucket to control the rate and burstiness of I/O workflows [17]. The bucket is configured with a maximum token capacity (*size*) and period to replenish the bucket (*refill period*). The rate at which the bucket serves requests is given in *tokens/s*. On `obj_init` the bucket is created with an initial *size* and *refill period*. On `obj_config`, a `rate(r)` routine changes the *size* according to a function between `r` and *refill period*. For each request, `obj_enf` verifies the *context's size* classifier and computes the number of tokens to be consumed. If not enough tokens are available, the request waits for the bucket to be refilled. To demonstrate the portability and maintainability of PAIO's I/O mechanisms, we apply the `DRL` object over two use cases composed of different layers and objectives.

I/O cost. We consider a constant cost for requests *e.g.,* each byte of a `read` or `write` request represents a token. Although the cost depends on several factors (*e.g.,* workload, type, cache hits), we continuously calibrate the token-bucket so its rate converges to the policies' goal. Our experiments show that this approach works well in our scenarios, as the bucket's rate converges within few interactions with the control plane. Nevertheless, determining the I/O cost is complementary to our work [24,50]. Combining PAIO with these could be useful under scenarios where policies are sensitive to the I/O cost.

Statistics, communication, and differentiation. PAIO implements per-workflow statistic counters at channels to record the bandwidth of intercepted requests, number of operations, and mean throughput between collection periods. Communication between the control plane and stages is established through UNIX Domain Sockets. To create unique identifiers that map requests to channels and enforcement objects, we used a computationally cheap hashing scheme [14] (*i.e.,* MurmurHash3) that hashes classifiers into a fixed-size token.

Context propagation. To propagate information from layers, we implemented a shared map, indexed by the *workflow identifier* (*e.g.,* thread-id), that stores the *context* of the requests being submitted, which is similar to those used in [36, 37].

Transparently intercepting I/O calls. PAIO uses `LD_PRELOAD` to intercept POSIX calls and route them either to the stage or to the kernel. It supports `read` and `write` calls, including different variations (*e.g.,* `pread`, `pwrite64`). We found that supporting this set of calls is sufficient to enforce data-oriented policies, as presented in §8.2. We defer the support of other calls and interfaces (*e.g.,* KVS, object store) to future work.

Control plane. We built a simple but fully-functional control plane with 3.6K lines of C++ code that enforces policies for the two use cases of this paper (§8). Policies were implemented as control algorithms. To calibrate enforcement objects, besides stage statistics, it collects I/O metrics generated by the targeted layer from the `/proc` file system [44]. Specifically, it inspects the `read_bytes` and `write_bytes` I/O counters, which represent the number of bytes read/written from/to the block layer, and compares them with the stage statistics to converge to the targeted performance goal.

8 Use Cases and Control Algorithms

We now present two use cases that showcase the applicability of PAIO for different applications and performance goals.

8.1 Tail Latency Control in Key-Value Stores

LSM KVSs [34] (*e.g.,* RocksDB) use *foreground flows* to attend client requests, which are enqueued and served in FIFO order. *Background flows* serve internal operations, namely flushes and compactions. Flushes are sequentially written to the first level of the tree (L_0) and only proceed when there is enough space. Compactions are held in a FIFO queue, waiting to be executed by a dedicated thread pool. Except for low level compactions ($L_0 \rightarrow L_1$), these can be made in parallel. A common problem of these however, is the interference between I/O workflows, generating latency spikes for client requests. Latency spikes occur when flushes cannot proceed because $L_0 \rightarrow L_1$ compactions and flushes are slow or on hold [16].

SILK. SILK [16], a RocksDB-based KVS, prevents this through an I/O scheduler that: allocates bandwidth for internal operations when client load is low; prioritizes flushes and low level compactions, as they impact client latency; and preempts high level compactions with low level ones. It employs these techniques through the following control algorithm. As these KVSs are embedded, the KVS I/O bandwidth is bounded to a given rate (KVS_B). It monitors clients' bandwidth (Fg), and allocates leftover bandwidth ($left_B$) to internal operations (I_B), given by $I_B = KVS_B - Fg$. To enforce rate I_B, SILK uses RocksDB's rate limiters [4]. Flushes and $L_0 \rightarrow L_1$ compactions have high priority and are provisioned with minimum I/O bandwidth (min_B). High level compactions have low priority and can be paused at any time. Because all compactions share the same thread pool, it is possible that, at some point, all threads are handling high level compactions. As such, SILK preempts one of them to execute low level compactions.

Applying these optimizations however, required reorganizing RocksDB's internal operation flow, changing core modules made of thousands of LoC including *background operation handlers*, *internal queuing logic*, and *thread pools allocated for internal work* [15]. Further, porting these optimizations to other KVS that would equally benefit from them, such as LevelDB [21] and PebblesDB [47], requires deep system knowledge and substantial re-implementation efforts.

Algorithm 1 Tail Latency Control Algorithm

Initialize: $KVS_B = 200$; $min_B = 10$
1: $\{Fg, Fl, L_0, L_N\} \leftarrow collect\ ()$
2: $left_B \leftarrow KVS_B - Fg$
3: $left_B \leftarrow max\ \{left_B \mid min_B\}$
4: **if** $Fl > 0 \wedge L_0 > 0$ **then**
5: $\quad \{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{left_B/2,\ left_B/2,\ min_B\}$
6: **else if** $Fl > 0 \wedge L_0 = 0$ **then**
7: $\quad \{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{left_B,\ min_B,\ min_B\}$
8: **else if** $Fl = 0 \wedge L_0 > 0$ **then**
9: $\quad \{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{\ min_B,\ left_B,\ min_B\}$
10: **else**
11: $\quad \{B_{Fl},\ B_{L_0},\ B_{L_N}\} \leftarrow \{\ min_B,\ min_B,\ left_B\}$
12: *enf_rule* $(\{B_{Fl},\ B_{L_0},\ B_{L_N}\})$
13: *sleep (loop_interval)*

PAIO. Rather than modifying the RocksDB engine, we found that several of these optimizations could be achieved by orchestrating the I/O workflows. Thus, we applied SILK's design principles as follows: a PAIO *data plane stage provides the I/O mechanisms for prioritizing and rate limiting background flows*, while the *control plane re-implements the I/O scheduling algorithm* to orchestrate the stage.

The stage intercepts all RocksDB workflows. We consider each RocksDB thread that interacts with the file system as a workflow. Channel differentiation is made using the *workflow id*. We instrumented RocksDB to propagate the *context* at which a given operation is created, namely flush (`flush`) or compaction (*e.g.,* `compaction_L0_L1`). Foreground flows are monitored for collecting clients' bandwidth (Fg). Background flows are routed to channels made of `DRL` objects. Flushes flow through a dedicated channel. As compactions with different priorities can flow through the same channel, each channel contains two `DRL` objects configured at different rates. The enforcement object differentiation is made through the *request context* classifier, and requests are enforced with the optimization described in §5. PAIO also collects the bandwidth of flushes (Fl), and low (L_0) and high level compactions (L_N).

The control plane implements the control portion of SILK's scheduling algorithm (Alg. 1). It uses a feedback control loop that performs the following steps. First, it collects statistics from the stage (`1`) and computes leftover disk bandwidth ($left_B$) to assign to internal operations (`2`). To ensure that background operations keep flowing, it defines a minimum bandwidth threshold (`3`), and distributes $left_B$ according to workflow priorities (`4-11`). If high priority tasks are executing it assigns them an equal share of $left_B$, while ensuring that high level compactions keep flowing (min_B), preventing low level ones from being blocked in the queue (`5`). If a single high priority task is being executed, $left_B$ is allocated to it and min_B to others (`6-9`). If no high priority task is executing, it reserves $left_B$ to low priority ones (`11`). It then generates and submits `enf_rule`s to adjust the rate of each enforcement object (`12`). For low priority compactions, it splits B_{L_N} between all `DRL`

Table 3: *Lines of code added to RocksDB and TensorFlow.*

	Lines added	
	RocksDB	TensorFlow (`LD_PRELOAD`)
Targeted code base size	≈335K [5]	≈2.3M [6]
Initialize PAIO stage	10	—
Context propagation	47	—
Create *Context* object	7	—
Instrument I/O calls	17	—
Verify *Result* object	4	—
Total	**85**	**0**

objects that handle these. Since high priority compactions are executed sequentially [9, 16], it assigns B_{L_0} to the respective objects. Rate B_{Fl} is assigned to those responsible for flushes.

Integration with RocksDB. Integrating PAIO in RocksDB only required adding 85 LoC (Table 3). Specifically:

1. Initialize PAIO stage and create additional structures to identify the task that each workflow is executing (10 LoC).
2. Instrument RocksDB's internal thread pools for identifying the workflows that run flush and compaction jobs (17 LoC). To differentiate high priority compactions from low priority ones, we instrumented the code where compaction jobs are created. For each job, we verify its level and update the structure with the task that the workflow will be executing (*e.g.,* `compaction_L0_L1`) (30 LoC).
3. Create a *Context* object with *workflow id*, *request type*, *context*, and *size* I/O classifiers (7 LoC).
4. Submit all `read` and `write` calls to the stage (17 LoC).
5. Verify the *Result* of the enforcement (4 LoC).

8.2 Per-Application Bandwidth Control

The ABCI supercomputer is designed upon the convergence between AI and HPC workloads. One of the most used AI frameworks on it is TensorFlow [11]. To execute TensorFlow jobs users can reserve a full node or a fraction of it (*i.e.,* jobs execute concurrently). Nodes are partitioned into resource-isolated *instances* through Linux's cgroups [39]. Each instance has exclusive access to CPU cores, memory space, a GPU, and local storage quota. However, the local disk bandwidth is still shared, and because each instance is agnostic of others, jobs compete for bandwidth leading to I/O interference and performance variation. Even if the block I/O scheduler is fair, all instances are provisioned with the same service level, preventing the assignment of different priorities.

Using cgroups's block I/O controller (*blkio*) allows static rate limiting `read` and `write` operations of each instance [2]. However, under ABCI, once the rate is set it cannot be dynamically changed at execution time, as it requires stopping the jobs, adjust the rate of all groups, and restart the jobs, being prohibitively expensive in terms of overall execution time. This creates a second problem where if no other job is executing in the node, the instance cannot use leftover bandwidth.

PAIO. To address this, we use a PAIO stage that implements the mechanisms to dynamically rate limit workflows at each instance, while the control plane implements a proportional sharing algorithm to ensure all instances meet their policies.

Algorithm 2 Max-min Fair Share Control Algorithm

Initialize: $Max_B = 1GiB$; $Active > 0$; $demand_i > 0$
1: $\{I_1, I_2, I_3, I_4\} \leftarrow collect\ ()$
2: $left_B \leftarrow Max_B$
3: **for** $i = 0$ in $[0, Active-1]$ **do**
4: **if** $demand_i \leq \frac{left_B}{Active-i}$ **then**
5: $rate_i \leftarrow demand_i$
6: **else**
7: $rate_i \leftarrow \frac{left_B}{Active-i}$
8: $left_B \leftarrow left_B - rate_i$
9: **for** $i = 0$ in $[0, Active-1]$ **do**
10: $rate_i \leftarrow \frac{left_B}{Active}$
11: $enf\_rule\ (\{rate_1, I_1\}, \{rate_2, I_2\}, \{rate_3, I_3\}, \{rate_4, I_4\})$
12: $sleep\ (loop\_interval)$

Our use case focuses on the model training phase, where each instance runs a TensorFlow job that uses a single workflow to read dataset files from the file system. TensorFlow's `read` requests are intercepted and routed to the stage, which contains a channel with a `DRL` enforcement object. Requests are enforced with the optimization described in §5.

The control plane implements a max-min fair share algorithm to ensure per-application bandwidth guarantees (Alg. 2), which is typically used for resource fairness policies [35, 54]. The overall disk bandwidth available (Max_B) and bandwidth demand of each application (*demand*) are defined *a priori* by the system administrator or the mechanism responsible for managing resources of different job instances [63]. The algorithm uses a feedback control loop that performs the following steps. First, the control plane collects statistics from each active instance's stage, given by I_i (`1`), as well as the bandwidth generated by each TensorFlow job (collected at `/proc`). Then, it computes the rate of each active instance (`3-10`). If an instance's *demand* is less than its fair share, the control plane assigns its *demand* (`4-5`), assigning the fair share otherwise (`7`). It then distributes leftover bandwidth ($left_B$) across instances (`9-10`). Then, it calibrates the rate of each instance in a function of I_i and $rate_i$, generating the `enf_rules` to be submitted to each stage (`11`). Finally, the control plane sleeps for *loop_interval* before beginning a new control cycle (`12`).

Integration with TensorFlow. Integrating TensorFlow with PAIO did not required any code changes (Table 3). We used `LD_PRELOAD` to intercept, and route to PAIO, TensorFlow's `read` and `write` calls. All supported calls implement the logic necessary for the request to be enforced, including the creation of the *Context* object using the *request type* and *size* classifiers; stage enforcement; verification of the enforcement *Result*; and its submission to the original execution path (file system).

9 Evaluation

Our evaluation seeks to demonstrate the performance of PAIO, and its ability and feasibility of enforcing policies over different scenarios. The results show that:

- Its performance scales with the number of channels, achieving high throughput and low latency (§9.1).
- It can be used to enforce policies over different I/O layers with distinct requirements (§9.2 and §9.3).
- By propagating application-level information to the data plane stage, PAIO outperforms RocksDB by at most 4× in tail latency, while enabling similar control and performance as system-specific optimizations (SILK) (§9.2).
- When internal system knowledge is not required, PAIO can enforce policies without application changes. By having global visibility, it provisions per-application bandwidth guarantees at all times, and improves overall execution time when compared to a static rate limiting approach (§9.3).

Experimental setting. Experiments were conducted under two hardware configurations. ***A***: a compute node of the ABCI supercomputer with two 20-core Intel Xeon processors (80 cores), 4 NVidia Tesla V100 GPUs, 384GiB of RAM, and a 1.6TiB Intel SSD DC P4600, running CentOS 7.5 with Linux kernel 3.10 and the `xfs` file system. ***B***: a server with two 18-core Intel Xeon processors (72 cores), 192GiB of RAM, a 1.6TiB Dell Express Flash PM1725b SSD (NVMe) and a 480GiB Intel D3-s4610 SATA SSD, running Ubuntu Server 20.04 LTS with kernel 5.8.9 and the `ext4` file system.

9.1 PAIO Performance and Scalability

We developed a benchmark that simulates an application that submits requests to a PAIO stage. This benchmark aims to demonstrate the maximum performance achievable with PAIO by stress-testing it in a loop-back manner. It generates and submits multi-threaded requests in a closed loop through the *Instance*'s `enforce` call, under a varying number of clients (*e.g.,* workflows) and request sizes. Request size and number of client threads range between 0 – 128KiB and 1 – 128, respectively. Each client thread submits 100M requests. A PAIO stage is configured with varying number of channels (matching the number of client threads), each containing a `Noop` enforcement object that copies the request's buffer to the *result* object. All reported results are the mean of at least ten runs and standard deviation is kept below 5%.

IOPS and bandwidth. Fig. 4 depicts the cumulative IOPS ratio with respect to a single channel. 0B represents a *context*-only request, as described in §5. Results marked with ∗ and + were conducted under configurations *A* and *B*, respectively.

For configuration *A*, under a 0B* request size, a single PAIO channel achieves a mean throughput of 3.05 MOps/s and a 327 ns latency. Since the workload is CPU-bound, the performance does not scale linearly, as client threads compete for processing time. Under 128 channels, it achieves a cumulative throughput of 97.4 MOps/s, a 31× performance increase. As the request size increases so does the total bytes processed by PAIO. When configured with 128 channels, it processes 128KiB* requests at 384 GiB/s. For a single channel, PAIO processes requests at 2.1 GiB/s and 11.7 GiB/s

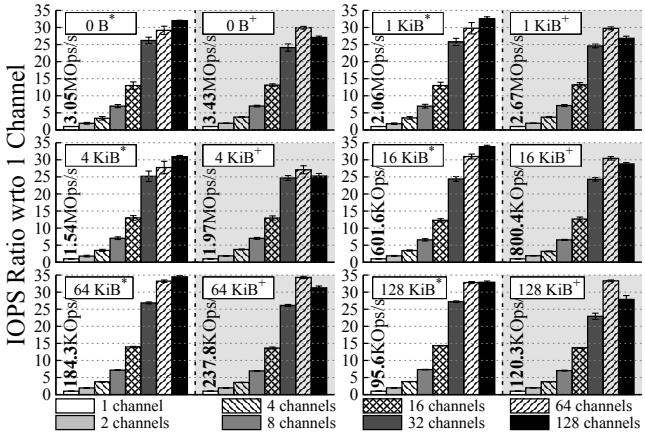


Figure 4: ***Cumulative IOPS*** *of* PAIO *under varying number of channels (1 – 128) and request sizes (0 – 128 KiB). Absolute IOPS value is shown above the 1 channel bar.*

for 1KiB* and 128KiB* request sizes. For configuration *B*, PAIO achieves higher throughput results as it operates under a later kernel version. Since the machine is configured with 72 cores, PAIO's performance peaks at 64 client threads. Under a 0B+ request size, PAIO achieves 3.43 MOps/s (1 channel) and 102.7 MOps/s (64 channels), representing a 30× performance increase. When configured with 64 channels, it is able to process 128KiB+-sized requests at 489 GiB/s. For a single channel, PAIO processes requests at 2.5 GiB/s and 14.7 GiB/s for 1KiB* and 128KiB* request sizes, respectively.

Profiling. We measured the execution time of each PAIO operation that appears in the main execution path. Depending on the hardware configuration, *Context* object creation takes between 17 – 19 ns, while the channel and enforcement object selection take 85 – 89 ns to complete (each). The duration of `obj_enf` ranges between 20 ns and 8.45 μs when configured with 0B and 128KiB request sizes.

Summary. Results show that PAIO has low overhead, as it is provided as a user-space library, which does not require costly context-switching operations. We expect that the main source of overhead will always be dependent on the type of enforcement object applied over requests. For the enforcement object used in the use cases of this paper (§9.2 – §9.3), we have not observed significant performance degradation.

9.2 Tail Latency Control in Key-Value Stores

We now demonstrate how PAIO achieves tail latency control under several workloads. We compare the performance of RocksDB [5]; Auto-tuned, a version of RocksDB with auto-tuned rate limiting of background operations enabled [28]; SILK [16]; and PAIO, *i.e.,* a PAIO-enabled RocksDB.

System configuration. Experiments were conducted under hardware configuration *B* using the available NVMe device (unless stated otherwise). All systems are tuned as follows. The `memtable-size` is set to 128MiB. We use 8 threads for client operations and 8 background threads for flush (1) and

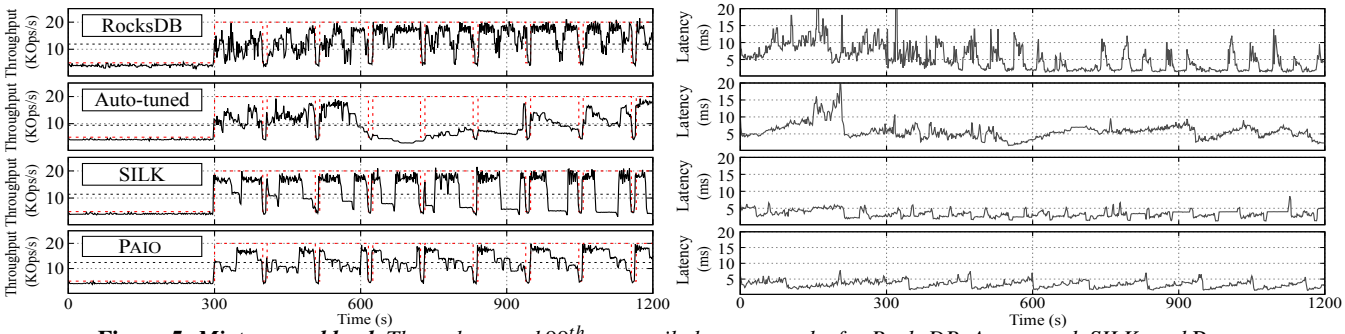


Figure 5: ***Mixture workload.*** *Throughput and 99th percentile latency results for RocksDB, Auto-tuned, SILK, and* PAIO.

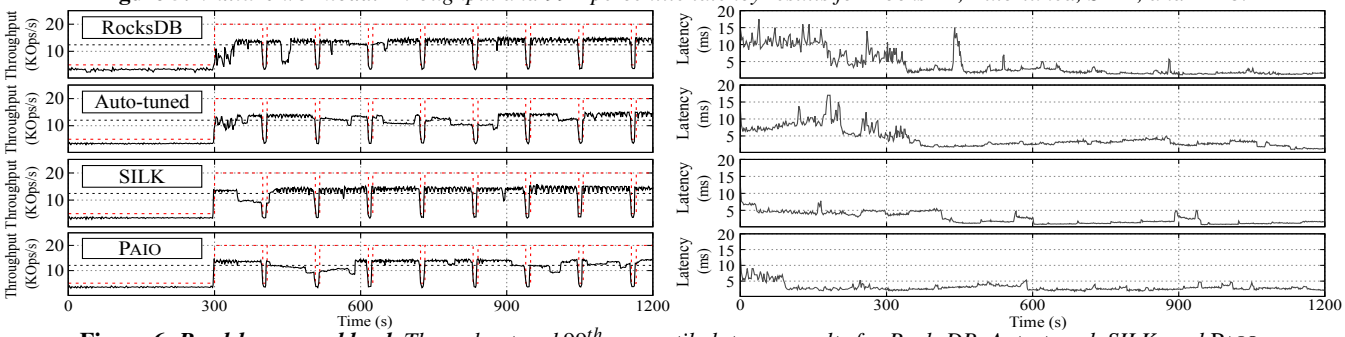


Figure 6: ***Read-heavy workload.*** *Throughput and 99th percentile latency results for RocksDB, Auto-tuned, SILK, and* PAIO.

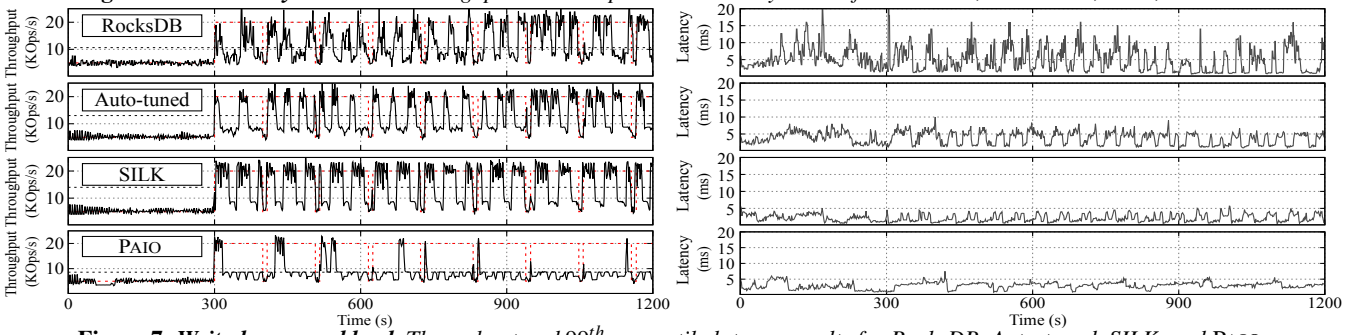


Figure 7: ***Write-heavy workload.*** *Throughput and 99th percentile latency results for RocksDB, Auto-tuned, SILK, and* PAIO.

compactions (7). The minimum bandwidth threshold for internal operations is set to 10MiB/s. To simplify results compression and commit logging are turned off. All experiments are conducted using the `db_bench` benchmark [3]. As used in the SILK testbed [16], we limit memory usage to 1GiB and I/O bandwidth to 200MiB/s (unless stated otherwise).

Workloads. We focus on workloads made of bursty clients, to better simulate existing services in production [16, 18]. Client requests are issued in a closed loop through a combination of peaks and valleys. An initial valley of 300 seconds submits operations at 5kops/s, and is used for executing the KVS internal backlog. Peaks are issued at a rate of 20kops/s for 100 seconds, followed by 10 seconds valleys at 5kops/s. All datastores were preloaded with 100M key-value pairs, using a uniform key-distribution, 8B keys and 1024B values.

We use three workloads with different read:write ratios: *mixture* (50:50), *read-heavy* (90:10), and *write-heavy* (10:90). *Mixture* represents a commonly used YCSB workload (workload A) and provides a similar ratio as Nutanix production workloads [16]. *Read-heavy* provides an operation ratio similar to those reported at Facebook [18]. To present a comprehensive testbed, we include a *write-heavy* workload. For each system, workloads were executed three times over 1-hour with uniform key distribution. For figure clarity, we present the first 20 minutes of a single run. Similar performance curves were observed for the rest of the execution. Fig. 5–9 depict throughput and 99th percentile latency of all systems and workloads. Theoretical client load is presented as a red dashed line. Mean throughput is shown as an horizontal dashed line.

Mixture workload (Fig. 5). Due to accumulated backlog of the loading phase, the throughput achieved in all systems does not match the theoretical client load. RocksDB presents high tail latency spikes due to constant flushes and low level compactions. Auto-tuned presents less latency spikes but degrades overall throughput. This is due to the rate limiter being agnostic of background tasks' priority, and because it increases its rate when there is more backlog, contending for disk bandwidth. SILK achieves low tail latency but suffers periodic drops in throughput due to accumulated backlog. Compared to RocksDB (11.9 kops/s), PAIO provides simi-

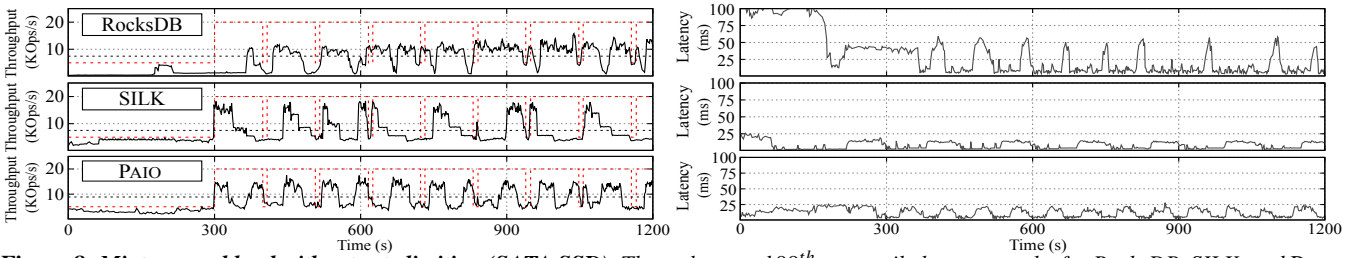


Figure 8: ***Mixture workload without rate limiting (SATA SSD).*** *Throughput and 99th percentile latency results for RocksDB, SILK, and* PAIO.

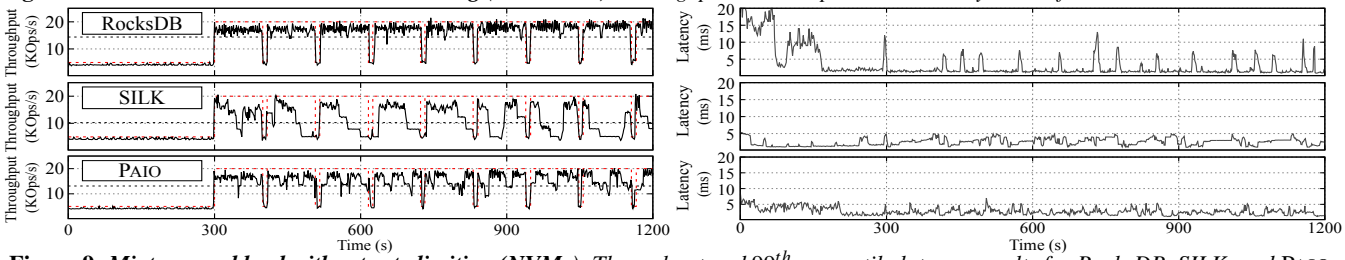


Figure 9: ***Mixture workload without rate limiting (NVMe).*** *Throughput and 99th percentile latency results for RocksDB, SILK, and* PAIO.

lar mean throughput (12.4 kops/s). As for tail latency, while RocksDB experiences peaks that range between 3–20 ms, PAIO and SILK observe a 4× decrease in absolute tail latency, with values ranging between 2–6 ms.

Read-heavy workload (Fig. 6). Throughput-wise all systems perform identically. At different periods, all systems demonstrate a temporary throughput degradation due to accumulated backlog. As for tail latency, the analysis is twofold. RocksDB and Auto-tuned present high tail latency up to the 400 s mark. After that mark, RocksDB does not have more pending backlog and achieves sustained tail latency (1–3 ms), while on Auto-tuned, some compactions are still being performed due to rate limiting, increasing latency by 1–2 ms. SILK and PAIO have similar latency curves. During the initial valley both systems significantly improve tail latency when compared to RocksDB. After the 400 s mark, SILK pauses high level compactions and achieves a tail latency between 1–2 ms. By preempting high level compactions and serving low level ones through the same thread pool as flushes, it ensures that high priority tasks are rarely stalled. SILK achieves this by modifying the RocksDB's queuing mechanism. In PAIO, while sustained, its tail latency is 1 ms higher than SILK's in the same observation period. Since PAIO does not modify the RocksDB engine, it cannot preempt compactions (§8.1).

Write-heavy workload (Fig. 7). Write-intensive workloads generate a large backlog of background tasks, leading RocksDB to experience high latency spikes. Auto-tuned limits all background writes, reducing latency spikes but still exceeding the 5 ms mark over several periods. SILK pauses high level compactions and only serves high priority tasks, improving mean throughput and keeping latency spikes below 5 ms. In PAIO, since flushes occur more frequently, the control plane slows down high level compactions more aggressively, which leads to low level ones to be temporary halted at the compaction queue, waiting to be executed. Even though mean throughput is decreased, PAIO significantly reduces tail latency, never exceeding 6 ms. The throughput difference between PAIO and SILK is justified by the latter preempting high level compactions, as described in the read-heavy workload.

Mixture workload without rate limiting (Fig. 8–9). We conducted an additional set of experiments to assess the impact of the tail latency control algorithm under a scenario where the KVS has access to the full storage device bandwidth. We compared the performance of RocksDB, SILK, and PAIO under both SSD and NVMe devices, without rate limiting, using the *mixture* workload. The KVS_B parameter was set with a value closer to the device's limit. For Auto-tuned, we report similar conclusions to those presented in Fig. 5.

Fig. 8 depicts the results under the SSD device. Due to accumulated backlog all systems experience poor throughput performance, averaging at 7.46 kops/s (RocksDB), 7.52 kops/s (SILK), and 8.88 kops/s (PAIO). During the loading phase, and until finishing the accumulated backlog (0–400 s), RocksDB experiences long periods of high tail latency, peaking at 111 ms. After that, it observes latency spikes due to constant flushes and low level compactions, with values ranging between 15–60 ms. SILK and PAIO present a more sustained latency performance, never exceeding the 25 ms mark throughout the overall observation period. Specifically, while RocksDB experienced a variability of 21 ms, SILK and PAIO achieved 4.7 ms and 5.8 ms, respectively.[2] Throughput-wise, both systems observe periodic drops due to accumulated backlog. However, PAIO is able to recover faster than SILK. Because it cannot preempt compactions, PAIO reserves more bandwidth (than SILK) to low priority compactions, ensuring that high priority tasks do not wait to be executed. As such, PAIO follows a proactive approach for assigning bandwidth to compactions, while SILK follows a reactive approach.

[2]The variability results correspond to the average of the absolute deviations of data points (*i.e.,* each tail latency measurement) from their mean.

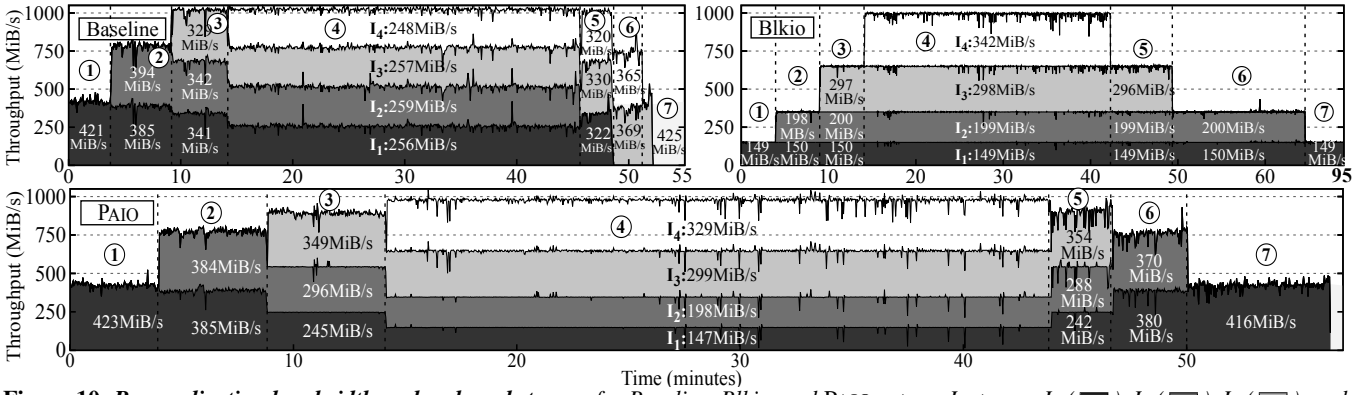


Figure 10: ***Per-application bandwidth under shared storage*** *for Baseline, Blkio, and* PAIO *setups. Instances I_1 (■), I_2 (■), I_3 (■), and I_4 (□) are assigned with minimum bandwidth of 150, 200, 300, and 350 MiB/s, and execute 6, 5, 5, and 4 training epochs, respectively.*

Fig. 9 depicts the results under the NVMe device. All systems experienced higher throughput performance, averaging at 14.39 kops/s (RocksDB), 10.27 kops/s (SILK), and 13.11 kops/s (PAIO). RocksDB follows a similar performance curve as the theoretical client load. The reason behind this is twofold. First, it completes all accumulated backlog during the initial valley (at the cost of high tail latency), which positively reflects in the remainder execution (*i.e,* no significant performance loss is observed). Second, since NVMe devices have higher throughput performance and parallelism than SSD devices (Fig. 8), RocksDB achieves a more sustained performance. After the initial valley, RocksDB observes latency spikes that range between 7–15 ms due to frequent flushes and low level compactions. SILK and PAIO follow similar tail latency curves, never exceeding the 6 ms mark. In detail, throughout the overall observation period, RocksDB observed a variability of 2.5 ms, while SILK and PAIO only observed a variability of 0.8 ms. Similarly to previous results, both system experience periodic throughput drops.

Summary. We demonstrate that through minor code changes, PAIO outperforms RocksDB by at most 4× in tail latency and enables similar control and performance as SILK, which required profound refactoring to the original code base.

9.3 Per-Application Bandwidth Control

We now demonstrate how PAIO ensures per-application bandwidth guarantees under a shared storage scenario. Our setup was driven by the requirements of the ABCI supercomputer.

System configuration. Experiments ran under hardware configuration *A* using TensorFlow 2.1.0 with the LeNet [30] training model, configured with a batch size of 64 TFRecords. We used the ImageNet dataset (≈150GiB) [48]. Each instance runs with a dedicated GPU and dataset, and its memory is limited to 32GiB. Overall disk bandwidth is limited to 1GiB/s. At all times, a node executes at most four instances with equal resource shares in terms of CPU, GPU, and RAM. Each instance executes a TensorFlow job, is assigned with a bandwidth policy, and executes a given number of training epochs. Namely, instances **1** to **4** are assigned with minimum bandwidth guarantees of **150**, **200**, **300**, and **350** MiB/s, and execute **6**, **5**, **5**, and **4** training epochs, respectively.

Setups. Experiments were conducted under three setups. *Baseline* represents the current setup supported at the ABCI supercomputer; all instances execute without bandwidth guarantees. *Blkio* enforces bandwidth limits using blkio [2]. In PAIO, each instance executes with a PAIO stage that enforces the specified bandwidth goals dynamically. Fig. 10 depicts, for each setup, the I/O bandwidth of all instances at 1-second intervals. Experiments include seven phases, each marking when an instance starts or completes its execution.

Baseline. Experiments were executed over 52 minutes. At ①, I_1 reads at 421 MiB/s. Whenever a new instance is added, the I/O bandwidth is shared evenly (②). At ③, the aggregated instance throughput matches the disk limit. At ④, instance performance converges to ≈256 MiB/s, leading to all instances experiencing the same service level. However, I_3 and I_4 cannot meet their goal, since I_1 and I_2 have more than their fair share. After 46 minutes of execution (⑤), I_3 terminates, and leftover bandwidth is shared with the remainder. Again, I_4 cannot achieve its targeted goal. At ⑥ and ⑦, active instances have access to leftover bandwidth and finish their execution.

Summary: I_3 and I_4 were unable to achieve their bandwidth guarantees, missing their objectives during 31 and 34 minutes.

Blkio. Experiments were executed over 95 minutes. From ① to ⑦, whenever a new instance is added, it is provisioned with its exact bandwidth limit. Because the rate of each instance is set using blkio, instances cannot use leftover bandwidth to speed up their execution. For example, while on *Baseline* I_1 executes under the 50-minutes mark, it takes 95 minutes to complete its execution in *Blkio*. To overcome this, a possible solution would require to stop and checkpoint the instance's execution, reconfigure blkio with a new rate, and resume from the latest checkpoint. However, doing this process every time a new instance joins or leaves the compute node would significantly delay the execution time of all running instances.

Summary: All instances achieve their bandwidth guarantees

but cannot be dynamically provisioned with available disk bandwidth, leading to longer periods of execution.

PAIO. Experiments were executed over 56 minutes. At ① and ②, instances are assigned with their proportional share, as the control plane first meets each instance demands and then distributes leftover bandwidth proportionally. At ③, contrary to *Baseline*, the control plane bounds the bandwidth of I_1 and I_2 to a mean throughput of 245 MiB/s and 296 MiB/s, respectively. At ④, instances are set with their bandwidth limit. During this phase, PAIO provides the same properties as blkio. From ⑤ to ⑦, as instances end their execution, active ones are provisioned as in ① to ③.

Summary: We show that PAIO can enforce per-application bandwidth guarantees without any code changes to applications. Contrary to *Baseline*, PAIO ensures that policies are met at all times, and whenever leftover bandwidth is available, PAIO shares it across active instances. Compared to *Blkio*, PAIO finishes 39, 15, and 3 minutes faster for I_1, I_2, and I_3.

10 Related Work

SDS systems. PAIO builds on a large body of work on SDS systems. IOFlow [54], sRoute [52], and PSLO [31] target the virtualization layer (*i.e.,* hypervisor, storage and network drivers) to enforce QoS policies. PriorityMeister [65] enforces rate limiting services at the Network File System. Mesnier *et al.* [41] employ caching optimizations at the block layer. Pisces [51] and Libra [50] enforce bandwidth guarantees in multi-tenant KVS. Malacology [49] improves the programmability of Ceph to build custom applications on top of it. Retro [35] and Cake [58] implement resource management services at the Hadoop stack. SafeFS [45] stacks FUSE-based file systems on top of each other, each providing a different service. Crystal [22] extends OpenStack Swift to implement custom services to be enforced over object requests.

All systems are targeted for specific I/O layers, as their design is *tightly coupled to* and *driven by* the architecture and specificities of the software stacks they are applied to. In contrast, PAIO is disaggregated from a specific software stack, enabling developers to build custom-made data plane stages applicable over different user-level layers, while requiring none to minor code changes — we demonstrate this by integrating PAIO over two different I/O layers (§8). Previous works are also unable to enforce the policies demonstrated in §8.1, as they do not provide context propagation [50, 51], inhibiting differentiating requests at a finer granularity (*i.e.,* foreground *vs* high-priority *vs* low-priority background tasks); or actuate at the kernel-level [41, 54], where the *context* is unreachable without significantly changing legacy APIs. Further, these are also unfit to achieve the policies demonstrated in §8.2, as solutions like [31, 52, 54] cannot be used under scenarios that require bare-metal access to resources, such as HPC infrastructures and bare-metal cloud servers.

Context propagation. Some works use context propagation techniques to tag data across kernel layers. Mesnier *et al.* [41] classifies and tags requests with classes to be differentiated at the block layer. IOFlow [54] tags requests to differentiate tenants that share the same hypervisor. Split-level scheduling [62] identifies the processes that caused a given I/O operation throughout the VFS, page cache, and block layer.

PAIO acts at the user-level and enables the propagation of additional information from the targeted I/O layer to the stage (*e.g.,* propagate the *context* at which a given request was created, as in §8.1), allowing more fine-grained differentiation and control over requests. Enabling the intended granularity by PAIO at kernel-level approaches would require breaking standard user-to-kernel and kernel-internal interfaces, reducing portability and compatibility [13]. Our contributions are also applicable under kernel-bypass storage stacks (*e.g.,* SPDK, PMDK), which is not the case for previous work.

Storage QoS. Many works ensure QoS SLOs at specific storage layers, including the block layer [2, 25, 33, 40, 57, 64], hypervisor [23, 24, 26, 31, 54], and distributed storage [46, 58–60]. These works are targeted for a specific I/O layer and storage objective. In contrast, PAIO is more general, providing a framework for building custom data plane stages applicable over different layers. Also, most of these solutions only differentiate requests based on their type. PAIO provides differentiation at workflow, request type, and request context. Approaches like [26, 33, 40, 64] follow a decoupled design that separates the QoS algorithm from the mechanism that applies it. While complementary to our work, these could be incorporated into our framework as new enforcement objects.

11 Conclusion

We have presented PAIO, a framework that enables system designers to build custom-made SDS data plane stages applicable over different I/O layers. PAIO provides differentiated treatment of requests and allows implementing storage mechanisms adaptable to different policies. By combining ideas from SDS and context propagation, we demonstrated that PAIO decouples system-specific optimizations to a more programmable environment, while enabling similar I/O control and performance, and requiring minor to none code changes.

Acknowledgments

We thank our shepherd, Sudarsun Kannan, and the anonymous reviewers for their insightful comments and feedback. We thank AIST for providing access to computational resources of ABCI. We thank Oana Balmau for discussions about SILK, and Vitor Enes and Cláudia Brito for their valuable input. This work was supported by the Portuguese Foundation for Science and Technology and the European Regional Development Fund, through the PhD Fellowship SFRH/BD/146059/2019 and projects POCI-01-0247-FEDER-045924 and UTA-EXPL/CA/0075/2019.

References

[1] AI Bridging Cloud Infrastructure. https://abci.ai/.

[2] BLKIO: Cgroup's Block I/O Controller. https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt.

[3] facebook/rocksdb: Benchmarking tools (db_bench). https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[4] facebook/rocksdb: Rate Limiter. https://github.com/facebook/rocksdb/wiki/Rate-Limiter.

[5] facebook/rocksdb: RocksDB v5.17.2. https://github.com/facebook/rocksdb/tree/v5.17.2.

[6] google/tensorflow: TensorFlow. https://github.com/tensorflow/tensorflow/tree/v2.1.0.

[7] ld.so: LD_PRELOAD. https://man7.org/linux/man-pages/man8/ld.so.8.html.

[8] Persistent Memory Development Kit. https://pmem.io/pmdk/.

[9] RocksDB: A persistent key-value store for fast storage environments. https://rocksdb.org/.

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

[11] 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 Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283. USENIX, 2016.

[12] Ian Ackerman and Saurabh Kataria. Homepage feed multi-task learning using TensorFlow. https://engineering.linkedin.com/blog/2021/homepage-feed-multi-task-learning-using-tensorflow.

[13] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Beyond Storage APIs: Provable Semantics for Storage Stacks. In *15th Workshop on Hot Topics in Operating Systems*. USENIX, 2015.

[14] Austin Appleby. appleby/smhasher: SMHasher test suite for MurmurHash family of hash functions. https://github.com/aappleby/smhasher, 2010.

[15] Oana Balmau. theoanab/SILK-USENIXATC2019: Prototype of the SILK key-value store. https://github.com/theoanab/SILK-USENIXATC2019, 2019.

[16] 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 *2019 USENIX Annual Technical Conference*, pages 753–766. USENIX, 2019.

[17] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050. Springer Science & Business Media, 2001.

[18] Zhichao Cao, Siying Dong, Sagar Vemuri, and David Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies*, pages 209–223. USENIX, 2020.

[19] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime Data Processing at Facebook. In *2016 International Conference on Management of Data*, page 1087–1098. ACM, 2016.

[20] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference*, pages 49–63. USENIX, 2020.

[21] Sanjay Ghemawat and Jeff Dean. google/leveldb: LevelDB, A Fast Key-Value Storage Library. https://github.com/google/leveldb.

[22] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. Crystal: Software-Defined Storage for Multi-Tenant Object Stores. In *15th USENIX Conference on File and Storage Technologies*, pages 243–256. USENIX, 2017.

[23] Ajay Gulati, Irfan Ahmad, and Carl A Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *7th USENIX Conference on File and Storage Technologies*, pages 85–98. USENIX, 2009.

[24] Ajay Gulati, Arif Merchant, and Peter Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *9th USENIX Symposium on Operating Systems Design and Implementation*, pages 437–450. USENIX, 2010.

[25] Ajay Gulati, Arif Merchant, and Peter J. Varman. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, page 13–24. ACM, 2007.

[26] Ajay Gulati, Ganesha Shanmuganathan, Xuechen Zhang, and Peter Varman. Demand Based Hierarchical QoS Using Storage Resource Pools. In *2012 USENIX Annual Technical Conference*. USENIX, 2012.

[27] 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*, pages 345–358. USENIX, 2017.

[28] Andrew Kryczka. RocksDB Blog: Auto-tuned Rate Limiter. https://rocksdb.org/blog/2017/12/18/17-auto-tuned-rate-limiter.html.

[29] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An Informed Storage Cache for Deep Learning. In *18th USENIX Conference on File and Storage Technologies*. USENIX, 2020.

[30] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[31] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. PSLO: Enforcing the X^{th} Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *11th European Conference on Computer Systems*. ACM, 2016.

[32] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. Elastic Machine Learning Algorithms in Amazon SageMaker. In *2020 ACM SIGMOD International Conference on Management of Data*, page 731–737. ACM, 2020.

[33] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *2nd USENIX Conference on File and Storage Technologies*. USENIX, 2003.

[34] Chen Luo and Michael J Carey. LSM-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[35] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation*, pages 589–603. USENIX, 2015.

[36] Jonathan Mace and Rodrigo Fonseca. Universal Context Propagation for Distributed System Instrumentation. In *13th European Conference on Computer Systems*. ACM, 2018.

[37] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Transactions on Computer Systems*, 35(4), December 2018.

[38] Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. A Survey and Classification of Software-Defined Storage Systems. *ACM Computing Surveys*, 53(3), 2020.

[39] Paul Menage. Linux Control Groups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[40] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: Quality-of-Service in Large Disk Arrays. In *8th ACM International Conference on Autonomic Computing*, page 245–254. ACM, 2011.

[41] Michael Mesnier, Feng Chen, Tian Luo, and Jason Akers. Differentiated Storage Services. In *23rd ACM Symposium on Operating Systems Principles*, pages 57–70. ACM, 2011.

[42] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. High Velocity Kernel File Systems with Bento. In *19th USENIX Conference on File and Storage Technologies*, pages 65–79. USENIX, 2021.

[43] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[44] Linux Man Page. proc - Process Information Pseudo-File Sytem, 1994.

[45] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. SafeFS: A Modular Architecture for Secure User-Space File Systems: One FUSE to Rule Them All. In *10th ACM International Systems and Storage Conference*. ACM, 2017.

[46] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, et al. A Configurable Rule Based Classful Token Bucket Filter Network Request Scheduler for the Lustre File System. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 6:1–6:12. ACM, 2017.

[47] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *26th ACM Symposium on Operating Systems Principles*, page 497–514. ACM, 2017.

[48] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3), 2015.

[49] Michael Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. Malacology: A Programmable Storage System. In *12th European Conference on Computer Systems*, page 175–190. ACM, 2017.

[50] David Shue and Michael Freedman. From Application Requests to Virtual IOPs: Provisioned Key-Value Storage with Libra. In *9th European Conference on Computer Systems*. ACM, 2014.

[51] David Shue, Michael Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation*, pages 349–362. USENIX, 2012.

[52] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. sRoute: Treating the Storage Stack Like a Network. In *14th USENIX Conference on File and Storage Technologies*, pages 197–212. USENIX, 2016.

[53] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *2020 ACM SIGMOD International Conference on Management of Data*, page 1493–1509. ACM, 2020.

[54] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *24th ACM Symposium on Operating Systems Principles*, pages 182–196. ACM, 2013.

[55] Raghav Tulshibagwale. RocksDB at Nutanix. https://www.nutanix.dev/2021/03/10/rocksdb-at-nutanix/.

[56] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies*, pages 59–72. USENIX, 2017.

[57] Matthew Wachs and Michael Abd-El-Malek. Argon: Performance Insulation for Shared Storage Servers. In *5th USENIX Conference on File and Storage Technologies*. USENIX, 2007.

[58] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling High-Level SLOs on Shared Storage Systems. In *3rd ACM Symposium on Cloud Computing*. ACM, 2012.

[59] Yin Wang and Arif Merchant. Proportional-Share Scheduling for Distributed Storage Systems. In *5th USENIX Conference on File and Storage Technologies*, pages 47–60. USENIX, 2007.

[60] Joel C. Wu and Scott A. Brandt. Providing Quality of Service Support in Object-Based File System. In *24th IEEE Conference on Mass Storage Systems and Technologies*, pages 157–170. IEEE, 2007.

[61] Miguel Xavier, Israel De Oliveira, Fabio Rossi, Robson Dos Passos, Kassiano Matteussi, and Cesar De Rose. A Performance Isolation Analysis of Disk-Intensive Workloads on Container-Based Clouds. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 253–260. IEEE, 2015.

[62] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini Kaushik, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Split-Level I/O Scheduling. In *25th ACM Symposium on Operating Systems Principles*, pages 474–489. ACM, 2015.

[63] Andy Yoo, Morris Jette, and Mark Grondona. Slurm: Simple Linux Utility for Resource Management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[64] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage Performance Virtualization via Throughput and Latency Control. *ACM Transactions on Storage*, 2(3):283–308, 2006.

[65] Timothy Zhu, Alexey Tumanov, Michael Kozuch, Mor Harchol-Balter, and Gregory Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *5th ACM Symposium on Cloud Computing*. ACM, 2014.

Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage

Qiuping Wang[1,2], Jinhong Li[1], Patrick P. C. Lee[1], Tao Ouyang[2], Chao Shi[2], Lilong Huang[2]
[1]*The Chinese University of Hong Kong* [2]*Alibaba Group*

Abstract

Log-structured storage has been widely deployed in various domains of storage systems, yet its garbage collection incurs write amplification (WA) due to the rewrites of live data. We show that there exists an optimal data placement scheme that minimizes WA using the future knowledge of block invalidation time (BIT) of each written block, yet it is infeasible to realize in practice. We propose a novel data placement algorithm for reducing WA, SepBIT, that aims to infer the BITs of written blocks from storage workloads and separately place the blocks into groups with similar estimated BITs. We show via both mathematical and production trace analyses that SepBIT effectively infers the BITs by leveraging the write skewness property in practical storage workloads. Trace analysis and prototype experiments show that SepBIT reduces WA and improves I/O throughput, respectively, compared with state-of-the-art data placement schemes. SepBIT is currently deployed to support the log-structured block storage management at Alibaba Cloud.

1 Introduction

Modern storage systems adopt the *log-structured* design [30] for high performance. Examples include flash-based solid-state drives (SSDs) [5,10], file systems [15,21,27,30,32], key-value stores [25,28], table stores [9], storage management [6], in-memory storage [31], RAID arrays [18], and cloud block services [40]. Log-structured storage transforms random write requests into sequential disk writes in an append-only log, so as to reduce disk seek overhead and improve write performance. It also brings various advantages in addition to high write performance, such as improved flash endurance in SSDs [21], unified abstraction for building distributed applications [6,9], efficient memory management in in-memory storage [31], and load balancing in cloud block storage [40]. Recent advances in zoned storage [4,7] also advocate the adoption of log-structured storage based on append-only interfaces for scalable performance.

The log-structured design writes live data blocks to the append-only log without modifying existing data blocks in-place, so it regularly performs *garbage collection (GC)* to reclaim the free space of stale blocks. GC works by reading a segment of blocks, removing any stale blocks, and writing back the remaining live blocks. The repeated writes of live blocks lead to *write amplification (WA)*. They not only incur I/O interference to foreground workloads [18], but also lead to reduced flash lifespans and unnecessary power consumption in data centers.

Mitigating WA in log-structured storage has been a well-studied topic in the literature (§5). In particular, a large body of studies focuses on designing *data placement* strategies by properly placing blocks in separate groups. He *et al.* [16] point out that a data placement scheme should group blocks by the *block invalidation time (BIT)* (i.e., the time when a block is invalidated by a live block; a.k.a. the death time [16]) to achieve the minimum WA. However, without obtaining the future knowledge of the BIT pattern, how to design an optimal data placement scheme with the minimum WA remains an unexplored issue. Existing temperature-based data placement schemes that group blocks by block temperatures (e.g., write/update frequencies) [12,20,27,33,35,42,43] are arguably inaccurate to capture the BIT pattern and fail to effectively group the blocks with similar BITs [16].

We propose SepBIT, a novel data placement scheme that aims for the minimum WA in log-structured storage. It infers the BITs of written blocks from the underlying storage workloads and separately places the written blocks into different groups, each of which stores the blocks with similar *estimated* BITs. Specifically, it builds on the *skewed* write patterns observed in the real-world cloud block storage workloads (e.g., Alibaba Cloud [23] and Tencent Cloud [46]). It separates the written blocks into *user-written blocks* and *GC-rewritten blocks* (defined in §2.1). It further separates each set of user-written blocks and GC-rewritten blocks by inferring the BIT of each block, so as to perform fine-grained separation of blocks into groups with similar estimated BITs. We summarize our contributions below:

- We first design an ideal data placement strategy that has the minimum WA in log-structured storage, based on the (impractical) assumption of having the future knowledge of BITs of written blocks. Our analysis not only motivates how to design a practical data placement scheme that aims to group the written blocks with similar BITs, but also provides an oracular baseline for our comparisons.
- We design SepBIT, which performs fine-grained separation of written blocks by inferring their BITs from the underlying storage workloads. We show via both mathematical and trace analyses that our BIT inference is effective in skewed workloads. SepBIT also achieves low memory overhead in its indexing structure for tracking block statistics.

- We evaluate SepBIT using real-world cloud block storage workloads at Alibaba Cloud [23] and Tencent Cloud [46]. Trace analysis on both workloads shows that SepBIT has the lowest WA compared with eight state-of-the-art data placement schemes. For example, for the Alibaba Cloud traces, SepBIT reduces the overall WA by 8.6-15.9% and 9.1-20.2% when the Greedy [30] and Cost-Benefit [30,31] algorithms are used for segment selection in GC, respectively. It also reduces the per-volume WA by up to 44.1%, compared with merely separating user-written and GC-rewritten blocks in data placement.
- We prototype a log-structured storage system that supports different data placement schemes and runs on an emulated zoned storage backend based on ZenFS [3]. Our prototype experiments show that SepBIT improves I/O throughput over most volumes due to its efficient WA reduction; for example, its median throughput is 20% higher than the second best data placement scheme.

SepBIT is currently deployed at Alibaba Cloud Enhanced SSDs (ESSDs) [1], which provide cloud block storage services for end-users or applications. Each ESSD is a block-level volume (or virtual disk) backed by flash-based SSD storage, and aims to support low-latency (e.g., around 100μs) and high-throughput (e.g., up to 1 M IOPS) I/O access. ESSDs are deployed atop Pangu [29], a general distributed storage platform that provides an append-only write interface. To be compatible with the append-only write interface of Pangu, ESSDs adopt the log-structured design and are abstracted as log-structured storage in our paper.

Our trace analysis scripts and prototype are open-sourced at `http://adslab.cse.cuhk.edu.hk/software/sepbit`.

2 Background and Motivation

2.1 GC in Log-Structured Storage

We consider a log-structured storage system that comprises multiple *volumes*, each of which is assigned to a user. Each volume is configured with a capacity of tens to hundreds of GiB and manages data in an append-only manner. It is further divided into *segments* that are configured with a maximum size (e.g., tens to hundreds of MiB). Each segment contains fixed-size *blocks*, each of which is identified by a *logical block address (LBA)* and has a size (e.g., several KiB) that aligns with the underlying disk drives. Each block, either from a new write or from an update to an existing block, is appended to a segment (called an *open* segment) that has not yet reached its maximum size. If a segment reaches its maximum size, the segment (called a *sealed* segment) becomes immutable. Updating an existing block is done in an *out-of-place* manner, in which the latest version of the block is appended to an open segment and becomes a *valid* block, while the old version of the block is invalidated and becomes an *invalid* block.

Log-structured storage needs to regularly reclaim the space occupied by the invalid blocks via GC. A variety of GC policies can be realized, yet we can abstract a GC policy as a three-phase procedure:

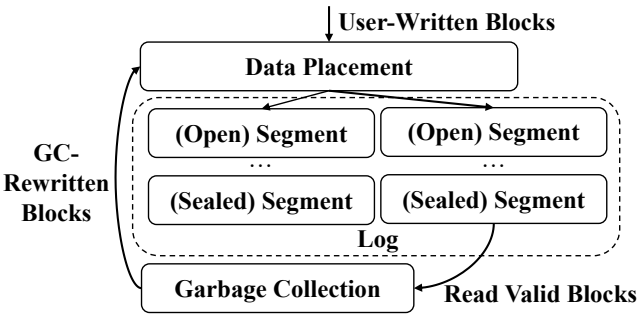


Figure 1: The workflow of a general data placement scheme.

- *Triggering*, which decides when a GC operation should be activated. In this work, we assume that a GC operation is triggered for a volume when its *garbage proportion (GP)* (i.e., the fraction of invalid blocks among all valid and invalid blocks) exceeds a pre-defined threshold (e.g., 15%).
- *Selection*, which selects one or multiple sealed segments for GC. In this work, we focus on two selection algorithms: (i) Greedy [30], which selects the sealed segments with the highest GPs, and (ii) Cost-Benefit [30,31], which selects the sealed segments that have the highest values $\frac{GP*age}{1-GP}$ (where *age* refers to the elapsed time of a sealed segment since it is sealed) for GC.
- *Rewriting*, which discards all invalid blocks from the selected sealed segments and writes back the remaining valid blocks into one or multiple open segments. The space of the selected sealed segments can then be reused.

A log-structured storage system sees two types of written blocks: each request that writes or updates an LBA in the workload generates one *user-written block* (i.e., a new block) and zero or more *GC-rewritten blocks* that are due to the rewrites of the block during GC. Thus, GC incurs *write amplification (WA)*, defined as the ratio of the total number of both user-written blocks and GC-rewritten blocks to the number of user-written blocks. In the deployment at Alibaba Cloud ESSDs (§1), we observe that the high WA from GC degrades both the effective I/O bandwidth and the SSD lifespans. It is thus critical to minimize WA.

In this work, we aim to design a general and lightweight data placement scheme that mitigates the WA due to GC in cloud-scale deployment. Figure 1 shows the workflow of a general data placement scheme, which separates all written blocks (i.e., user-written blocks and GC-rewritten blocks) into different groups and writes the blocks to the open segments of the respective groups. The data placement scheme is compatible with any GC policy (i.e., independent of the triggering, selection, and rewriting policies).

2.2 Ideal Data Placement

We present an ideal data placement scheme that minimizes WA (i.e., WA=1). We also elaborate why it is infeasible to

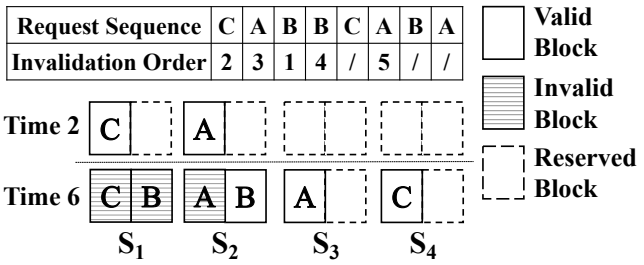


Figure 2: Example of the ideal data placement scheme.

realize in practice, so as to motivate the design of an effective practical data placement scheme.

System model. We first define the notations. Consider a write-only request sequence of blocks that are written to a log-structured storage system. Let m be the number of user-written blocks in the request sequence and s be the segment size (in units of blocks). Let $k = \lceil \frac{m}{s} \rceil$ be the number of sealed segments in the system, and let $S_1, S_2, \cdots, S_k$ denote the corresponding k sealed segments. Let o_i (where $o_i \geq 1$) be the *invalidation order* of the i-th block in the request sequence based on the BITs of all blocks (where $1 \leq i \leq m$), meaning that the i-th block is the o_i-th invalidated block among all invalid blocks.

Placement design. For the ideal placement scheme, we make the following assumptions. Suppose that the system has the future knowledge of the BITs of all blocks, and hence the invalidation order o_i of the i-th block in the request sequence (where $1 \leq i \leq m$). It also allocates k open segments for storing incoming blocks, and performs a GC operation whenever there are s invalid blocks in the system (i.e., one segment size of invalid blocks).

The system writes the i-th block to the $\lceil \frac{o_i}{s} \rceil$-th open segment. If the j-th (where $1 \leq j \leq k$) open segment is full, it is sealed into the sealed segment S_j. Thus, S_j stores the blocks with the invalidation orders in the range of $[(j-1) \cdot s+1, j \cdot s]$. The first GC operation is triggered when there exist s invalid blocks; according to the placement, all such blocks must be stored in S_1. Thus, the first GC operation will choose S_1 for GC, and there will be no rewrites as all blocks in S_1 must be invalid. In general, the j-th GC operation (where $1 \leq j \leq k$) will choose S_j for GC, and there will be no rewrites as S_j contains only invalid blocks.

Figure 2 depicts an example of the ideal data placement scheme. Consider a write-only request sequence with $m = 8$ blocks with three LBAs A, B, and C, and the i-th block is written at time i (where $1 \leq i \leq m$). We fix the segment size as $s = 2$. We show the status of the volume at time 2 and time 6 when the second block and the sixth block are written, respectively. At time 2, we have appended C to S_1 and A to S_2, as their invalidation orders are 2 and 3, respectively. Note that all blocks in S_1 become invalid when block C is updated at time 5, and at this time we can perform a GC operation to reclaim the free space occupied by S_1. Note that the GC operation does not incur any rewrite. Later, at time 6, the system appends A to S_3 since its invalidation order is 5.

Limitations and lessons learned. While the ideal data placement scheme achieves the minimum WA, there exist two practical limitations. First, the scheme needs to have future knowledge of the BIT of every block to assign the blocks to the corresponding open segments, but having such future knowledge is infeasible in practice. Second, the scheme needs to provision $k = \lceil m/s \rceil$ open segments to hold all m blocks in the request sequence in the worst case, as well as k corresponding sealed segments for keeping the blocks from the k open segments. Such provisioning incurring high memory and storage costs as m increases. Also, having too many open and sealed segments incurs substantial random writes that lead to performance slowdown.

A practical data placement scheme should address the above two limitations. Without the future knowledge of BITs, it should effectively *infer* the BIT of each written block. With only a limited number of available open segments, it should group written blocks by *similar BITs* instead of placing them in strict invalidation order. Our goal is to address the limitations driven by real-world cloud block storage workloads.

2.3 Trace Overview

We consider the *public* block-level I/O traces from two cloud block storage systems, Alibaba Cloud [23] and Tencent Cloud [46]. The Alibaba Cloud traces contain I/O requests (in multiples of 4 KiB blocks) from 1,000 virtual disks, referred to as *volumes*, over a one-month period in January 2020. The Tencent Cloud traces have 4,995 volumes over a nine-day period in October 2018. In this paper, we mainly focus on the Alibaba Cloud traces, while we verified that the Tencent Cloud traces show similar findings [39].

The Alibaba Cloud traces comprise a variety of workloads (e.g., virtual desktops, web services, key-value stores, and relational databases), and hence are representative to drive our analysis. We treat each volume in the traces as a standalone volume in the log-structured storage system (§2.1), such that each volume performs data placement and GC independently. Our goal is to mitigate the overall WA across all volumes.

We pre-process the traces for our analysis and evaluation as follows. We only consider write requests as they are the only contributors of WA. Since some volumes in the traces have limited write requests to trigger sufficient GC operations, we remove such volumes to avoid biasing our analysis. Specifically, we focus on the volumes with sufficient write requests: each volume has a write working set size (WSS) (i.e., the number of unique LBAs being written multiplied by the block size) above 10 GiB and a total write traffic size (i.e., the number of written bytes) above 2× its write WSS. To this end, we select 186 volumes from the Alibaba Cloud traces, which account for a total of over 90% of write traffic of all 1,000 volumes. The 186 volumes contain 10.9 billion write requests, 410.2 TiB of written data (with 390.2 TiB of

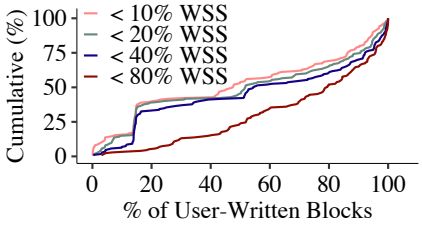


Figure 3: Percentages of user-written blocks with different short lifespans.

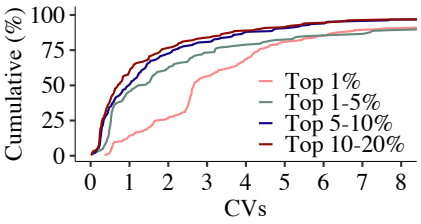


Figure 4: CVs of the lifespans of frequently updated blocks.

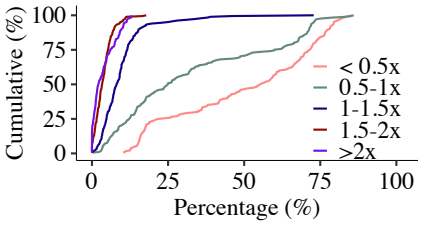


Figure 5: Percentages of rarely updated blocks with different lifespans.

updates), 20.3 TiB of write WSS (with 17.2 TiB of update WSS). Each of the 186 volumes has a write WSS ranging from 10 GiB to 1 TiB and a write traffic size ranging from 43 GiB to 36.2 TiB. Since the WSS varies across volumes, we configure the maximum storage space of each volume as $\frac{WSS}{1-GPT}$, where GPT denotes the GP threshold to trigger GC.

2.4 Motivation

We show via trace analysis that existing data placement schemes cannot accurately capture the BIT pattern and group the blocks with similar BITs for effective WA mitigation. We consider the 186 selected volumes from the Alibaba Cloud traces (§2.3). We define the lifespan of a block as the number of bytes written by the workload from when a block is written until it is invalidated (or until the end of the trace). A block is invalidated when the workload updates the same LBA. We make three key observations.

Observation 1: User-written blocks generally have short lifespans. We say that a block has a *short lifespan* if its lifespan is smaller than the write working set size (WSS) (i.e., the number of unique written LBAs multiplied by the 4 KiB block size). We examine the percentages of user-written blocks that fall into different lifespan range groups with short lifespans that are represented as the fractions of the write WSS for each volume. Figure 3 shows the cumulative distributions of the percentages of user-written blocks across all volumes in different lifespan groups. In a large fraction of volumes, their user-written blocks tend to have short lifespans. For example, half of the volumes have more than 79.5% of user-written blocks with lifespans smaller than 80% of their write WSSes, and have more than 47.6% of user-written blocks with lifespans smaller than only 10% write WSS. In contrast, GC-rewritten blocks generally have long lifespans. By definition, GC-rewritten blocks are rewritten as they remain valid in the GC-reclaimed segments. In both Greedy and Cost-Benefit selection algorithms, GC tends to select segments that either show a high GP or exist for a long time, implying that GC-rewritten blocks tend to have long lifespans.

Our findings suggest that user-written blocks and GC-rewritten blocks can have vastly different BIT patterns, in which user-written blocks tend to have short lifespans, while GC-rewritten blocks tend to have long lifespans. Existing data placement schemes either mix user writes and GC writes [12, 20, 27, 35], or focus on user writes [33, 42, 43], in the data placement decisions. Failing to distinguish between user-written blocks and GC-rewritten blocks can lead to inefficient WA mitigation. Instead, it is critical to separately identify the BIT patterns of user-written blocks and GC-rewritten blocks.

Observation 2: Frequently updated blocks have highly varying lifespans. We investigate *frequently updated blocks*, referred to as the blocks whose *update frequencies* (i.e., the number of updates) rank in the top 20% in the write working set (i.e., the set of LBAs being written) of a volume. Specifically, for each volume, we divide the frequently updated blocks into four groups based on their ranks of update frequencies, namely top 1%, top 1-5%, top 5-10%, and top 10-20%, so that the blocks in each group have similar update frequencies. The medians of the minimum update frequency in the four groups across all volumes are 37.5, 8.5, 6.0, and 5.0, respectively. To avoid evaluation bias, we exclude the blocks that have not been invalidated before the end of the traces. For each group of a volume, we calculate the *coefficient of variation (CV)* (i.e., the standard deviation divided by the mean) of the lifespans of the blocks; a high CV (e.g., larger than one) implies a high variance in the lifespans.

Figure 4 shows the cumulative distributions of CVs across all volumes (note that 6, 6, 20, and 18 volumes in the four groups have CVs exceeding 8, respectively). We see that frequently updated blocks with similar update frequencies have high variance in their lifespans (and hence the BITs); for example, 25% of the volumes have their CVs exceeding 4.34, 3.20, 2.14, and 1.82 in the four groups top 1%, top 1-5%, top 5-10%, and top 10-20%, respectively. Our findings also suggest that existing temperature-based data placement schemes that group the blocks with similar write/update frequencies [12, 20, 27, 33, 35, 42, 43] cannot effectively group blocks with similar BITs, and hence the WA cannot be fully mitigated.

Observation 3: Rarely updated blocks dominate and have highly varying lifespans. We examine the write working set of each volume and define the *rarely updated blocks* as those that are updated no more than four times during the one-month trace period. We see that rarely updated blocks occupy a high percentage in the write working sets of a large fraction of volumes. In half of the volumes, more than 72.4% of their write working sets contain rarely updated blocks. We further examine the lifespans of those rarely updated blocks. For each volume, we divide the rarely updated blocks into

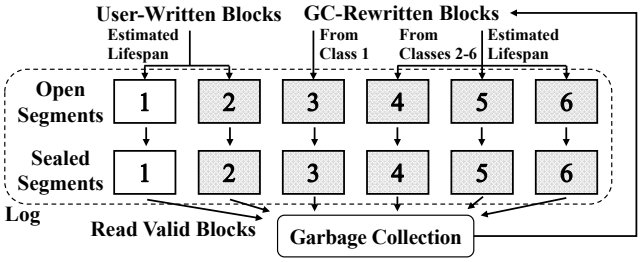


Figure 6: SepBIT workflow.

five groups that are partitioned by the lifespans of 0.5×, 1×, 1.5×, and 2× of their write WSSes. We then calculate the percentage of those blocks that fall into each group.

Figure 5 shows the cumulative distributions of the percentages of rarely updated blocks in different lifespan groups across all volumes. In 25% of the volumes, more than 71.5% of the rarely updated blocks have their lifespans smaller than 0.5× write WSS. For the remaining four groups, the medians of the percentages are 24.9%, 8.1%, 3.3%, and 2.2%, respectively. In other words, the lifespans of rarely updated blocks can span both short and long lifespan ranges, and hence show high deviations of BITs in a volume. As in Observation 2, our findings again suggest that existing temperature-based data placement schemes cannot effectively group the rarely updated blocks with similar BITs. Rarely updated blocks are often treated as cold blocks with low write frequencies, so they tend to be grouped together and separated from the hot blocks with high write frequencies. However, their vast differences in BIT patterns make temperature-based data placement schemes inefficient in mitigating WA.

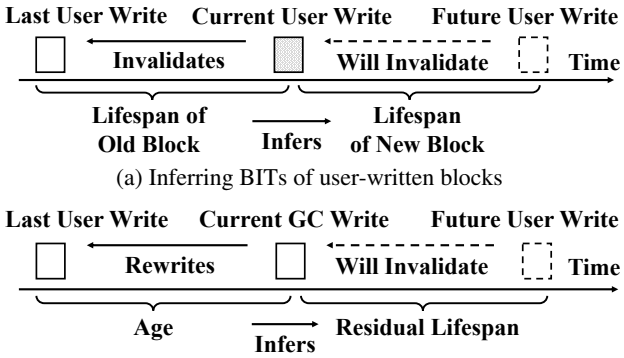
3 SepBIT Design

3.1 Design Overview

We design SepBIT based on our observations in §2.4. SepBIT first separates blocks into user-written blocks and GC-rewritten blocks due to their different BIT patterns (Observation 1). It further separates both user-written blocks and GC-rewritten blocks by inferring their BITs instead of using block temperatures as in existing temperature-based approaches (Observations 2 and 3).

Figure 6 depicts the workflow of SepBIT. Our current design of SepBIT defines *six* classes of segments, in which Classes 1-2 correspond to the segments of user-written blocks, while Classes 3-6 correspond to the segments of GC-rewritten blocks. Each class is now configured with one open segment and has multiple sealed segments. If an open segment reaches the maximum size, it is sealed and remains in the same class.

SepBIT infers the lifespans of blocks and in turn the corresponding BITs of blocks. For user-written blocks (i.e., Classes 1-2), SepBIT stores the *short-lived* blocks (with short lifespans) in Class 1 and the remaining *long-lived* blocks (with long lifespans) in Class 2. For GC-rewritten blocks (i.e., Classes 3-6), SepBIT appends the blocks from Class 1 that are rewritten by GC into Class 3, and groups the remaining



(a) Inferring BITs of user-written blocks

(b) Inferring BITs of GC-rewritten blocks

Figure 7: Ideas of inferring BITs in SepBIT.

GC-rewritten blocks into Classes 4-6 by similar BITs inferred.

The main idea of SepBIT is as follows. For each user-written block, SepBIT examines its *last user write time* to infer its lifespan. Specifically, for the write time, SepBIT uses a monotonic timer (instead of the real timestamp) that increments by one for each user-written block. If the user-written block is issued from a new write, SepBIT assumes that it has an infinite lifespan. Otherwise, if the user-written block updates an old block, SepBIT uses the lifespan of the old block (i.e., the number of user-written bytes in the whole workload since its last user write time until it is now invalidated) to estimate the lifespan of the user-written block, as shown in Figure 7(a). Our intuition is that *any user-written block that invalidates a short-lived block is also likely to be a short-lived block* (§3.2). Then if the short-lived blocks are written at about the same time, their corresponding BITs will be close, so SepBIT groups them into same class (i.e., Class 1). For the long-lived blocks (including the user-written blocks from new writes), SepBIT groups them into Class 2.

For each GC-rewritten block, SepBIT examines its *age*, defined as the number of user-written bytes in the whole workload since its last user write time until it is rewritten by GC, to infer its *residual lifespan*, defined as the number of user-written bytes since it is rewritten by GC until it is invalidated (or until the end of the traces), as shown in Figure 7(b). As a result, the lifespan of a GC-rewritten block is its age plus its residual lifespan. Our intuition is that *any GC-rewritten block with a smaller age has a higher probability to have a short residual lifespan* (§3.3), implying that GC-rewritten blocks with different ages are expected to have different residual lifespans. Thus, SepBIT can distinguish the blocks of different residual lifespans based on their ages and group the GC-rewritten blocks with similar ages into the same classes.

Our design builds on the assumption that the access pattern is *skewed* for inferring the BITs of blocks. We justify our assumption via the mathematical analysis for skewed distributions and the trace analysis for real-world workloads (§3.2 and §3.3). To adapt to changing workloads and GC policies, SepBIT monitors the workloads to separate user-written blocks and GC-rewritten blocks into different classes (§3.4).

3.2 Inferring BITs of User-Written Blocks

We show via both mathematical and trace analyses the effectiveness of SepBIT in estimating the BITs of user-written blocks based on the lifespans. Let n be the total number of unique LBAs in a working set; without loss of generality, each LBA is denoted by an integer from 1 to n. Let p_i (where $1 \le i \le n$) be the probability that LBA i is being written in each write request. Consider a write-only request sequence of blocks, each of which is associated with a sequence number b and the LBA A_b. Let b and b' (where $b' < b$) denote the sequence numbers of a new user-written block and the corresponding invalid old block, respectively (i.e., $A_b = A_{b'}$).

Recall from §3.1 that SepBIT estimates the lifespan (denoted by u) of the user-written block b using the lifespan (denoted by v) of the old block b', so the estimated BIT of block b is equal to the current user write time plus the estimated lifespan u; note that both u and v are expressed in units of blocks. We claim that if v is small, u is also likely to be small. To validate the claim, let u_0 and v_0 (both in units of blocks) be two thresholds. We then examine the conditional probability of $u \le u_0$ given the condition that $v \le v_0$ subject to a workload of different skewness. If the conditional probability is high for small u_0 and v_0, then our claim holds.

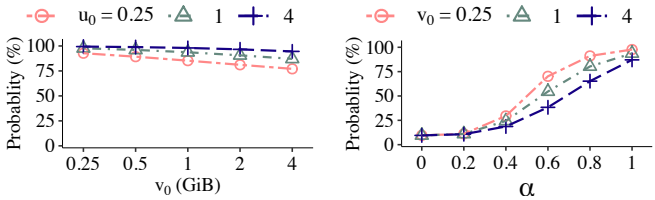
Mathematical analysis. We examine the following conditional probability (see derivation in our technical report [39]):

$$\Pr(u \le u_0 \mid v \le v_0) = \frac{\Pr(u \le u_0 \text{ and } v \le v_0)}{\Pr(v \le v_0)} = \frac{\sum_{i=1}^{n}(1-(1-p_i)^{u_0}) \cdot (1-(1-p_i)^{v_0}) \cdot p_i}{\sum_{i=1}^{n}(1-(1-p_i)^{v_0}) \cdot p_i}.$$

We analyze the conditional probability via the Zipf distribution, given by $p_i = (1/i^{\alpha})/\sum_{j=1}^{n}(1/j^{\alpha})$, where $1 \le i \le n$ for some skewness parameter $\alpha \ge 0$. A larger α implies a more skewed distribution. We fix $n = 10 \times 2^{18}$, which corresponds to a working set of 10 GiB with 4 KiB blocks. We then study how the conditional probability $\Pr(u \le u_0 \mid v \le v_0)$ varies across u_0, v_0, and α.

Figure 8(a) first shows the conditional probability for varying u_0 and v_0, where we fix $\alpha = 1$. We focus on short lifespans by varying u_0 and v_0 of up to 4 GiB, which is less than the write WSS (§2.4). Overall, the conditional probability is high for different u_0 and v_0; the lowest one is 77.1% for $v_0 = 4$ GiB and $u_0 = 0.25$ GiB. This shows that a user-written block is highly likely to have a short lifespan if its invalidated block also has a short lifespan. In particular, the conditional probability is higher if v_0 is smaller (i.e., the invalidated blocks have shorter lifespans), implying a more accurate estimation of the lifespan of the user-written block.

Figure 8(b) next shows the conditional probability for varying v_0 and α, where we fix $u_0 = 1$ GiB. Note that for $\alpha = 0$, the Zipf distribution reduces to a uniform distribution. Overall, the conditional probability increases with α (i.e., more skewed). For example, for $\alpha = 1$, the conditional probability



(a) $\alpha = 1$; varying u_0, v_0 (b) $u_0 = 1$ GiB; varying v_0, α

Figure 8: Inferring BITs of user-written blocks: $\Pr(u \le u_0 \mid v \le v_0)$ versus v_0 and α.

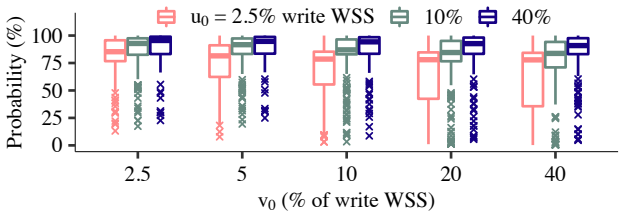


Figure 9: Inferring BITs of user-written blocks: Boxplots of $\Pr(u \le u_0 \mid v \le v_0)$ for different u_0 and v_0 in real-world workloads.

is at least 87.1%. However, for $\alpha = 0$, the conditional probability is only 9.5%. This indicates that the high accuracy of lifespan estimation only holds under skewed workloads.

Trace analysis. We use the block-level I/O traces from Alibaba Cloud (§2.3) to validate if the conditional probability remains high in real-world workloads. To compute the conditional probability, we first find the set of user-written blocks that invalidate old blocks with $v \le v_0$. Then the conditional probability is the fraction of blocks with $u \le u_0$ in the set. We vary both v_0 and u_0 as different percentages of the write WSS to examine different conditional probabilities. Figure 9 shows the boxplots of the conditional probabilities over all volumes for different u_0 and v_0. In general, the conditional probability remains high in most of the volumes. For example, for v_0 being 40% of write WSS, the medians of the conditional probabilities are in the range of 77.8-90.9%, and the 75th percentiles are in the range of 84.3-97.6%. Also, the conditional probability tends to be higher for a smaller v_0.

3.3 Inferring BITs of GC-Rewritten Blocks

We further show via both mathematical and trace analyses the effectiveness of SepBIT in estimating the BITs of GC-rewritten blocks based on the *residual* lifespans. Recall from §3.1 that SepBIT estimates the residual lifespan of a GC-rewritten block using its age, so the estimated BIT of the GC-rewritten block is equal to the current GC write time plus the estimated residual lifespan. However, characterizing directly GC-rewritten blocks is non-trivial, as it depends on the actual GC policy (e.g., when GC is triggered and which segments are selected for GC) (§2.1). Instead, we model GC-rewritten blocks based on user-written blocks. If a user-written block has a lifespan above a certain threshold, we assume that it is rewritten by GC and treat it as a GC-rewritten block with an age equal to the threshold. We can then apply a similar

analysis for user-written blocks as in §3.2.

We define the following notations. As each GC-rewritten block is a user-written block before being rewritten by GC, we identify each GC-rewritten block by its corresponding user-written block with sequence number b. Let u, g, and r be its lifespan, age, and residual lifespan, respectively, such that $u = g + r$; each of the variables is measured in units of blocks. We claim that r has a higher probability to be small with a smaller g. To validate the claim, let g_0 and r_0 (both in units of blocks) be the thresholds for the age and the residual lifespan, respectively. We examine the conditional probability of $u \leq g_0 + r_0$ given the condition that $u \geq g = g_0$ subject to a workload of different skewness. The conditional probability specifies the fraction of GC-rewritten blocks whose residual lifespans are shorter than r_0 among all GC-rewritten blocks with age g_0 (note that the GC-rewritten blocks are modeled as user-written blocks with lifespans above g_0). If the conditional probability is higher for a smaller g_0 subject to a fixed r_0, then our claim holds.

Mathematical analysis. We examine the following conditional probability (see derivation in our technical report [39]):

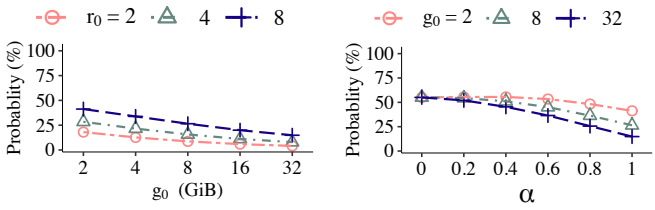
$$\Pr(u \leq g_0 + r_0 \mid u \geq g_0) = \frac{\Pr(g_0 \leq u \leq g_0 + r_0)}{\Pr(u \geq g_0)} = \frac{\sum_{i=1}^{n} p_i \cdot ((1-p_i)^{g_0} - (1-p_i)^{g_0+r_0})}{\sum_{i=1}^{n} p_i \cdot (1-p_i)^{g_0}}.$$

As in §3.2, we use the Zipf distribution and fix $n = 10 \times 2^{18}$ unique LBAs. We study how the conditional probability $\Pr(u \leq g_0 + r_0 \mid u \geq g_0)$ varies across g_0, r_0, and α.

Figure 10(a) first shows the conditional probability for varying g_0 and r_0, where we fix $\alpha = 1$. We focus on a large value of g_0 of up to 32 GiB since we target long-lived blocks. We also vary r_0 up to 8 GiB. Overall, for a fixed r_0, the conditional probability decreases as g_0 increases. For example, given that $r_0 = 8$ GiB, the probability with $g_0 = 2$ GiB is 41.2%, while the probability for $g_0 = 32$ GiB drops to 14.9%. This validates our claim that GC-rewritten blocks with different ages are expected to have different residual lifespans. Thus, we can distinguish the GC-rewritten blocks of different residual lifespans based on their ages.

Figure 10(b) further shows the conditional probability for varying g_0 and α, where we fix $r_0 = 8$ GiB. For a small α, the conditional probability has a limited difference for varying v, while the difference becomes more significant as α increases. For example, for $\alpha = 0$ (i.e., the uniform distribution), there is no difference varying g_0; for $\alpha = 0.2$, the difference of the conditional probability between $g_0 = 2$ GiB and $g_0 = 32$ GiB is only 3.5%, while the difference for $\alpha = 1$ is 26.4%. This indicates that our claim holds under skewed workloads, and we can better distinguish the GC-rewritten blocks of different residual lifespans under more skewed workloads.

Trace analysis. We also use block-level I/O traces from Alibaba Cloud (§2.3) to examine the conditional probability in



(a) $\alpha = 1$; varying r_0, g_0 (b) $r_0 = 8$ GiB; varying g_0, α

Figure 10: Inferring BITs of GC-rewritten blocks: $\Pr(u \leq g_0 + r_0 \mid u \geq g_0)$ versus g_0 and α.

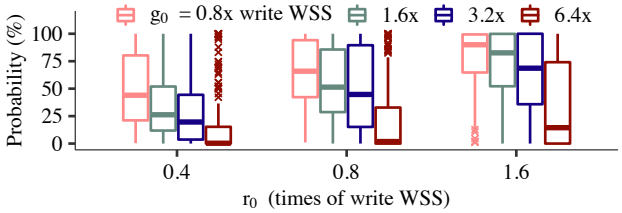


Figure 11: Inferring BITs of GC-rewritten blocks: Boxplots of $\Pr(u \leq g_0 + r_0 \mid u \geq g_0)$ for different r_0 and g_0 in real-world workloads.

real-world workloads. We first identify the set of blocks with $u \geq g_0$ in the workload, and then compute the conditional probability as a fraction of blocks with $u \leq g_0 + r_0$ in the set. We vary both r_0 and g_0 as different percentages of the write WSS. Figure 11 depicts the boxplots of the conditional probabilities over all volumes for different g_0 and r_0. For a fixed r_0, the conditional probabilities have significant differences for varying g_0. For example, if we fix r_0 as 1.6× of write WSS and g_0 increases from 0.8× to 6.4× of write WSS, the median probabilities drop from 90.0% to 14.5%.

3.4 Implementation Details

Threshold selection. We assign blocks into different classes by their estimated BITs with multiple thresholds: for user-written blocks, we define a *lifespan threshold* for separating short-lived blocks and long-lived blocks; for GC-rewritten blocks, we need multiple *age thresholds* to separate them by ages (§3.1). We configure the thresholds via the *segment lifespan* of a segment, defined as the number of user-written bytes in the workload since the segment is created (i.e., the time when the first block is appended to the segment) until it is reclaimed by GC. Specifically, we monitor the average segment lifespan, denoted by ℓ, among a fixed number of recently reclaimed segments in Class 1. For each user-written block, if it invalidates an old block with a lifespan less than ℓ, we write it to Class 1; otherwise, we write it to Class 2. For GC-rewritten blocks, we set the age thresholds as multiples of ℓ (see below).

Algorithmic details. Algorithm 1 shows the pseudo-code of SepBIT, which consists of three functions: `GarbageCollect`, `UserWrite`, and `GCWrite`. Each class always corresponds to one open segment. If an open segment is full, it becomes a sealed segment, and SepBIT creates a new open segment

Algorithm 1 SepBIT

```
1: t = 0; ℓ = +∞; ℓ_tot = 0; n_c = 0, where t is the global timestamp
2: function GarbageCollect( )
3:     Select a segment S by selection algorithm
4:     if S is from Class 1 then
5:         n_c = n_c + 1, ℓ_tot = ℓ_tot + (t − S.creation_time)
6:     end if
7:     if n_c = 16 then
8:         ℓ = ℓ_tot/n_c; n_c = 0; ℓ_tot = 0
9:     end if
10:    for each valid block b in S do
11:        GCWrite(b)
12:    end for
13: end function
14: function UserWrite(b)
15:    Find lifespan v of the invalidated block b′ due to b
16:    if v < ℓ then
17:        Append b to open segment of Class 1
18:    else
19:        Append b to open segment of Class 2
20:    end if
21:    t = t + 1
22: end function
23: function GCWrite(b)
24:    if b is from Class 1 then
25:        Append b to open segment of Class 3
26:    else
27:        g = t − b.last_user_write_time
28:        If g ∈ [0, 4ℓ), append b to open segment of Class 4
29:        If g ∈ [4ℓ, 16ℓ), append b to open segment of Class 5
30:        If g ∈ [16ℓ, +∞), append b to open segment of Class 6
31:    end if
32: end function
```

within the same class. SepBIT initializes the average segment lifespan $\ell = +\infty$, which is updated on-the-fly. It also tracks a global timestamp t, which records the sequence number of the current user-written block.

`GarbageCollect` is triggered by a GC operation according to the GC policy (§2.1). It performs GC and monitors the runtime information of the reclaimed segments. It selects a segment S for GC based on the selection algorithm (e.g., Greedy or Cost-Benefit (§2.1)). It sums up the lifespans of collected segments from Class 1 as ℓ_{tot}, and computes the average lifespan $\ell = \ell_{tot}/n_c$ for every fixed number n_c (e.g., $n_c = 16$ in our current implementation) of reclaimed segments.

`UserWrite` processes each user-written block b. It first computes the lifespan v of the invalidated old block b'. If v is less than ℓ, `UserWrite` appends b (which is treated as a short-lived block) to the open segment of Class 1; otherwise, it appends b (which is treated as a long-lived block) to the open segment of Class 2.

`GCWrite` processes each GC-rewritten block that corresponds to some user-written block b. If b is originally stored in Class 1, `GCWrite` appends b to the open segment of Class 3; otherwise, `GCWrite` appends b to one of the open segments of Classes 4-6 based on the age of b. Currently, we configure the age thresholds as three ranges, $[0, 4\ell)$, $[4\ell, 16\ell)$, and $[16\ell, +\infty)$, for Classes 4-6, respectively, based on our evaluation findings. Nevertheless, we have also experimented with different numbers of classes and thresholds [39], and we observe only marginal differences in WA.

Memory usage. SepBIT only stores the last user write time of each block as the metadata alongside the block *on disk*, without maintaining a mapping from every LBA to its last user write time in memory. Putting metadata alongside a block is feasible, as SSDs typically associate a small spare region (e.g., of size 64 bytes) with each flash page for storing metadata. Specifically, for user-written blocks, SepBIT only needs to know whether the lifespan of an invalidated block is shorter than a threshold. It thus suffices for SepBIT to track only the recently written LBAs. In our current implementation (written in C++), SepBIT maintains a first-in-first-out (FIFO) queue to record recently written LBAs. It dynamically adjusts the queue length according to the value ℓ. If the FIFO queue is full, each insert of an element will dequeue one element from the queue. If ℓ increases, the FIFO queue allows more inserts without dequeueing any element; if ℓ decreases, the FIFO queue dequeues two elements for each insert until the number of elements drops below ℓ. If the LBA exists in the FIFO queue and its user write time is within the recent ℓ user writes, SepBIT writes it into Class 1. To efficiently query the FIFO queue, SepBIT creates a `std::map` structure in the C++ standard template library to record each unique LBA in the FIFO queue and its latest queue position. When we enqueue the LBA of a newly written block into the FIFO queue, we insert or update the LBA with its current queue position in the `std::map` structure; when we dequeue an LBA from the FIFO queue, we remove the LBA from the `std::map` structure if its recorded queue position is equal to the dequeued one.

For GC-rewritten blocks, SepBIT retrieves them during GC and examines the user write time directly from the metadata, so as to assign the GC-rewritten block to the corresponding class without any memory overhead incurred.

Prototype. We prototype a log-structured block storage system that realizes SepBIT and existing data placement schemes. We choose to deploy our prototype on zoned storage [4], whose append-only interfaces favor log-structured storage deployment. Specifically, our prototype runs on an emulated zoned storage backend based on ZenFS [3] (due to the lack of a real zoned storage device, we currently emulate the zoned storage backend using Intel Optane Persistent Memory [2]). Each segment in the prototype is a one-to-one mapping to a *ZoneFile*, the basic unit in the zoned storage backend in ZenFS. Then ZenFS stores ZoneFiles in different zones without incurring device-level GC. For the metadata and the FIFO queue in SepBIT, the prototype stores them in separate files and accesses them using `mmap` for memory efficiency; for other existing data placement schemes, the prototype stores

all metadata in memory. When the prototype triggers GC (at the system level), it reads only valid blocks from storage and rewrites the blocks into different segments.

The reasons of choosing emulated zoned storage based on ZenFS in our prototype are three-fold. First, zoned storage has a similar storage abstraction to Pangu (§1), as both of them support append-only writes and large-size append-only units (e.g., up to hundreds of MiB). Second, emulated zoned storage provides minimal external interference, making the performance evaluation reproducible; in contrast, the performance of traditional SSDs can be easily disturbed by device-level GC. Finally, ZenFS is a lightweight user-space zone-aware file system that readily supports zoned storage.

4 Evaluation

4.1 Data Placement Schemes

We compare SepBIT with eight existing temperature-based data placement schemes, namely Dynamic dAta Clustering (DAC) [12], SFS [27], MultiLog (ML) [35], extent-based identification (ETI) [33], MultiQueue (MQ) [42], Sequentiality, Frequency, and Recency (SFR) [42], Fading Average Data Classifier (FADaC) [20], and WARCIP [43]. Note that these existing schemes are mainly designed for mitigating the flash-level WA in SSDs, yet they are also applicable for general log-structured storage. Take DAC [12] as an example. DAC associates each LBA with a temperature-based counter (quantified based on the write count) and writes blocks to the segments of different temperature levels. Each user write promotes the LBA to a hotter segment while each GC write demotes the LBA to a colder segment. Other temperature-based data placement schemes follow the similar idea of DAC. Specifically, the above designs adopt different metrics to measure block temperatures, such as access frequencies (in ML [35], MQ [42], and ETI [33]), recency (in FADaC [20]), hotness (in SFS [27]), access counts (in DAC [12]), sequentiality (in SFR [42]), and update intervals (in WARCIP [43]).

We also consider three baseline strategies.

- **NoSep** appends any written blocks (either user-written blocks or GC-rewritten blocks) to the same open segment.
- **SepGC [37]** separates written blocks by user-written blocks and GC-rewritten blocks, and writes them into two different open segments.
- **Future knowledge (FK)** assumes that the BIT of each written block is known in advance. For a written block (either a user-written block or a GC-rewritten block), if its invalidation will occur within t bytes since the written time, we write the block to the $\lceil \frac{t}{s} \rceil$-th open segment, where s is the segment size (in bytes). Given the limited number of open segments, FK uses the last open segment to store all user-written blocks and GC-rewritten blocks if their BITs do not belong to the prior open segments. We annotate the lifespan of each block in the traces in advance, so that we can compute the BITs during evaluation.

Note that FK represents an *oracular* baseline that leverages future knowledge for placement decisions. It is identical to the ideal scheme (§2.2) if there are unlimited memory and storage budgets. Otherwise, with limited memory and storage budgets, it applies future knowledge to group a subset of blocks in a limited number of segments, and applies trivial data placement for the remaining blocks. Thus, FK represents both the ideal data placement scheme that has no memory and storage constraints and the trivial data placement scheme with the memory and storage constraints; the latter serves the baseline in our experiments.

By default, we configure six classes (each containing one open segment) for data placement for all schemes, except for NoSep, SepGC, and ETI. For NoSep, we configure one class for all written blocks; for SepGC, we configure two classes, one for user-written blocks and one for GC-rewritten blocks; for ETI, we configure two classes for user-written blocks and one class for GC-rewritten blocks. For MQ, SFR, and WARCIP, as they focus on separating user-written blocks only, we configure five classes for user-written blocks and the remaining class for GC-rewritten blocks. For DAC, SFS, ML, FADaC, and FK, since they do not differentiate user-written blocks and GC-rewritten blocks, we let them use all six classes for all written blocks. We adopt the default settings as described in the original papers of the existing schemes.

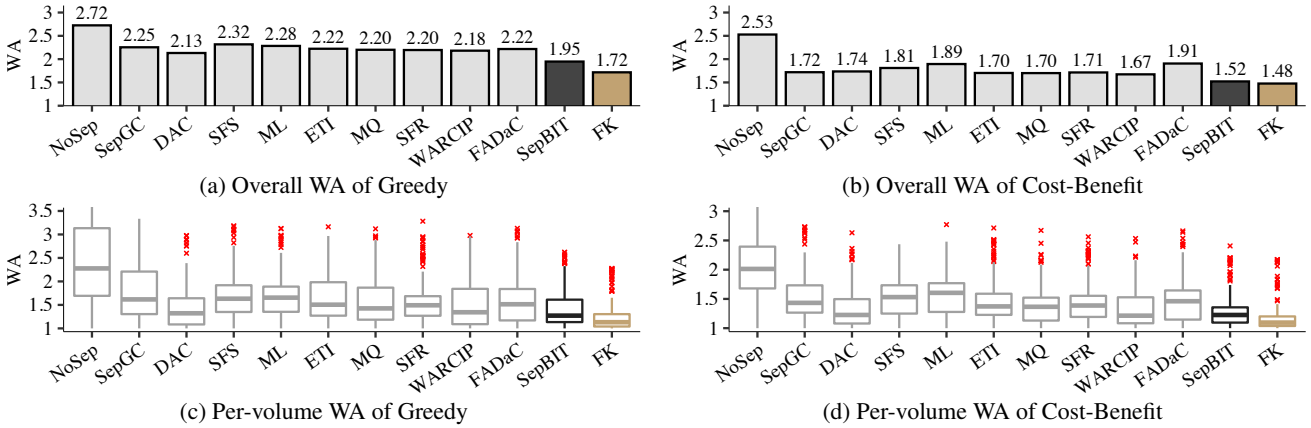
4.2 Results

Summary of findings. Our major findings include:

- SepBIT achieves the lowest WA among all data placement schemes (except FK) for different segment selection algorithms (Exp#1), different segment sizes (Exp#2), and different GP thresholds (Exp#3).
- We show that SepBIT provides accurate BIT inference (Exp#4).
- We provide a breakdown analysis on SepBIT, and show that it achieves a low WA by separating each set of user-written blocks and GC-rewritten blocks independently (Exp#5).
- SepBIT achieves the lowest WA in the Tencent Cloud traces (Exp#6).
- SepBIT shows high WA reduction for highly skewed workloads (Exp#7).
- We provide a memory overhead analysis and show that SepBIT achieves low memory overhead for a majority of the volumes (Exp#8).
- Our prototype evaluation shows that SepBIT achieves the highest throughput in a majority of the volumes (Exp#9).

Default configuration. Our default GC policy uses Cost-Benefit [30, 31] for segment selection and fixes the segment size and the GP threshold for triggering GC as 512 MiB and 15%, respectively; in Exp#1-Exp#3, we vary each of the configurations for evaluation. For real-world workloads, we use the Alibaba Cloud traces except for Exp#5.

Exp#1 (Impact of segment selection). We compare SepBIT with existing data placement schemes using Greedy [30] and



(a) Overall WA of Greedy

(b) Overall WA of Cost-Benefit

(c) Per-volume WA of Greedy

(d) Per-volume WA of Cost-Benefit

Figure 12: Exp#1 (Impact of segment selection).

Cost-Benefit [30, 31] for segment selection in GC (§2.1).

Figures 12(a) and 12(b) depict the overall WA across all 186 volumes under Greedy and Cost-Benefit, respectively. With separation in data placement, SepBIT reduces the overall WA of NoSep by 28.5% and 39.8% under Greedy and Cost-Benefit, respectively. More importantly, SepBIT achieves the lowest WA compared with all existing data placement schemes (except FK). It reduces the overall WA of SepGC and the eight state-of-the-art data placement schemes (i.e., excluding NoSep and FK) by 8.6-15.9% and 9.1-20.2% under Greedy and Cost-Benefit, respectively. Compared with FK, the overall WA of SepBIT is 13.5% and 3.1% higher under Greedy and Cost-Benefit, respectively. In short, SepBIT is highly efficient in WA mitigation under real-world workloads. Note that some data placement schemes even show a higher WA than SepGC, which performs simple separation of user-written blocks and GC-rewritten blocks, mainly because they fail to effectively group blocks with similar BITs (§2.4).

Figures 12(c) and 12(d) show the boxplots of per-volume WAs over all 186 volumes under Greedy and Cost-Benefit, respectively (we omit outliers of NoSep with very high WAs). SepBIT has the lowest 75th percentiles (1.61 and 1.36) among all existing data placement schemes (except FK) under Greedy and Cost-Benefit, while the second lowest one is DAC (1.64 and 1.50), respectively. This shows that SepBIT effectively reduces WAs in individual volumes with diverse workloads. In particular, Cost-Benefit is more effective in the WA reduction of SepBIT than Greedy, as the gap of the 75th percentiles between SepBIT and the second lowest one increases from 1.8% in Greedy to 9.4% in Cost-Benefit. Compared with FK, for 75th percentiles, SepBIT has 23.6% and 12.9% higher WA under Greedy and Cost-Benefit, respectively.

Exp#2 (Impact of segment sizes). We vary the segment size from 64 MiB to 512 MiB. For fair comparisons, we fix the amount of data (both valid and invalid data) to be retrieved in each GC operation as 512 MiB, meaning that a GC operation collects eight, four, two, and one segment(s) for the segment sizes of 64 MiB, 128 MiB, 256 MiB, and 512 MiB, respectively. We focus on comparing NoSep, SepGC, WARCIP, SepBIT, and FK, as they show the lowest WAs among existing data placement for various segment sizes. We present the complete results in our technical report [39].

Figures 13 depicts the overall WA versus the segment size. Overall, using a smaller segment size yields a lower WA, as a GC operation can perform more fine-grained selection of segments for more efficient space reclamation. Again, SepBIT achieves the lowest WA compared with all existing data placement schemes; for example, its WAs are 5.5%, 8.2%, and 10.0% lower than WARCIP for the segment sizes of 64 MiB, 128 MiB, and 256 MiB, respectively. Interestingly, SepBIT even has a lower WA (by 3.9-5.7%) than FK when the segment size is in the range of 64 MiB to 256 MiB. The reason is that FK currently groups blocks of close BITs in five open segments, while the last open segment stores all blocks (we now configure six classes in total) (§4.1). If the segment size is smaller, FK can only group fewer blocks in the limited number of open segments, so it becomes less effective of grouping blocks of close BITs.

Exp#3 (Impact of GP thresholds). We vary the GP thresholds from 10% to 25%. We again focus on comparing the overall WAs of NoSep, SepGC, WARCIP, SepBIT, and FK as in Exp#2. Figure 14 shows the overall WA versus the GP threshold. A larger GP threshold has a lower WA in general, as it is easier for a GC operation to select segments with high GPs. SepBIT still shows the lowest WA. It has 5.0-13.8% lower WAs than WARCIP for different GP thresholds. Compared with FK, SepBIT has comparable WAs with differences smaller than 1.8%, for different GP thresholds.

Exp#4 (BIT inference analysis). We study the effectiveness of the BIT inference in SepBIT. Note that SepBIT does not explicitly compute the estimated BIT of a block, but instead assigns blocks into classes corresponding to different ranges of estimated BITs (§3.4). To examine the effectiveness of BIT inference, our intuition is that each valid block that is rewritten during GC indicates that we incorrectly infer its BIT and places it into an incorrect segment. Thus, we can examine the GP of each collected segment to estimate the inference

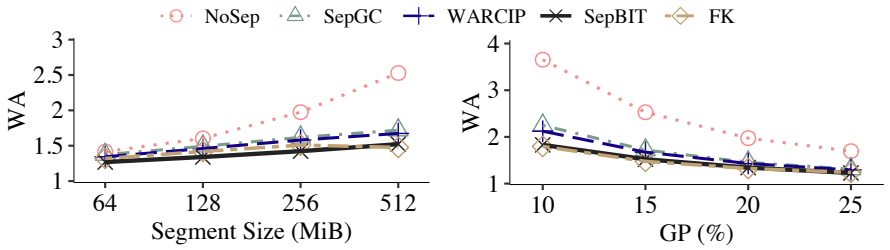


Figure 13: Exp#2 (Impact of segment sizes).

Figure 14: Exp#3 (Impact of GP thresholds).

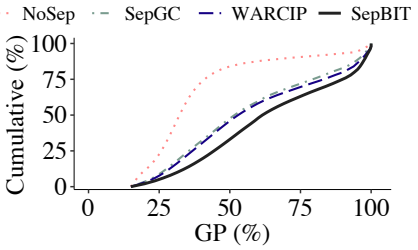
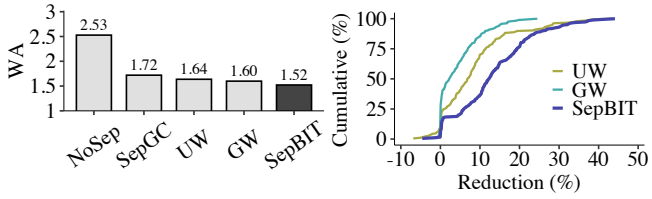


Figure 15: Exp#4 (BIT inference analysis).



(a) Overall WA (b) Per-volume WA reduction

Figure 16: Exp#5 (Breakdown analysis).

accuracy, such that a higher GP implies more accurate inference. We use the Cost-Benefit selection algorithm and fix the segment size and GP for triggering GC as 512 MiB and 15%, respectively. We study NoSep, SepGC, WARCIP, and SepBIT (WARCIP has the second lowest WA). We aggregate the collected segments during GC for all 186 volumes.

Figure 15 depicts the cumulative distributions of collected segments across different GPs for different schemes. The median GPs of the collected segments for NoSep, SepGC, WARCIP, and SepBIT are 32.3%, 51.6%, 52.9%, and 61.5%, respectively. SepBIT has the highest GPs, implying that it also has the highest accuracy in inferring BITs. WARCIP only shows a slightly higher GP of the collected segments than SepGC, so its WA reduction over SepGC is marginal.

Exp#5 (Breakdown analysis). We analyze how different components of SepBIT contribute to WA reduction. Recall that SepBIT separates written blocks into the user-written blocks and GC-rewritten blocks, and further separates each set of user-written blocks and GC-rewritten blocks independently. In our analysis, we consider NoSep (i.e., without separation), SepGC (i.e., separating written blocks into the user-written blocks and GC-rewritten blocks), and two variants:

- *UW*: It further separates user-written blocks based on SepGC, but without separating GC-rewritten blocks. It maintains three classes: Classes 1 and 2 store short-lived blocks and long-lived blocks as in SepBIT, respectively, while Class 3 stores all GC-rewritten blocks.
- *GW*: It further separates GC-rewritten blocks based on SepGC, but without separating user-written blocks. It maintains four classes: Class 1 stores all user-written blocks, and Classes 2-4 store GC-rewritten blocks as in Classes 4-6 of SepBIT.

Figure 16(a) shows the overall WAs of different data placement schemes. UW and GW reduce WA by 35.2% and 36.7% compared with NoSep, respectively; they also reduce WA by 4.8% and 7.0% compared with SepGC, respectively. The findings show that more fine-grained separation of each set of user-written blocks and GC-rewritten blocks brings further WA reduction. Also, SepBIT reduces WA by 7.0% and 4.9% compared with UW and GW, respectively, meaning that SepBIT can combine the benefits of UW and GW.

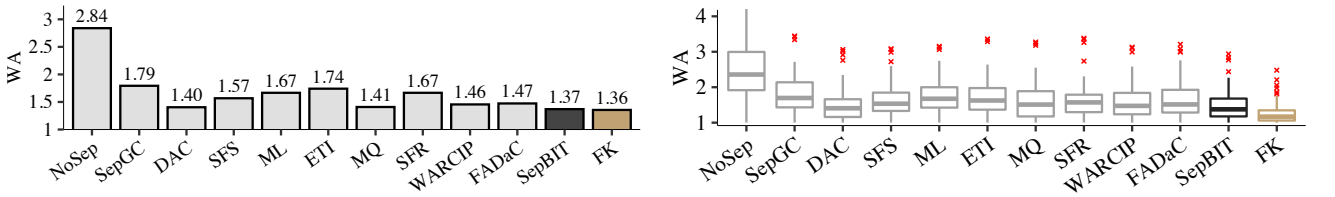
Figure 16(b) further shows the cumulative distributions of the WA reductions of UW, GW, and SepBIT compared with SepGC across all volumes. UW, GW, and SepBIT can reduce the WA of most of the volumes. The 75th percentiles of reductions of UW and GW are 11.4% and 6.9%, respectively, and their highest WA reductions are 43.3% and 24.5%, respectively. By combining UW and GW, the 75th percentile of the WA reductions of SepBIT compared with SepGC improves to 19.3% with the highest WA reduction as 44.1%.

Exp#6 (Results on the Tencent Cloud traces). We validate the effectiveness of SepBIT on the Tencent Cloud traces [46]. We pre-process the traces the same as for the Alibaba Cloud traces (§2.3) and select 271 out of 4,995 volumes. We run all the schemes as in Exp#1, using Cost-Benefit for segment selection and fixing the segment size and the GP threshold as 512 MiB and 15%, respectively.

Figure 17 depicts the overall WA and the per-volume WA across all 271 volumes. Among all existing data placement schemes, SepBIT achieves the lowest overall WA. Its overall WA is 2.5-21.3% lower than those of the eight existing schemes and 1.1% higher than that of FK. Compared with the second lowest scheme DAC, SepBIT has similar 50th and 75th percentiles of per-volume WA, and reduces the 90th percentile of per-volume WA from 2.09 to 1.97.

Exp#7 (Impact of workload skewness). We study how SepBIT works in workloads of different skewness. We set the selection algorithm as Greedy instead of Cost-Benefit, since Cost-Benefit also leverages the workload skewness to reduce WA and we want to exclude its impact from our analysis.

We inspect the skewness of each volume in the Alibaba Cloud traces, and analyze the correlation between the per-volume skewness and the WA reduction percentage of SepBIT over NoSep. We also present the results for synthetic workloads in our technical report [39]. Since not all real-world workloads have good fitness to a Zipf distribution [45], we describe the per-volume skewness according to how write traffic aggregates in the most frequently updated blocks. Specifically,



(a) Overall WA

(b) Per-volume WA

Figure 17: Exp#6 (Results on the Tencent Cloud traces).

Skewness α	0	0.2	0.4	0.6	0.8	1
Pct. (%)	20	27.6	38.1	52.4	71.1	89.5

Table 1: The percentage of write traffic over top-20% blocks in Zipf workloads of different skewness.

we compute the percentage of aggregated write traffic over the top 20% frequently written blocks. To show the relationship between the percentage of aggregated writes and the skewness factor of the Zipf distribution, Table 1 shows the percentage of write traffic over the top 20% frequently written blocks and the corresponding skewness factor α; note that the numbers are generated using 10 GiB of write WSS.

Figure 18 shows the results. Each point represents one volume. The x-axis is the percentage of aggregated write traffic over top 20% frequently written blocks and the y-axis is the WA reduction of SepBIT over NoSep. We see a positive correlation between the percentage of aggregated write traffic and the WA reduction (we also find that the p-value is smaller than 0.01 for the Pearson correlation coefficient 0.75, meaning that the positive correlation is statistically significant). For the volumes with percentages of aggregated write traffic larger than 80%, SepBIT reduces the WA by at least 38.0% with a maximum reduction of 76.7%.

Exp#8 (Memory overhead analysis). We analyze the memory overhead of SepBIT using the Alibaba Cloud traces. Recall that SepBIT tracks only the unique LBAs inside the FIFO queue (§3.4), instead of maintaining the mappings for all LBAs in the write working set. We report the memory overhead reduction of SepBIT as one minus the ratio of the number of unique LBAs in the FIFO queue to the number of unique LBAs in the write working set. To quantify the reduction, for each volume, we collect all values of the number of unique LBAs in the FIFO queue at runtime when ℓ (§3.4) is updated. To avoid bias due to the cold start of trace replay, for each volume, we exclude the beginning 10% of the values. We also collect the number of unique LBAs at the end of the traces. We consider two cases, namely (i) the worst case and (ii) the snapshot case. In the worst case, we use the maximum number of unique LBAs in the FIFO queue for all volumes; it assumes that each volume has its peak number of unique LBAs in the FIFO queue and incurs the most memory. In the snapshot case, we use the number of unique LBAs at the end of the traces, representing a snapshot of the system status.

From our analysis, we find that in the worst case, SepBIT reduces the overall memory overhead by 44.8%, while in the snapshot case, SepBIT reduces the overall memory overhead by 71.8%. To calculate the actual memory overhead, suppose that the mapping for each LBA has 8 bytes, in which both the LBA and the FIFO position are of size 4 bytes each (a 4-byte LBA can represent an address space of $2^{32} \times 2^{12} =$ 16 TiB for 4-KiB blocks). Since the aggregated write WSS of the 186 volumes is 20.3 TiB (§2.3), SepBIT reduces the overall memory overhead from $20.3 \cdot \frac{2^{40}}{2^{12}} \cdot 8 = 41.6$ GiB to $41.6 \cdot (1 - 71.8\%) = 11.7$ GiB.

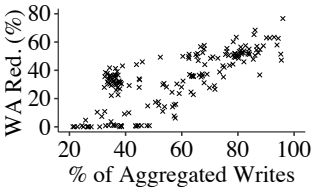


Figure 18: Exp#7 (Impact of workload skewness).

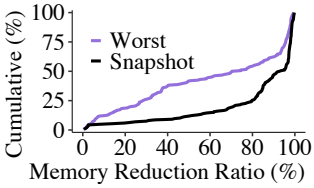
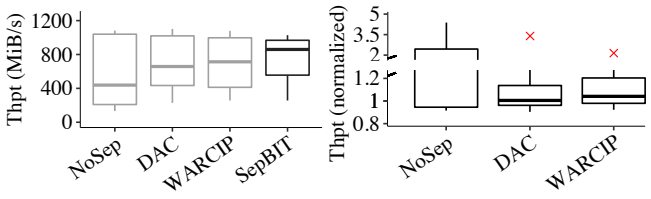


Figure 19: Exp#8 (Memory overhead).

Figure 19 further depicts the cumulative distributions of the memory overhead reductions across volumes under both the worst case and the snapshot case. In the worst case, SepBIT reduces the memory overhead by more than 72.3% in half of the volumes and the highest memory overhead reduction is 99.5%; in the snapshot case, the median reduction is 93.1% with the highest reduction as 99.7%. In the snapshot case, the 25th, 50th, and 75th percentiles of the number of unique LBAs across volumes are 99 K, 1,063 K, and 6,190 K, respectively, while the 25th, 50th, 75th percentiles of the number of total LBAs in the FIFO queue across volumes are 398 K, 2,242 K, and 8,857 K, respectively. The reason of the differences among volumes is their different degrees of skewness. The volumes with higher skewness see more aggregated traffic patterns, and hence the number of recently updated LBAs is much smaller compared with the write WSS.

Exp#9 (Prototype evaluation). We deploy our log-structured block storage system prototype (§3.4) on a machine equipped with an Intel Xeon Silver 4215 CPU, 96 GiB DDR4 RAM, and 4×128 GiB Intel Optane Persistent Memory modules. The machine runs Ubuntu 20.04.2 LTS with kernel 5.4.0.

Due to the limited storage capacity in our testbed machine, we focus on 20 volumes whose write traffic ranks the top 31-50 among the 186 volumes in the Alibaba Cloud traces. Their write traffic ranges from 0.82 TiB to 2.82 TiB, and their WAs under NoSep range from 1.00 to 4.96. Specifically, 9 volumes have their WAs less than 1.1, while 7 volumes have



(a) Absolute throughput (b) Normalized throughput

Figure 20: Exp#9 (Prototype evaluation).

their WAs greater than 3.0.

Also, our evaluation rate-limits user writes while GC is running due to the capacity constraint. The reason is that a GC operation removes the invalid blocks only after rewriting all valid blocks. If we issue user writes at full speed while GC is running, the storage space may run out. Thus, we limit the rate of user writes as 40 MiB/s while GC is running; otherwise, we issue user writes at full speed. We measure the *write throughput* (i.e., the number of user-written bytes divided by the total time for replaying each volume).

We compare SepBIT with NoSep, DAC, and WARCIP, based on our previous experiments that DAC and WARCIP perform the best among existing schemes and NoSep serves as the baseline. We configure the segment selection algorithm, the segment size, and the GP threshold as Cost-Benefit, 512 MiB, and 15%, respectively.

Figures 20(a) and 20(b) show the boxplots of the absolute write throughput and the normalized write throughput of SepBIT (w.r.t. NoSep, DAC, and WARCIP) in individual volumes for different schemes, respectively. SepBIT achieves the highest throughput for the 25th and 50th percentiles, at 556.1 MiB/s and 859.4 MiB/s, which are 28.3% and 20.4% higher than the second best, respectively.

For the 75th percentile, the absolute throughput of SepBIT is 6.9%, 5.2%, and 3.0% lower than those of NoSep, DAC, and WARCIP, respectively (Figure 20(a)). The reason is that such volumes (with top-25% throughput) have low WAs (less than 1.1) and hence are less affected by GC. Compared with other schemes, SepBIT spends extra time to access the FIFO queue (§3.4) and has slightly degraded throughput.

5 Related Work

GC in SSDs. We evaluated several existing data placement schemes (§4.1) for mitigating the WA of flash-level GC in SSDs. Other data placement schemes build on the use of program contexts [19] or the prediction of block temperature based on neural networks [44]. Some empirical studies evaluate the data placement algorithms on an SSD platform [22], or characterize how real-world I/O workloads affect GC performance [41]. In particular, Yadgar *et al.* [41] also investigate the impact of the number of separated classes in data placement based on the temperature-based data scheme MultiLog [35]. In contrast, SepBIT builds on the BIT for data placement, backed by the empirical studies from real-world I/O traces. ML-DT [8] uses neural networks to predict the block death time. Compared with ML-DT, SepBIT infers BITs only with the last user write time in a simpler manner.

Besides data placement, existing studies propose segment selection algorithms to reduce the WA of flash-level GC. In addition to Greedy and Cost-Benefit (§2.1), Cost-Age-Times [11] considers the cleaning cost, data age, and flash erasure counts in segment selection. Windowed Greedy [17], Random-Greedy [24], and d-choices [36] are variants of Greedy in segment selection. Desnoyers [14] models the WA of different segment selection algorithms and hot-cold data separation. SepBIT can work in conjunction with those algorithms.

GC in file systems. Several studies examine the GC performance for log-structured file systems. Matthew *et al.* [26] improve the GC performance by adapting GC to the system and workload behaviors. SFS [27] separates blocks by hotness (i.e., write frequency divided by age). Some studies reduce WA using file system semantics in data placement; for example, WOLF [38] groups blocks by files or directories, while hFS [47] and F2FS [21] separate data and metadata. Extending SepBIT with file system awareness is a future work.

GC for RAID and distributed storage. Some studies address the GC performance issues in RAID and distributed storage, such as reducing the WA of Log-RAID systems [13] and mitigating the interference between GC and user writes via GC scheduling in RAID arrays [19, 34]. RAMCloud [31] targets persistent distributed in-memory storage. It proposes two-level cleaning to maximize memory utilization by coordinating GC operations in memory and disk backends. It also corrects the original Cost-Benefit algorithm [30] for accurate segment selection. Our work focuses on data placement for WA mitigation and is orthogonal to those studies.

6 Conclusion

We propose SepBIT, a novel data placement scheme that mitigates WA caused by GC in log-structured storage by grouping blocks with similar estimated BITs. Inspired from the ideal data placement that minimizes WA (i.e., WA=1) using future knowledge of BITs, SepBIT leverages the skewed write patterns of real-world workloads to infer BITs. It separates written blocks into user-written blocks and GC-rewritten blocks and performs fine-grained separation in each set of user-written blocks and GC-rewritten blocks. To group blocks with similar BITs, it infers the BITs of user-written blocks and GC-rewritten blocks by estimating their lifespans and residual lifespans, respectively. Evaluation on production traces shows that SepBIT achieves the lowest WA compared with eight state-of-the-art data placement schemes.

Acknowledgements. We thank our shepherd, Keith Smith, and the anonymous reviewers for their comments. This work was supported in part by Alibaba Group via the Alibaba Innovation Research (AIR) program. The corresponding author is Patrick P. C. Lee.

References

[1] Alibaba Cloud ESSDs. https://www.alibabacloud.com/help/doc-detail/122389.htm.

[2] Intel Optane Persistent Memory 128GB Module. https://ark.intel.com/content/www/us/en/ark/products/190348/intel-optane-persistent-memory-128gb-module.html.

[3] ZenFS. https://github.com/westerndigitalcorporation/zenfs.

[4] Zoned storage. https://zonedstorage.io/.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proc. of USENIX FAST*, 2008.

[6] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. CORFU: A distributed shared log. *ACM Trans. on Computer Systems*, 31(4):10, 2013.

[7] 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 *Proc. of USENIX ATC*, 2021.

[8] C. Chakraborttii and H. Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In *Proc. of ACM SYSTOR*, 2021.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of USENIX OSDI*, 2006.

[10] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proc. of ACM SIGMETRICS*, Jun 2009.

[11] M. Chiang and R. Chang. Cleaning policies in mobile computers using flash memory. *Journal of Systems and Softwares*, 48(3):213–231, 1999.

[12] M.-L. Chiang, P. C. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, 29(3):267–290, 1999.

[13] T.-c. Chiueh, W. Tsao, H.-C. Sun, T.-F. Chien, A.-N. Chang, and C.-D. Chen. Software orchestrated flash array. In *Proc. of ACM SYSTOR*, 2014.

[14] P. Desnoyers. Analytic models of SSD write performance. *ACM Trans. on Storage*, 10(2):1–25, 2014.

[15] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *Proc. of ACM SOSP*, 1993.

[16] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. The unwritten contract of solid state drives. In *Proc. of ACM EuroSys*, 2017.

[17] X. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proc. of ACM SYSTOR*, 2009.

[18] J. Kim, K. Lim, Y. Jung, S. Lee, C. Min, and S. H. Noh. Alleviating garbage collection interference through spatial separation in all flash arrays. In *Proc. of USENIX ATC*, 2019.

[19] T. Kim, D. Hong, S. S. Hahn, M. Chun, S. Lee, J. Hwang, J. Lee, and J. Kim. Fully automatic stream management for multi-streamed SSDs using program contexts. In *Proc. of USENIX FAST*, 2019.

[20] K. Kremer and A. Brinkmann. FADaC: A self-adapting data classifier for flash memory. In *Proc. of ACM SYSTOR*, 2019.

[21] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A new file system for flash storage. In *Proc. of USENIX FAST*, 2015.

[22] J. Lee and J.-S. Kim. An empirical study of hot/cold data separation policies in solid state drives (SSDs). In *Proc. of ACM SYSTOR*, 2013.

[23] J. Li, Q. Wang, P. P. C. Lee, and C. Shi. An in-depth analysis of cloud block storage workloads in large-scale production. In *Proc. of IEEE IISWC*, 2020.

[24] Y. Li, P. P. C. Lee, and J. C. S. Lui. Stochastic modeling of large-scale solid-state storage systems: Analysis, design tradeoffs and optimization. In *Proc. of ACM SIGMETRICS*, 2013.

[25] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *Proc. of USENIX FAST*, 2016.

[26] J. N. Matthews, D. S. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proc. of ACM SOSP*, 1997.

[27] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random write considered harmful in solid state drives. In *Proc. of USENIX FAST*, 2012.

[28] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[29] Z. Pang, Q. Lu, S. Chen, R. Wang, Y. Xu, and J. Wu. ArkDB: A key-value engine for scalable cloud storage services. In *Proc. of ACM SIGMOD*, 2021.

[30] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. on Computer Systems*, 10(1):26–52, 1992.

[31] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured memory for DRAM-based storage. In *Proc. of USENIX FAST*, 2014.

[32] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proc. of USENIX ATC*, 1993.

[33] M. Shafaei, P. Desnoyers, and J. Fitzpatrick. Write amplification reduction in flash-based SSDs through extent-based temperature identification. In *Proc. of USENIX HotStorage*, 2016.

[34] J. Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon. Gecko: Contention-oblivious disk arrays for cloud storage. In *Proc. of USENIX FAST*, 2013.

[35] R. Stoica and A. Ailamaki. Improving flash write performance by using update frequency. *Proc. of the VLDB Endowment*, 6(9):733–744, 2013.

[36] B. Van Houdt. A mean field model for a class of garbage collection algorithms in flash-based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):191–202, 2013.

[37] B. Van Houdt. On the necessity of hot and cold data identification to reduce the write amplification in flash-based SSDs. *Performance Evaluation*, 82:1–14, 2014.

[38] J. Wang and Y. Hu. WOLF - A novel reordering write buffer to boost the performance of log-structured file systems. In *Proc. of USENIX FAST*, 2002.

[39] Q. Wang, J. Li, P. P. C. Lee, T. Ouyang, C. Shi, and L. Huang. Separating data via block invalidation time inference for write amplification reduction in log-structured storage. Technical report, The Chinese University of Hong Kong, 2022. `https://www.cse.cuhk.edu.hk/~pclee/www/pubs/tech_sepbit.pdf`.

[40] E. Xu, M. Zheng, F. Qin, Y. Xu, and J. Wu. Lessons and actions: What we learned from 10k SSD-related storage system failures. In *Proc. of USENIX ATC*, 2019.

[41] G. Yadgar, M. Gabel, S. Jaffer, and B. Schroeder. SSD-based workload characteristics and their performance implications. In *ACM Trans. on Storage*, 2021.

[42] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan. AutoStream: Automatic stream management for multi-streamed SSDs. In *Proc. of ACM SYSTOR*, 2017.

[43] J. Yang, S. Pei, and Q. Yang. WARCIP: Write amplification reduction by clustering I/O pages. In *Proc. of ACM SYSTOR*, 2019.

[44] P. Yang, N. Xue, Y. Zhang, Y. Zhou, L. Sun, W. Chen, Z. Chen, W. Xia, J. Li, and K. Kwon. Reducing garbage collection overhead in SSD based on workload prediction. In *Proc. of USENIX HotStorage*, 2019.

[45] Y. Yang and J. Zhu. Write skew and Zipf distribution: Evidence and implications. *ACM Trans. on Storage*, 12(4):1–19, 2016.

[46] Y. Zhang, P. Huang, K. Zhou, H. Wang, J. Hu, Y. Ji, and B. Cheng. OSCA: An online-model based cache allocation scheme in cloud block storage systems. In *Proc. of USENIX ATC*, 2020.

[47] Z. Zhang and K. Ghose. hFS: A hybrid file system prototype for improving small file and metadata performance. In *Proc. of EuroSys*, 2007.

CacheSifter: Sifting Cache Files for Boosted Mobile Performance and Lifetime

Yu Liang[1,2], *Riwei Pan*[1], *Tianyu Ren*[1], *Yufei Cui*[1], *Rachata Ausavarungnirun*[3], *Xianzhang Chen*[4*], *Changlong Li*[5*], *Tei-Wei Kuo*[1,6,7], *and Chun Jason Xue*[1]

[1] *Department of Computer Science, City University of Hong Kong*

[2] *School of Cyber Science and Technology, Zhejiang University*

[3] *TGGS, King Mongkut's University of Technology North Bangkok*

[4] *College of Computer Science, Chongqing University*

[5] *School of Computer Science and Technology, East China Normal University*

[6] *Department of Computer Science and Information Engineering, National Taiwan University*

[7] *NTU High Performance and Scientific Computing Center, National Taiwan University*

Abstract

Mobile applications often maintain downloaded data as cache files in local storage for a better user experience. These cache files occupy a large portion of writes to mobile flash storage and have a significant impact on the performance and lifetime of mobile devices. Different from current practice, this paper proposes a novel framework, named CacheSifter, to differentiate cache files and treat cache files based on their reuse behaviors and main-memory/storage usages. Specifically, CacheSifter classifies cache files into three categories online and greatly reduces the number of writebacks on flash by dropping cache files that most likely will not be reused. We implement CacheSifter on real Android devices and evaluate it over representative applications. Experimental results show that CacheSifter reduces the writebacks of cache files by an average of 62% and 59.5% depending on the ML models, and the I/O intensive write performance of mobile devices could be improved by an average of 18.4% and 25.5%, compared to treating cache files equally.

1 Introduction

Mobile devices are now dominant in people's daily lives [23, 39]. Almost all mobile applications need to download files or data from networks because of the dynamic nature of applications and overall system optimization. Even with the high bandwidth of modern communication networks (e.g., WiFi and 5G), many applications still rely heavily on data cached on mobile devices to avoid re-downloading data through the network and meet their execution latency demands. Current mobile devices store cache files in the main memory first and then write them back to flash storage. These applications' cached data are usually managed as cache files and can be re-accessed quickly [9, 41]. However, the number and the size of applications' cache files have grown exponentially in recent years, as applications demand increasing amounts of data. For example, Facebook can generate 1.2GB of cache files on a mobile device in two hours [27]. In addition to performance degradation, most cache files are eventually written to the flash storage of a mobile device, increasing writes and thus decreasing the lifetime of flash devices [2, 45].

A number of research studies on mobile systems have been performed in recent years [11, 12, 16, 18, 20, 23, 26, 27, 32, 37, 38]. These techniques include optimization of memory management [23, 26], defragmentation [11], storing cache files in memory [32, 38], re-designing the directory cache of mobile systems [37], an application-aware swapping mechanism [20], and I/O management [12, 16, 18]. Unfortunately, little work exists that has differentiated cache files in management. Although Liang et al. [27] elucidate differences among cache files, a solution was not proposed. Since the total size of cache files has increased dramatically, improper writebacks of cache files to flash storage will markedly reduce the lifetime of the flash storage of a mobile device. It is also worth noting that some cache files are used only once throughout their lifetime while others may be re-accessed multiple times before deletion. In current practice, however, cache files are treated equally.

Android operating systems store cache files in local storage in consideration of performance and latency [9]. However, cached data on a mobile device can significantly reduce the lifetime of its flash storage, as the replacement cycle of smartphones increases [40]. Recent works propose to store cached data in DRAM to reduce writebacks, and thus can improve both system performance and lifetime [32, 38]. These techniques suffer from two major problems as applications increase their demand for cached files. First, cached data vary greatly in frequency of access, lifetime, and size. Treating them equally leads to inefficiency. Second, the available memory is insufficient in mobile devices, maintaining useless cache files could degrade the overall system performance because of memory competition. *The goal* in this work is to improve *both* system performance and the lifetime of flash storage by managing cache files according to their reuse behaviors.

The proposed novel cache file management framework,

*Corresponding authors: Changlong Li, Email: clli@cs.ecnu.edu.cn; Xianzhang Chen, Email: xzchen@cqu.edu.cn.

named CacheSifter, dynamically categorizes cache files into different categories using a light-weighted machine learning (ML) algorithm and dynamically places the cache files of different categories in DRAM or flash storage based on their data access patterns. Three cache-file categories are proposed based on their revisiting possibility: Burn-After-Reading (BAR) files, Transient files, and Long-living files. A quasi-in-memory file system is proposed for better management of Transient files in DRAM and to avoid the operating system from accidentally evicting them out of DRAM. A cache-file eviction mechanism is also developed to utilize DRAM more effectively in keeping cache files.

CacheSifter adheres to the semantics of the Android cache file management and does not produce new safety vulnerability. Experimental results over popular applications show that CacheSifter reduces the writebacks of cache files by an average of 62% and 59.5% depending on the ML models, and the I/O intensive write performance of mobile devices could be improved by an average of 18.4% and 25.5%. [1]

2 Cache Files in Mobile Systems

Unlike servers' applications, most mobile applications frequently download fresh data such as news and videos from networks. Mobile systems generally store the downloaded data as cache files in local storage temporarily to reduce redundant data downloads. For example, Android systems maintain temporary cache files in the main memory for a period of time (30 seconds by default) and then write them back to flash storage [9, 41]. This is similar in spirit to how Linux manages its files, which treats all files equally. Writing *all* of the cache files back into flash storage will significantly degrade system performance [8], reduce the lifetime of flash storage [2], and occupy large storage space, which is markedly limited in mobile devices. With the exponential growth of mobile applications' cache files induced by high-speed networks in recent years, optimization of their management has become urgent.

While previous works [32, 38] aim to maintain all cache files of targeted applications in the main memory to accelerate cache files' accesses, the total size of cache files for an application can occupy a significant portion of the main memory, which could substantially degrade the performance of the other running applications via memory contention. This paper, however, aims to manage cache files according to their access patterns and thus only necessary cache files will be stored in main memory or storage.

2.1 Required Space and Writes of Cache Files

This section aims to quantify cache file usages in current Android systems via both static and dynamic methods.

[1] Note that the reduction of writebacks can substantially improve the lifetime of flash-memory storage and notably benefit I/O-intensive phases in application execution, application launch, and application installation, which are crucial to the mobile user experience [3].

Required space for cache files. The storage occupation of cache files is determined by taking snapshots of cache files in storage. We survey 60 volunteers[2] with real mobile device usage, including 42 models of 5 vendors, for one week. We collect the snapshots of cache files for the commonly-used applications (4-15 applications according to volunteers' usage behaviors) once per day on 50 of the mobile devices. On the other ten mobile devices, data are collected three times per day. Table 1 presents the total size of cache files of each mobile device and different applications. We choose the most-commonly-used application for each type on each smartphone. Some smartphones might be missing certain types of applications due to different user behaviors.

Table 1: Cache files' sizes of different devices or applications.

Group by	Code name	No. of devices	Average (GB)	Max (GB)
Vendors	Huawei	30	0.4	1.79
	Oppo	6	4.11	8.82
	Vivo	6	1.58	1.85
	Xiaomi	17	2.17	4.55
	Meizu	1	1.68	1.7
Apps	Social media	60	0.35	2.73
	Video	57	0.22	2.23
	Website	60	0.26	5.25
	Game	11	0.18	1.33

The collected data shows that cached file size varies greatly between vendors and applications. Based on the data collected from these mobile devices, it is found that some mobile systems or third-party software delete cache files. Moreover, some users habitually delete cache files to alleviate the shortage of storage space. However, even though these cache data are deleted after writes, their damage to the performance and lifetime of flash storage has already occurred. As a consequence, it is critical to evaluate the actual writes of cache files during run-time.

Write behaviors. The writes of cache files are now profiled from two perspectives: user behaviors and representative applications. We first collect the write size of cache files under volunteers' usage behaviors by instrumenting the source code of Linux in the experimental mobile devices to collect every write of cache files to flash storage. Five volunteers used the experimental mobile devices for three days.

The collected data reveals that the writes of cache files can reach 500MB per day, even for users that spend less than three hours per day on their mobile devices. Data from a mobile device vendor (top-five) shows that the total writes of their testing users is about 10GB on average and up to 30GB per day. Cache file writes is approximately 6.4GB on average per day and up to 19.2GB because cache file writes represent an average of 64% of total writes to storage for mobile devices based on experimental results, as shown in Figure 1.

The writes of cache files of the top-20 representative applications are collected, including social media, map, game, video, and browser. In this experiment, the volunteers used each application continuously for two hours. The ratio of

[2] The volunteers are 18-60 years old.

cache file writes to total writes for the twenty applications is presented in Figure 1. The write count ratio is the ratio of the number of writes of cache files to the number of writes of total files. It is found that most applications write a large amount of cache files. For example, the write count ratio of Facebook is up to 92.6%, and the write size ratio of YouTube is as high as 95.7%. In contrast, CandyCrush is a stand-alone game that does not need to download much data from the network.

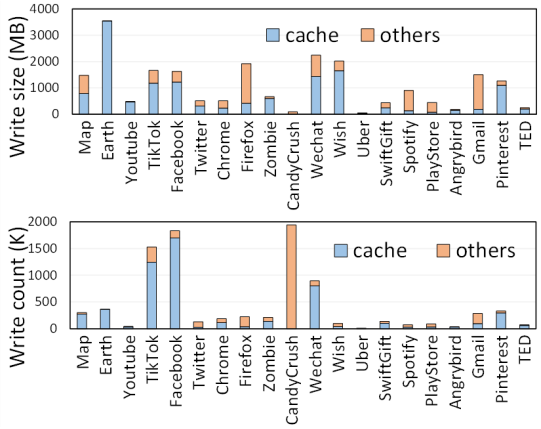


Figure 1: Write count/size of representative applications in two hours.

Existing applications write many cache files into flash storage during run-time. Even though many cache files are deleted by the system or applications, the cache files still occupy large storage space. Notably, the write and delete operations of cache files not only increase I/O contention, which could degrade I/O performance, but also shorten the lifetime of flash devices [10, 21]. In addition, the problem will become increasingly severe with the continual increase of network speed, growing usage of applications, and use of newer flash chips (e.g., TLC, QLC, and 3D-NAND flash) with shorter endurance [13, 17, 44].

2.2 Differences among Cache Files

As mentioned above, existing Android systems treat all cache files as normal files that always require persistent storage. Cache files are time-sensitive data, however, and it is often unnecessary for them to be persistently stored. Based on observations at the block layer, Liang et al. [27] proposed to classify all cache files into three categories, i.e., Burn-After-Reading (BAR), Transient, and Long-living, according to the distinctly varied access patterns of cache files in flash storage.

After defining the categories of files, numerous questions arise regarding how files are categorized and how categories are managed in a practical system, none of which offer straightforward answers. However, all of these questions are addressed in this paper. Furthermore, paper [27] demonstrates differences between cache files at the block layer. This paper finds that access information at the VFS layer is more suitable for categorizing cache files because the cache files should be handled prior to the block level. Therefore, this paper reuses the name of three types of files in paper [27] but with different definitions.

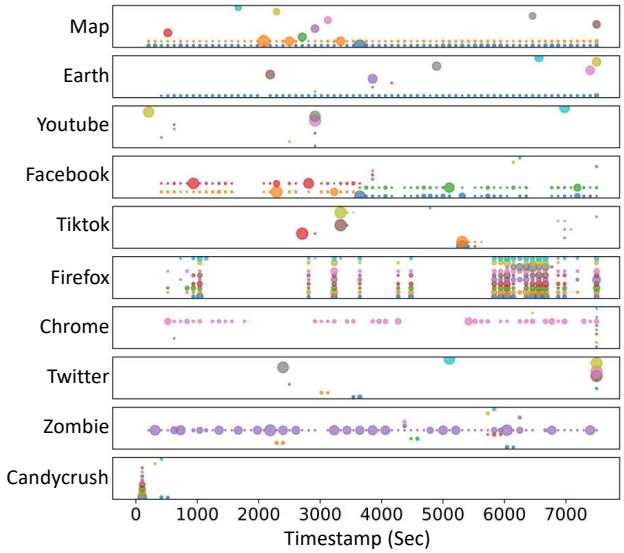


Figure 2: Access patterns of the top-10 accessed cache files of ten representative applications.

Burn-After-Reading (BAR) represents cache files that only have tiny re-accesses that take place at the beginning of their lifetime. For many typical mobile applications, most cache files are rarely re-accessed, which is similar to the conclusion reached in a previous study [4]. In particular, some cache files in the flash storage were deleted without being re-accessed at all. Consequently, there is no need to write such cache files to storage.

Transient refers to cache files that have a large re-access count as well as a short active period. Figure 2 shows the access patterns of the top-10 accessed cache files (denoted by ten colors) of ten representative applications. The size of a circle indicates the access count of the corresponding cache file within 100 seconds. The access patterns of the cache files vary greatly. Moreover, some of these files have a short active period and a large access count. For example, the third cache file (labeled in green) of Map is only re-accessed within a short time after it is created. Accordingly, we deem such cache files as Transient files, since the applications only re-access them in the near future.

Long-living represents the rest of cache files, especially the files that are re-accessed frequently over a long period of time.

2.3 Challenges in Cache File Management

Even though Liang et al. [27] proposed to manage cache files following their access patterns, they did not explore classification or management methods of cache files. Two major challenges exist in the management of cache files. First, cache files' behaviors change over time. For this reason, it is im-

portant that management should be adaptive to the run-time behavior of cache files. Second, existing systems store cache files according to the same routine. It is necessary, however, to manage dissimilar types of cache files by different policies. The main goal of this paper is to improve both system performance and storage lifetime. We will explore the access patterns of applications' cache files and consider the characteristics of DRAM-based main memory and flash-based storage of mobile devices in terms of performance and endurance.

3 CacheSifter Design

We propose CacheSifter to categorize cache files and manage them by exploiting their access patterns.

3.1 Overview of CacheSifter

3.1.1 Design Principles

We discuss five design principles for categorizing and managing cache files in mobile systems.

User application transparency. CacheSifter should have an insignificant impact on user experience. CacheSifter should also be compatible with the semantics of existing mobile systems requiring zero changes in existing user applications.

Online Categorization. While offline profiling simplifies the categorization process, an offline classifier cannot adapt to the dynamic system status and the configuration of users during usage of the mobile device. As a result, CacheSifter needs to be able to categorize cache files online while avoiding the extra overhead of storing BAR files and Transient files.

Adaptive memory management. CacheSifter always attempts to maintain the cache files that will be used in the main memory to achieve high file access performance. However, using too much memory for the cache files may degrade system performance. In this case, CacheSifter should adaptively adjust its memory usage along with different active applications.

Adapt to changes in user behavior. CacheSifter should adapt to changes in user behaviors. A categorized file may need to be re-categorized. For example, a file is categorized as a BAR file because it is only used once immediately after it is downloaded. When user behavior changes, and it is used many time repeatedly, it should be re-categorized into a Transient or Long-living file to avoid frequent re-download.

Ensure safety when deleting cache files. CacheSifter should not produce any new vulnerabilities as compared to existing mobile systems. Since CacheSifter may discard BAR files and Transient files during execution of applications, it is critical that discarding data by CacheSifter will not cause an application crash or user data loss.

3.1.2 CacheSifter Framework

Following these principles, we design CacheSifter to categorize cache files and manage them in DRAM/flash storage according to their reuse patterns to avoid unnecessary writing back. Figure 3 shows the framework of CacheSifter. CacheSifter lives in the kernel rather than in an intermediate or less-privileged layer. CacheSifter can directly categorize cache files on the page cache without additional memory consumption and data copy. CacheSifter also does not require any changes in existing user applications, i.e., it is transparent to user applications.

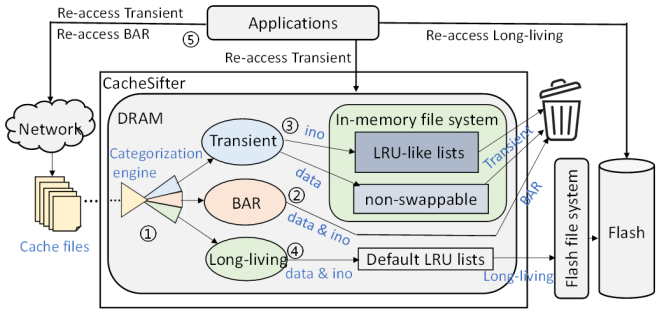


Figure 3: Framework of CacheSifter.

All newly downloaded cache files are first maintained in the main memory and wait for categorization. ① CacheSifter adopts a lightweight machine-learning-based categorization engine (See Section 3.2.1) to divide the newly downloaded cache files into three types, i.e., BAR, Transient, and Long-living in two stages online. ② To better utilize memory and storage, CacheSifter discards all BAR files because they are typically not reused. ③ For Transient files, CacheSifter keeps them in DRAM by using a quasi-in-memory file system, which is designed to avoid an accidental swap-out of Transient files. CacheSifter discards Transient files when there is insufficient DRAM space using an LRU-like file eviction policy (See Section 3.2.2). ④ For Long-living files, CacheSifter writes them to the flash storage by exploiting the default LRU-based eviction scheme of the Android system. Moreover, CacheSifter deletes the cache files from their corresponding storage when they are invalidated by applications. ⑤ The deleted files will be re-downloaded from the network when they are accessed in the future, which provides an opportunity to change the categorization of cache files according to changes in user behavior (See Section 3.3). Finally, CacheSifter exploits a safe list mechanism to maintain known potential paths of cache files that are important to users, or in cases in which their deletion could threaten system stability (See Section 3.4).

CacheSifter provides three key benefits. First, CacheSifter avoids pushing-out the BAR and Transient files to flash storage, which reduces write contention, extends the lifetime of the flash storage, improves overall system performance, and conserves storage space. Second, Transient files are accessed directly from DRAM, improving the access latency of this type of cache files. Third, CacheSifter significantly optimizes the management of cache files with a lightweight machine-learning-based engine in the kernel, which not only has negligible overhead but is also transparent to user applications.

3.2 Feature-based Cache Files Management

The effectiveness of CacheSifter relies highly on the accuracy of the categorization engine. The overhead of the categorization engine and the cache file management mechanism of cache files should be as small as possible to minimize the impact on system performance. In this section, we describe the design of these two key components.

3.2.1 Lightweight Categorization of Cache Files

Machine learning based categorization. According to the design principles, categorization should be both lightweight and conducted online. Based on our observations, we know the categorization of each cache file by observing its reuse patterns. However, to avoid writebacks of BAR and Transient files, this method requires storing all cache files and their access information in main memory for a long time period for categorization, which imposes a high cost. Heuristic-based methods, such as suffix based methods [14, 24], can categorize files with a small cost. However, they do not consider the access patterns of cached files, and thus cannot be used to recognize BAR, Transient, and Long-living files. For example, a video (.exo) file could be any type of cache file according to user behaviors. Moreover, since the users' behavior and access pattern of cache files are different across different applications, we expect non-ML approaches to be less flexible and generalized. Thus, the categorization engine in CacheSifter utilizes machine-learning-based schemes to automatically perform categorization based on features within a short period of time and observation-based labels.

This paper chooses a lightweight neural network method (MLP [5]) in the experiments because of its performance and low cost.[3] To further reduce cost, categorization is divided into two phases (BAR/non-BAR and Transient/Long-living) by exploiting two MLP models because we find that Long-living cache files cannot be recognized by short-time information. We train these two MLP models offline and use them for online categorization, and thus this method needs to retrain the models after a period of time to adapt to applications' changes. Certainly, one can also choose a lightweight reinforcement learning method [35] to avoid retraining, which is beyond the scope of this paper.

Metrics for analyzing prediction models. Three metrics are used to evaluate our categorization models. First, we use *Accuracy* to reflect how correctly the model predicts the categories of files, as shown in Equation 1:

$$Accuracy = (TP + TN)/Total\_Instances \quad (1)$$

Where TP is an outcome in which the model correctly predicts the positive class; a true negative TN is an outcome in which the model correctly predicts the negative class. Considering the penalty of misclassification, positive class is non-BAR in the first phase of categorization in which the negative class is BAR. In the second phase of categorization, Long-living is denoted as the positive class, while Transient is the negative class. Based on our observation, the data of each category is highly unbalanced. Therefore, the above *Accuracy* cannot represent the accuracy of each type of file. Accordingly, we introduce another metric, *Recall*, in Equation 2:

$$Recall = TP/(TP + TN). \quad (2)$$

In the high-recall model, we care more about the predicted accuracy of files with high mis-predicted overhead, such as long-living files. If a long-living file is incorrectly predicted as a BAR or Transient file, it could induce redownload overhead. Finally, to visualize the results, we also use the third metric, *PR curve,* which is simply a graph with Precision values on the y-axis and Recall values on the x-axis. A good PR curve has a large area under curve (AUC).

Based on these three metrics, we train high-recall and high-accuracy models with a high PR curve. The high-recall model aims to reduce writebacks of cache files and minimize re-download overhead; whereas, the high-accuracy model aims to reduce writebacks of cache files with minimum mis-categorization.

3.2.2 Cache File Management Mechanism

To better utilize memory/storage to reduce writebacks of cache files and minimize re-download penalty, cache files are processed according to their categorization.

BAR file. BAR files are deleted immediately after categorization because these files are not likely to be reused.

Transient file. Since the usage of Transient files in mobile applications exhibits both strong locality and time sensitivity in a certain period of time, CacheSifter always attempts to maintain the Transient files in the main memory during their active period to achieve higher file access performance. At first, we try to exploit an existing file system, such as tmpfs or ramfs, to manage Transient files. However, to avoid writeback operations prior to the categorization of cache files, each cache file will have two inodes, i.e., one in F2FS and tmpfs/ramfs each, which complicates the implementation and brings additional overhead. As a consequence, a quasi-in-memory file system (QMFS) is proposed to manage Transient files in the main memory during their active period.

QMFS is implemented by two LRU-like lists (an active list and an inactive list), as shown in Figure 4. The active list is designed to ensure that files will not be deleted within their active period. The inactive list is used to balance memory pressure and file performance. Specifically, when memory is sufficient, files will be maintained in memory for a longer time to reduce the penalty of mis-classifications. In the default memory management, the LRU list of page cache is page-granularity since the pages of files cached in the main memory will be written back to storage. If a page of a Transient cache file is deleted, however, this means that the whole cache file

[3] We compare the performance of MLP, Random Forest, Linear Regression and Logistic Regression and find MLP to be the most effective.

in QMFS is invalid. Therefore, the LRU-like lists of QMFS in CacheSifter are maintained in file-granularity.

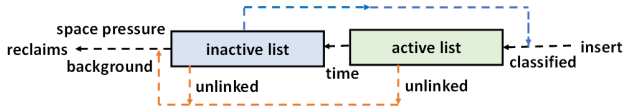


Figure 4: Eviction scheme of LRU-like lists in QMFS.

In QMFS, a cache file is put into the active list after categorization. Subsequently, the file in the *active list* may be moved into the *inactive list*, depending on the size of free memory and its existing time in the *active list*. If the existing time of a file is longer than its active period (a threshold), it will be moved to the *inactive list* to wait for deletion. Insufficient memory also triggers the movement action. If a file in the *inactive list* is referenced before it is deleted, it will be moved back to the *active list*. If a file is deleted, it will be deleted from the corresponding lists. If a file is truncated, CacheSifter works in the same manner as the default Android system. Specifically, the file's pages will be deallocated whereas the inode number will be still maintained in the LRU-like lists.

When the active period of Transient files ends, the Transient files generally will not be used again. For this reason, the longest-lived Transient files should have the highest priority to be evicted. Furthermore, to improve the performance of foreground applications, the cache files generated by a background application should also have a higher priority for eviction. We use UID to identify the files of background applications, as previously described [12]. After file eviction, memory space will be reclaimed. The parameters for reclaim are established in Section 5.2.

Long-living file management. Unlike the eviction schemes of BAR and Transient files, Long-living files are managed by the default page-based eviction scheme of the page cache in Android systems. Long-living files are maintained in the default LRU-based lists of the page cache. When a page of a Long-living file is unused for a long period of time, it will be evicted from the page cache and written back into the flash storage if it is dirty. Consequently, Long-living files will be stored in storage infinitely unless the applications delete them.

3.3 User Behavior Adaptation

Even if the feature-based cache file management worked well, user behaviors could change. Therefore, CacheSifter should be able to re-categorize cache files when user behavior changes to avoid frequent re-downloads. There are four types of state changes, as listed in Table 2. **BR**, **TR**, and **LL** represent the BAR category, the Transient category, and the Long-living category, respectively. Thus, "BR-> TR, LL" means that a BAR file shifts to a Transient file or a Long-living file.

Table 2 shows actions that trigger state changes of cache files, and the corresponding benefit or cost. When user behavior changes, CacheSifter only updates the category of cache files after re-download since CacheSifter performs categorization only when a file is newly-downloaded from the network.

Table 2: Actions based on state changes.

Types	State changes	Action	Benefit/Cost
(1)	**BR-> TR, LL**	Re-categorize files after re-download	None
(2)	**TR -> BR**	Do nothing and wait for discard	Memory space
(3)	**TR -> LL**	Re-categorize files after re-download	High performance
(4)	**LL -> BR, TR**	Do nothing	Flash space

CacheSifter treats and re-categorizes the re-downloaded file as a new file, and thus CacheSifter can adapt to stage change types (1) and (3) in Table 2. When type (2) stage change occurs, CacheSifter does not need to do anything, since Transient files will be discarded just like BAR files. Compared with BAR files, Transient files will remain in the main memory for a longer period of time and consume memory space. Type (4) is similar to type (2). Therefore, CacheSifter also does nothing, which consumes flash storage for a short time.

3.4 Safety Mechanism

CacheSifter discards the BAR and Transient files eventually. To make these operations safe for user data and applications, CacheSifter exploits a *safe_list* approach for cache file directories. It is not difficult to track and manage *safe_list* paths. In fact, Android now exploits these paths, which can be seen through the cache-delete button in the Android setting [6]. CacheSifter uses the same paths of the cache-delete button as the *safe_list* paths. Moreover, the *safe_list* can be managed offline. If vendors wish to optimize certain specific application, such as YouTube, they can obtain the cache paths of YouTube in advance and put them into the *safe_list*.

4 CacheSifter in Android

We implement CacheSifter in the Android system as a case study for mobile systems. In our implementation, CacheSifter categorizes cache files by using a dedicated thread. In this section, we first present the details of MLP-based categorization. We then show how the categorized cache files are managed by the flash file system, F2FS [24], and the proposed QMFS. Finally, we discuss implementation considerations of CacheSifter.

4.1 MLP-based Cache File Categorization

Categorization features and labels. In order to avoid unnecessary writes of cache files, categorization should be completed as rapidly as possible by using as few features as possible. The challenge here is that the Long-living files cannot be recognized by short-time features easily. To accurately categorize cache files, we first perform a fast categorization to detect BAR files and then dedicate additional time for the second pass to further separate Transient and Long-living files. Importantly, the memory space overhead is not large because there are not many Long-living files. In addition, categorization should also adapt to changes in user behaviors. Therefore, the objective of feature design is to characterize the access patterns and attributes of each file with a low memory cost.

To achieve this goal, the access patterns (read, write, and I/O size) and attributes of files (file size and active period) are selected as the features for machine-learning methods. The designed features are shown in Figure 5.

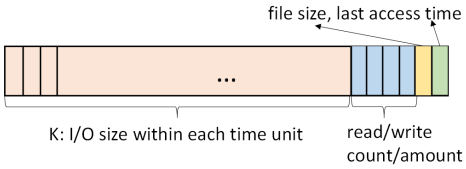


Figure 5: Features for learning.

In general, the system maintains K+6 features for each data point. The first K dimensions of a data point are sequential data that correspond to access information within the first K time units after creating a file, where each value represents the sum of access I/O size within one time unit. The following 4 features are read amount, read count, write amount, and write count within K time units, while the last 2 features are file size and active period. File size is the maximum size of each file within the K time units. The active period is calculated by the last access time minus the first access time within the K time units. If a file is accessed only once within this period, the last access time is set to be K time units. To achieve high accuracy and low memory cost, the value of K of the first phase of categorization (BAR or other) is smaller than the second phase of categorization (Transient or Long-living). When a new cache file is created, its access information during time K (e.g., 30s) will be recorded.

In addition to the features, the labels of cache files can be used to train the categorization model. Although we cannot use the observation-based categorization online due to its high overhead, the labels for the cache files in the training dataset can be obtained based on the observations of the reuse patterns throughout their lifetime. Specifically, we label a file as a BAR file ("1"), or a Transient file ("2") or a Long-living file ("3") according to their active period.

Dataset Collection. We instrument the source code of the Android kernel and use the Android Debug Bridge (**adb**) tool [7] to collect the access information and file size of cache files at the VFS layer in fs/read_write.c. Based on the collected data[4], the labels and features are obtained to train our machine learning models.

To make the model as general as possible, many data are collected. Our collected data includes four parts: (1) information from ten representative applications gathered over 20 hours; (2) information of the same applications by different users, in order to include more user behaviors; (3) information of the same applications after three months for checking the retraining period; and (4) information of different applications in order to assess the prediction accuracy of untrained applications. This case study aims to optimize these ten applications.

Categorization methods. With sufficient data with features $\mathbf{x}$ and labels y collected, the subsequent step is to find a proper machine learning model that learns the mapping $f(\cdot) : \mathbf{x} \rightarrow y$. In this work, we compare some simple machine learning methods and choose to use the popular Multi-Layer Perceptron (MLP) as it theoretically approximates any function if given sufficient capacity, according to the universal approximation theorem [5]. We choose to use a lightweight MLP layer that takes the features as input and outputs the categorization results. A large network capacity (size) causes great CPU and memory consumption while reducing the network capacity might decrease the performance. We use a grid search to find the best network capacity. We start from an over-parameterized neural network and evaluate its classification accuracy on the validation set. The network size is gradually decreased by re-training the network until its performance on accuracy starts decreasing. The same strategy is applied to other network hyper-parameters, which will be elaborated in Section 5.1.

We first train the categorization models and evaluate them offline (on a PC), which can assist tuning the parameters to identify the best models for cache file categorization. Then, the trained models will be used in the Linux kernel for dynamic categorization. When the optimized applications are upgraded, the models could need to be retrained. Based on our dataset (3), the model can still accurately predict the new data that are generated after at least three months. Therefore, the period of retraining could be longer than three months in our case. In this case, we also provide a fuse mechanism, CheckStop, to stop CacheSifter once the prediction accuracy is lower than a threshold. To avoid retraining, one can choose a lightweight reinforcement learning model for the categorization.

4.2 Management of Categorized Cache files

The management of categorized cache files mainly includes two parts: handling data pages and managing inodes. All of these pages and inodes are managed and maintained by several lists.

There are three lists in CacheSifter for inode management: *temp_list*, *category_list1*, *category_list2*. The categorization engine, which is a dedicated thread, wakes up periodically to scan these lists and control the migration of inodes among them. prior to categorization, the inodes of all cache files are maintained in *temp_list* after creation and their data pages are managed in the *unevictable_list* in the page cache layer to avoid accidental eviction caused by the Android system.

Two-phase Categorization. For categorization, the inodes in *temp_list* are moved to *category_list1* periodically to improve concurrency. In the first phase, the categorization engine scans *category_list1* and determines whether an inode is BAR. Then, the BAR inodes are deleted, and the remaining inodes in *category_list1* are migrated to *category_list2*. After the second phase of categorization, the data pages of Transient files remain in the *unevictable_list*, while the inodes of Transient

[4] Released in https://github.com/yliang323/CacheSifter.

files are stored in our LRU-like lists of the QMFS. The data pages and the inodes of Long-living files are moved to the default LRU lists (inactive file list) of the page cache layer, and they are set as dirty. They are then written back into flash storage by the default Android system.

4.3 Implementation Discussions

Adaptation of CacheSifter. Vendors train models by using the dataset of targeted applications. When CacheSifter is deployed on different mobile devices, the machine learning model does not need to be retrained because it is based on the behaviors of applications. A large training dataset can cover extensive user behavior with a small implementation overhead under the selected machine learning method.

Stop CacheSifter in unforeseen cases. To handle some rare cases, we design a lightweight prediction checking mechanism, named CheckStop, to determine if CacheSifter should be stopped. The main idea here is to calculate the re-download rate by recording the hash values of downloaded files and deleted files in a time window. If the rate is larger than a threshold, CacheSifter is suspended. To minimize overhead, CheckStop only works when CacheSifter detects abnormal signals such as a significant change in the number of write-backs or file creations with the same hash value.

CacheSifter in the future. CacheSifter could be more useful for future generations of mobile devices for the following three reasons. First, with a faster network, more data could be accessed and cached per time unit, and thus the amount of cache files could be increased. Second, with the usage of new flash chips(e.g., TLC, QLC), storage lifetime is becoming increasingly crucial since the endurance of many new flash devices have become smaller. Third, the memory capacity of mobile devices is growing, which can support more in-memory cache files and better machine-learning-based categorization methods. Additionally, CacheSifter can be used in other Internet of Things (IoT) systems or automotive systems.

5 Evaluation Methodology

We implement and evaluate CacheSifter on real mobile devices with two different categorization models.

5.1 Categorization Models

In current Android systems, cache files are maintained in the main memory for 30 seconds by default and then written back to the flash storage. For this reason, in the evaluation, we label a file as a BAR file ("1") if its active period is shorter than 30 seconds to avoid extra memory usage. Rather than writing BAR files back to flash storage, CacheSifter deletes them after their categorization. If the active period of a file is longer than 30 seconds but smaller than 90 seconds, it will be labeled as a Transient file ("2"). Otherwise, it is labeled as a Long-living file ("3"). According to the active period of a set of cache shown in Figure 6, the majority of cache files (93%) in this dataset are BAR files. Consequently, a large number of writebacks of cache files can be avoid.

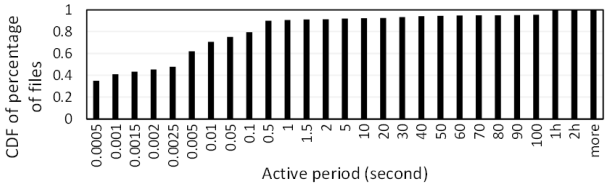


Figure 6: Active period of cache files.

To avoid using too much main memory, CacheSifter categorizes all of the cache files within 30 seconds. We utilize MLP as we found the mechanism to be the most accurate out of all other simple machine learning methods (random forest, linear regression and logistic regression). We test different parameters and present the results in Table 3.

Table 3: Categorization results by using different features.

Total time	Time unit	K	High Recall		High Accuracy	
			Recall	Accuracy	Recall	Accuracy
1s	0.01s	100	0.92	0.56	0.45	0.89
1s	0.05s	20	0.82	0.56	0	0.90
5s	0.25s	20	0.89	0.56	0	0.90
10s	0.5s	20	0.88	0.59	0	0.90
20s	1s	20	0.80	0.65	0.56	0.86
40s	1s	40	0.94	0.85	0.94	0.85
60s	1s	60	0.95	0.88	0.95	0.88
80s	1s	80	0.94	0.89	0.94	0.89

Our goal is to achieve enough accuracy or recall with small memory usage (smaller K). Therefore, we choose 20 and 60 as K_1 and K_2 for the first and second phases, respectively. Based on the feature within 20s, we can only choose high accuracy or high recall, and thus we use two models for different purposes.

Training models. The collected data are grouped by applications for training. We divide our datasets (1) and (2) (See Section 4.1) into 80% training and 20% testing instances. We use the training dataset to train a MLP network and exploit its "Accuracy" and "Recall" by evaluating the trained model on the testing dataset. We gradually decrease the neural network size by re-training the network until its accuracy performance starts to decrease. The same strategy is applied to other network hyper-parameters. All of the hyper-parameters are listed in Table 4.

The trained/re-trained model can be deployed to users' mobile devices as a system update. It is used for online categorization, which includes three parts: online feature collection, implementation of the trained MLP models, and the model-based categorization. First, we collect features of every new file for 20 seconds and maintain them in the main memory. A dedicated thread periodically wakes up to check the features and categorize the files. CacheSifter deletes BAR files after categorization and continues collecting features for files in other categories for an additional 40 seconds.

Categorization results of MLP models. We evaluate the

Table 4: Summary of hyper-parameters.

Hyper-parameters	For the first phase	For the second phase
number of hidden layers	4	4
hidden layer size	[512, 200, 2]	[512, 200, 2]
activation function 1	Tanh	Tanh
activation function 2	ReLU	ReLU
The function of output layer	Softmax	Softmax
loss_function	Focal loss function	Focal loss function
learning rate	0.1+MultiStepLR	0.1+MultiStepLR
optimizer	SGD + momentum = 0.5	SGD + momentum = 0.5
weight decay	1.00E-04	1.00E-05
sampler	WeightedRandomSampler	WeightedRandomSampler
batch size	200	200

trained models by ten representative applications and their random combinations. For the combinations, we use the first two letters to identify the application's name, and the results of which are shown in Figure 7. Some values are missed because the testing dataset may have just one type of cache files. For example, there are no non-BAR files in the testing dataset of Earth, and all of its data are predicted as the BAR class. Thus, it *AUC* is N/A, *Recall* is N/A, and *Accuracy* is 1. The results show that as long as an application has been trained, the model can classify its files well, irrespective of with what applications it is combined.

5.2 Evaluation Setup

We evaluate all of the experiments on two smartphones: (1) P9 equipped with an ARM Cortex-A72 CPU, 32GB internal flash memory and 3GB DRAM running Android 7.0 with Linux kernel version 4.1.18, and (2) Mate30 equipped with an ARM Cortex-A76 CPU, 128GB internal flash memory and 8GB DRAM running Android 10 with Linux kernel version 4.14.116. Ten representative applications, including social media, map, game, video, and browser, are used to collect features of cache files and evaluate CacheSifter. Their workload profiles (i.e., cache file ratio and data access patterns of their cache files) are presented in Figure 1 and Figure 2. We revise the kernel to print the access information of each file and the file attribute in functions new_sync_read() and new_sync_write() in fs/read_write.c. We filter the cache files by using the specific cache path of applications (/data/<packagename>/cache/).

We compare CacheSifter with the management scheme of cache files in default Android systems. The parameters of CacheSifter applied in the evaluation are listed in Table 5. To make a fair comparison, both the user and activities are the same for each comparison. For each testing, we follow the same sequence of actions: 1) we close all apps and clean their cache files prior to reboot to eliminate the impact of old cache files; 2) after reboot, we clean the cache to eliminate the impact of potentially buffered data; 3) we use the same application, login with the same user account, and conduct the same sequence of activities; and 4) We attempt our best to make each test the same, and we also conduct each test more than five times to eliminate possible nuances.

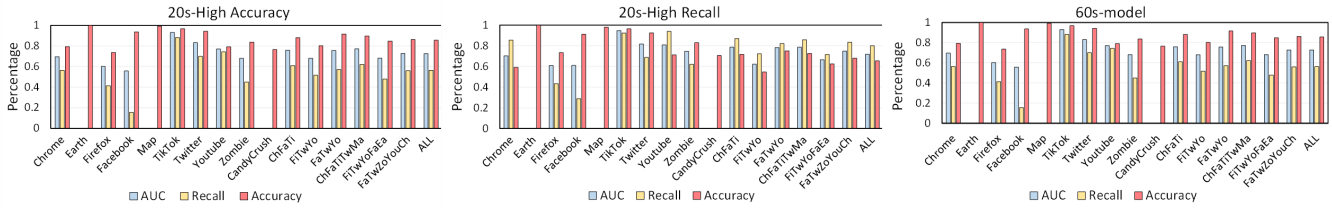
The parameters are selected based only on the targeted applications and independently of the experimental platform. The two smartphones run the same version of applications and use the same parameters.

Table 5: Summary of parameters used by CacheSifter.

Symbols	Semantics	Setting
K_1	The time for the first phase of categorization	20 seconds
K_2	The time for the second phase of categorization	60 seconds
T_1	The period of time for waking up the thread	10 seconds
E_1	Period of time to inactive	20 seconds
S_1	Size of each background reclaim	To W_1
T_2	The period of time for background reclaim	20 seconds
MS	Maximum RAM size for Transient files	20MB
W_1	Low watermark for background reclaim	50%*20MB
W_2	High watermark for foreground reclaim	90%*20MB
S_2	Size of each foreground reclaim	10%*20MB

Parameter configurations. K_1 and K_2 are the time to collect features of cache files for corresponding phases of categorization. Their values are determined as discussed in Section 5.1. T_1 is relative to CPU and memory consumption. If it is too small, the dedicated thread would run frequently and thus consume CPU time. On the other hand, if it is too large, the cache files will stay in the main memory for a long period of time to wait for categorization even if they already have enough features. Since we find the features within 20 seconds to be sufficient for the first phase of categorization, we choose 10 seconds to make sure the first phase can be finished within the default 30 seconds to avoid extra memory usage and frequent wake up. E_1 is the time that Transient files can be deleted. Since the active period of Transient files is 90 seconds in our evaluation, E_1 is 20 seconds (90-K_2-$T1$). S_1 is the reclaim space that to prepare for future usage, and it is related to W_1. T_2 does not need to be frequent because the reclaimed memory is enough for the next usage of Transient file within K_2. Therefore, we set it as 20 seconds according to our experience to reduce the CPU consumption. It is also not sensitive to the performance. These parameters do not need to be modified for different models of mobile devices if they use the same version of applications. If the versions of targeted applications are updated, the parameters MS, W_1, W_2, and S_2 may need to be changed due to workload changes. To show how to select these three parameters, we first present the cache file's size that was produced within 60 seconds in Figure 8.

Based on the data from Figure 8, we find that the maximum size of cache files of targeted applications is 21MB. Because not all files are transient, we configure MS, which is the upper bound of memory usage of the Transient files of targeted applications within K_2, to 20MB. This allows more memory to be used for other purposes. W_1 and W_2 are the watermarks of reclaims. Overall, the larger are their values, the more space will be used by Transient files; Thus, the re-access performance of Transient files is better but the performance of other applications could be worse because of memory contention. W_1 should be the maximum value of memory usage of the



(a) High accuracy model for the first phase. (b) High recall model for the first phase. (c) Model for the second phase.

Figure 7: Predict results of two MLP models for different applications and their combinations. For the combinations, the first two letters are used to identify the application's name. "ChFaTi" represents the combination of Chrome, Facebook, and TikTok.

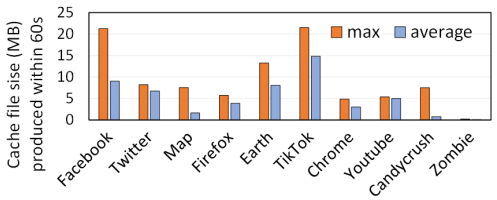


Figure 8: Cache files produced by applications within 60s.

Transient files of targeted applications within K_2, and 10MB (50% of MS) is enough for our case. W_2 and S_2 should be the minimum value that is just enough for the Transient files of targeted applications within K_2, and 2MB (10% of MS) is enough for our case. These parameters can be changed later for different platforms and manufacturers.

6 Evaluations

We evaluate CacheSifter's performance using two key metrics: (1) the reduction in writebacks of cache files and extension in lifetime of mobile flash storage; and (2) the improvement in read and write performance under intensive I/Os. We show the writeback reduction on two platforms while only show the other results on one platform since they are similar on different platforms and we do not have enough space for them.

6.1 Lifetime Improvement

Reduction in writebacks of cache files. We compare the writebacks of cache files and the number of block I/Os of CacheSifter against the default system. Since the results can vary under different user behaviors, each test is conducted ten times, the average results of which are shown in Figure 9. We evaluate both the high-recall model and the high-accuracy model.

The results reveal that writebacks of cache files vary for different applications. The reduction in writebacks when using the high-recall model is similar to that of the high-accuracy model in this experiment. Theoretically, the high-recall model constitutes a conservative-delete scheme that tends to keep cache files in the mobile device to reduce the penalty of re-download. In contrast, the high-accuracy model is a radical-delete scheme to pursue higher overall predict accuracy.

On P9, the writebacks of cache files are reduced by the high-recall model and the high-accuracy model by an average of 62% and 59.5%, respectively. The number of total I/Os is also significantly decreased by both models, i.e., an average of 29.7% and 31.2%, respectively. The high-accuracy model treats all of the three classes with the same priority. The high-recall model attempts to minimize incorrect predictions in the two cases (LL->BAR and LL->Transient) to reduce the re-download penalty. Since the long-living files are a small part of all cache files (less than 5%), as shown in Figure 6, the write reduction is similar under these two models.

On Mate30, the writebacks of cache files are reduced even more by both models, i.e., an average of 88.3% and 85.5%, respectively. The number of I/Os is also decreased more by both models, i.e., an average of 47.7% and 46.6%, respectively. The results on Mate30 show that the models trained by the data collected from P9 also work well on Mate30 because CacheSifter is platform-independent. There are two main reasons for the difference between the results on P9 and Mate30: different user behaviors, and the default system management schemes.

Based on Figure 6, 93% of the cache files are BAR in that dataset, but writebacks are not reduced as much in this case primarily because 1) the directory of cache files must be written back to flash storage to maintain consistency because there are some Long-living files that uses the directory information; 2) different user behaviors; and 3) the predict accuracy is not 100%.

Sensitivity Study. To evaluate the sensitivity of CacheSifter, we use the same parameters and the same models on P9 and Mate 30. The write reduction shown in Figure 9 indicates that both P9 and Mate 30 achieve similar benefits from CacheSifter. Moreover, we conduct a sensitivity study for the parameters in Table 5. The write reduction results on Mate30 with different MS are presented in Table 6. The sensitivity results show that the total writes could be affected by the value of MS due to different memory usage.

Boosted lifetime of mobile flash storage. Cai et al. [2] present the following method to compute the lifetime im-

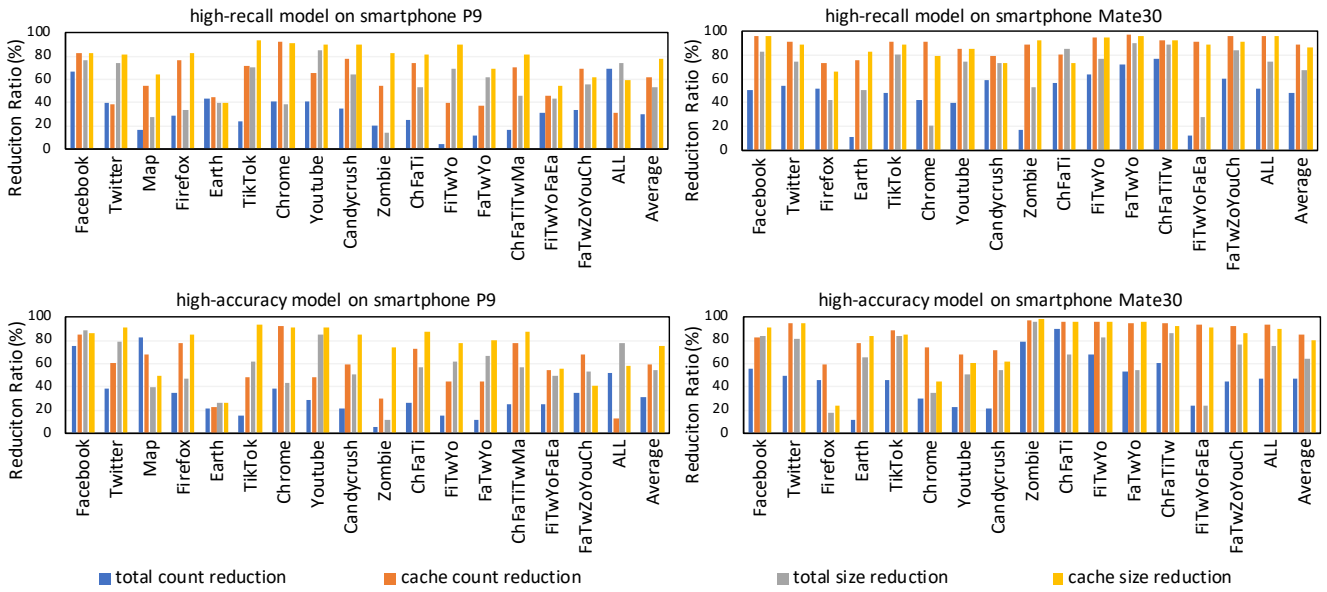


Figure 9: Normalized reduction ratio of cache files' writebacks and total I/Os. We evaluate the trained models with ten representative applications and their combinations. For the combination, we use the first two letters to identify the application's name. "ChFaTi" represents the combination of Chrome, Facebook, and TikTok.

Table 6: Sensitivity study with parameter *MS*.

MS	Total count	Cache count	Total size	Cache size
20MB	55%	83%	83%	91%
15MB	53%	77%	76%	94%
10MB	39%	72%	73%	84%
5MB	34%	76%	57%	91%

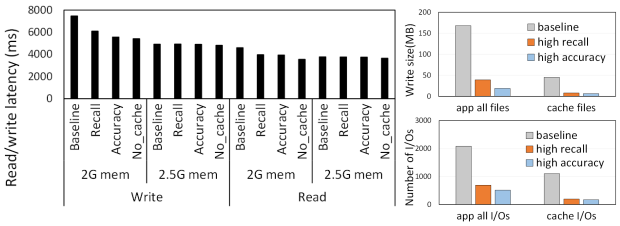
provement:

$$lifetime = \sum_{i=1}^{n} \frac{PEC_i \times (1 + OP_i)}{365 \times DWPD \times WA_i \times R_{Compress}} \quad (3)$$

In Equation 3, WA_i and OP_i are the write amplification and over provisioning factor for ECC_i, respectively, and PEC_i is the number of P/E cycles for which ECC_i is used. In our case, other parameters are constants, and thus the lifetime is inversely proportional to the number of full disk writes per day (DWPD) which depends on the amount of data written. Taking P9 as an example, we can reduce the amount of I/O by an average of 53.2% and 54.7%, respectively by the two models. Therefore, the lifetime can be improved by an average of 113.7% (1/(1-53.2%)-1) and 120.8% (1/(1-54.7%)-1), respectively.

6.2 Performance Improvement

Read/write performance improvement. Reduction in writebacks of cache files could improve read and write performance because of the reduction in I/O contention. To quantify the impact of writebacks of cache files on read and write performance, especially under intensive I/O, we assess the latency of running read/write micro-benchmarks when using a cache-intensive application, i.e., Facebook. Since most I/O sizes on mobile devices are in the size of 4KB [4], we sequentially write with fsync or read 512MB in size of 4KB by using the micro benchmarks to evaluate read/write performance. We scroll news on Facebook for five minutes and collect the latency of read and write in the default system (`Baseline`) and in the system with CacheSifter (`Recall` and `Accuracy`). `No_cache` represents the latency of read and write without using Facebook so that there is no interference of cache files generated by Facebook. We use memtester [34] to occupy physical memory, so that cache files will be written back quickly (to general the situation that memory is insufficient). To reduce bias, we conduct the experiment five times, and the average latency of the entire 512MB operation is presented in Figure 10a. To show more breakdown information, the I/O and writebacks produced by Facebook are presented in Figure 10b.



(a) Read/write performance. (b) I/O reduction.

Figure 10: The impact of CacheSifter on read and write performance under different memory pressure. The system could occupy approximately 2GB memory in this device.

In the baseline system, writebacks of cache files generated by Facebook degrade read and write performance by an average of 29.6% and 38%, respectively under memory pressure (with 2GB memory). Compared to `baseline`, the read and

write latency are reduced by an average of 13.9% and 18.4%, respectively by the high-recall model, while the numbers are 14.4% and 25.5%, respectively, when using the high-accuracy model. When there is sufficient memory (at least 2.5GB), the impact of cache files is marginal on read and write performance because few cache files will be written back to flash storage to generate I/O contentions with the read/write of the micro-benchmark. The performance improvement of CacheSifter derives from the write reduction of cache files. The benefit is significant under I/O intensive workloads or when memory is insufficient. According to paper [26], eight background applications are common. Memory pressure occurs frequently even in mobile devices with relatively large memory (8GB) as shown in Table 7.

Table 7: Free memory in mobile devices when running various numbers of applications.

Devices	Total memory	1 App	3 Apps	5 Apps	8 Apps	10 Apps
P9	3G	88M	80MB	90MB	82MB	80MB
Mate30	8G	1.8GB	1GB	680MB	167MB	95MB
Pixel6	8G	1.5GB	177MB	166MB	172MB	106MB

Impact of CacheSifter on framerate. Even though CacheSifter can improve read and write performance, re-accessing discarded cache files from networks can negatively impact user experience. We measure the possible loss in user experience with Frame Per Second (FPS) by PerfDog, a popular gaming benchmark [43]. Figure 11 shows the average FPS of Twitter. We choose Twitter as a foreground application, that is denoted as "F" because Twitter is another one of the most cache-intensive applications that could be relatively more affected by CacheSifter. There are various numbers of background applications, and "3B" means that there are three background applications. Background applications are randomly selected from the optimized ten applications.

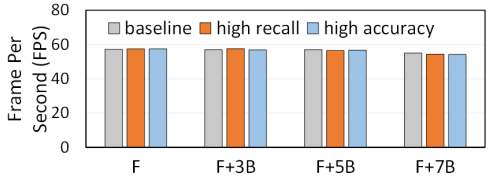


Figure 11: Impact of CacheSifter on application execution.

For the average FPS, the results in Figure 11 show that neither the high-recall model nor the high-accuracy model has a noticeable impact on FPS of the application execution. We also obtain the two important factors that impact FPS: CPU and peak memory. Table 8 lists the details of CPU usage and peak memory of Twitter. The results reveal that cache files being re-accessed by CacheSifter has a minimal impact on CPU usage and peak memory.

6.3 Overhead Analysis

Network overhead. Similar to the state changes shown in Table 2, there are six types of miscalssifications : "BR->TR,LL", "TR->BR,LL", and "LL->BR,TR". "BR-> TR, LL" means

Table 8: Information of the foreground application.

Factors	Methods	F	F+3B	F+5B	F+7B
Peak memory	baseline	334MB	323MB	302	304MB
	high recall	333MB	337MB	308MB	301MB
	high accuracy	343MB	328MB	315MB	323MB
CPU	baseline	9.9%	10%	8.9%	9%
	high recall	10%	10.1%	10.3%	10.7%
	high accuracy	10.7%	10.9%	10.5%	11.9%

that a BAR file is misclassified as a Transient file or a Long-living file. Notably, only three misclassifications, "TR->BR" and "LL->BR,TR", could induce re-download. Amount of these three misclassifications, the "LL->TR" case has small a possibility to be re-downloaded while other two cases have a large possibility to be re-downloaded. Based on this, we show the re-download upper bound and lower bound in Figure 12.

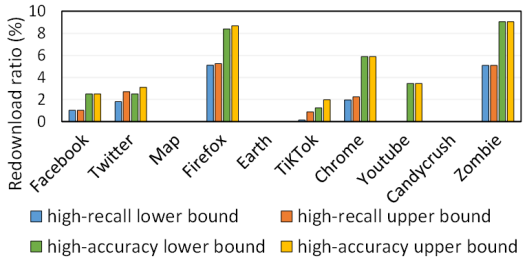


Figure 12: Re-download ratios of optimized applications.

The upper bound is equal to the total number (accurate and misclassified cases) divided by the sum of the number of these three cases. The lower bound is equal to the total number divided by the sum of the number of "LL,TR->BR" cases. The results show that the re-download penalty of high-recall mode is smaller than high-accuracy mode because of its goal (reducing re-download). This allows the operating system to deploy either high-recall or high-accuracy mode based on the users' network and data plan.

Memory overhead. Three components of CacheSifter introduce extra memory overhead: categorization, maintaining Transient files, and the ML inference. For categorization, cache files are maintained in the main memory until they are categorized. The extra memory usage depends on the size of Long-living files that are generated within 60 seconds because they are usually written back to the flash storage in default systems. The average memory overhead of this part is 492KB in our evaluations. The Transient files are stored in our QMFS with a maximum size of 20MB. When more than 10MB is occupied, a reclaim thread wakes up to free the memory, which ensures that the overhead stays below 10MB. Memory is used to run the machine learning method, specifically the inference step. Ten matrices for each model remain continually in the main memory for inference, which occupies approximately 2MB (0.89MB for the first model and 1.13MB for the second model). In summary, memory overhead is usually smaller than 12.5MB.

CPU time overhead. Training/retraining is conducted offline, and the overhead on smartphones is only the cost of categorization and eviction. We train a model for 20 applications by

using the data of 20 hours on a PC, and the training time is approximately one day. This training cost occurs only once over three months in our study. A dedicated thread wakes up periodically to conduct categorization and eviction. Categorization takes an average of 82ms out of 10s in our evaluation, as the matrices are relatively small. Moreover, the eviction scheme is used to shrink the in-memory file system. For this part, only the list move/insert operations are needed, and the overhead (1.9 ms out of 10s on average) is negligible. In summary, CPU time overhead is an average of 84ms for each iteration (10 seconds in our evaluation).

7 Related Works

Cache file optimization. User experience could be degraded due to too many cache files. Establish guidelines [10, 21] indicate that deleting the cache files of browsers can improve the overall performance of mobile devices. However, the deleted data must be re-downloaded from the network when users re-access them. This could degrade the performance, especially for the frequently-used data. Previous works [32, 38] show that keeping all of the cache files in the main memory can improve the performance because of their fast re-accessing. The benefits only occur when the cached files are accessed frequently. Otherwise, additional memory consumption may degrade the overall performance. Currently, the Android system and the existing works treat all cache files equally. Liang et al. [27] show that cache file vary greatly and should be managed differently but do not provide a corresponding solution.
Categorization of cache files. Caching files in memory is widely used to improve system performance. Korner et al. [22] firstly studied a knowledge-based remote file caching model and used multiple LRU lists to manage cache files on a server platform. Madhyastha et al. [29] employed a hidden Markov model to automatically classify file access patterns and tune the policies of the file system to improve global performance based on the observed patterns. In addition to a server platform, researchers introduced some cache file categorization schemes on mobile platforms. For example, Immanuel et al. [15] proposed a cache taxonomy that can decode several Android cache formats and display the contents in an accessible manner.
Eviction scheme. To our best knowledge, the eviction scheme used in CacheSifter is the first file-based eviction scheme to do so from within the kernel. Numerous page-based eviction schemes exist, which are usually designed based on the access locality of pages. The Linux firstly began to work on a page eviction mechanism from Kernel 2.4 [42], also termed page aging, which attempts to perform background scanning of the pages and use inactive lists to manage pages which are already idle. Liang et al. [25,28] proposed a size-tuning scheme which can reduce pre-fetched pages in order to avoid a high page cache eviction ratio.
Server caching. The cache not only exists in mobile devices but also on servers. As the information provider, the caching mechanism on the servers is different from that on mobile devices, which are contents consumer for the most of time. The consistency of the cache [33] on distributed servers is the main concern. The cache used on the servers is also designed to reduce latency of responding to clients [1, 36]. Mital et al. [31] proposed a framework to store files across multiple SBSs. Jiang et al. [19] introduced a new DRAM caching techniques based on filter caches, and also presented two filter caching techniques and specified when they should be employed. Meng et al. [30] designed a dynamic, self-adaptive framework, called vCacheShare, which automate server flash space for the cache in virtual environments.

8 Conclusion

Current mobile systems treat cache files equally, storing them in the main memory first and then writing them back into flash storage. Mobile device performance depends heavily on cache utilization, with the challenges of tackling variable file patterns and flash durability. This paper proposes a cache file management scheme, named CacheSifter, to sift cache files by a lightweight machine-learning-based categorization engine and manage them by a set of eviction schemes to shield flash from ephemeral cache data writes. CacheSifter is evaluated on two Android devices and over a collection of representative applications. Evaluation results demonstrate that CacheSifter can reduce writebacks of cache files by an average of 62% and 59.5%, by using different models, and the I/O intensive write performance of mobile devices is improved by an average of 18.4% and 25.5%. We conclude that CacheSifter provides significant benefits to both I/O performance and storage lifetime with marginal overhead.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd Prof. YouJip Won for their feedbacks and guidance. This paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (No.11204718) and National Natural Science Foundation of China (No. 61772092 and 61802038).

References

[1] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, I. Tsang, S. Gjessing, G. Fairhurst, C. Griwodz, and M. Welzl. Reducing internet latency: A survey of techniques and their merits. *IEEE Communications Surveys Tutorials*, 18(3):2149–2196, 2016.

[2] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu. Error characterization, mitigation, and recovery in flash-

memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[3] David Chu, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *ACM MobiSys*. ACM, June 2012.

[4] J. Courville and F. Chen. Understanding storage i/o behaviors of mobile applications. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, May 2016.

[5] Balázs Csanád Csáji et al. Approximation with artificial neural networks. 2001.

[6] Android Developers. Android systems delete cache files. https://developer.android.com/training/data-storage/app-specificjava, 2021.

[7] Engineers. Android debug bridge (adb) tool. https://androidmtk.com/download-minimal-adb-and-fastboot-tool, 2019.

[8] Congming Gao, Liang Shi, Mengying Zhao, Chun Jason Xue, Kaijie Wu, and Edwin H.-M. Sha. Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives. In *30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, 2014.

[9] Google. Android source tree. https://source.android.com/setup/build/downloading, 2020.

[10] Michelle Greenlee. How to clear the cache on your android phone to make it run faster. https://www.businessinsider.com/how-to-clear-cache-on-android-phone, 2019.

[11] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 759–771, Santa Clara, CA, July 2017.

[12] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. Fasttrack: Foreground app-aware i/o management for improving user experience of android smartphones. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 15–28, 2018.

[13] 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, July 2021.

[14] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. *ACM Transactions on Computer Systems (TOCS)*, 30(3):1–39, 2012.

[15] Felix Immanuel, Ben Martini, and Kim-Kwang Raymond Choo. Android cache taxonomy and forensic process. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 1094–1101. IEEE, 2015.

[16] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. Boosting quasi-asynchronous i/o for better responsiveness in mobile devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 191–202, 2015.

[17] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 61–74, Santa Clara, CA, February 2014.

[18] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/o stack optimization for smartphones. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 309–320, 2013.

[19] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. Chop: Adaptive filter-based dram caching for cmp server platforms. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[20] Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. Application-aware swapping for mobile systems. *ACM Trans. Embed. Comput. Syst.*, 16(5s):182:1–182:19, September 2017.

[21] Adrian Kingsley-Hughes. Hidden android tricks to speed up your smartphone. https://www.lifehacker.com.au/2019/11/android-smartphone-running-slow-try-deleting-the-app-cache/, 2019.

[22] Kim Korner. Intelligent caching for remote file service. In *Proceedings., 10th International Conference on Distributed Computing Systems*, pages 220–221. IEEE Computer Society, 1990.

[23] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 873–887, July 2020.

[24] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[25] Yu Liang, Yajuan Du, Chenchen Fu, Riwei Pan, Liang Shi, and Chun Jason Xue. Boosting read-ahead efficiency for improved user experience on mobile devices. *ACM SIGBED Review*, 16(3):75–80, 2019.

[26] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 897–910, July 2020.

[27] Yu Liang, Jinheng Li, Xianzhang Chen, Rachata Ausavarungnirun, Riwei Pan, Tei-Wei Kuo, and Chun Jason Xue. Differentiating cache files for fine-grain management to improve mobile performance and lifetime. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, July 2020.

[28] Yu Liang, Riwei Pan, Yajuan Du, Chenchen Fu, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Read-ahead efficiency on mobile devices: Observation, characterization, and optimization. *IEEE Transactions on Computers*, 2020.

[29] Tara M Madhyastha and Daniel A Reed. Exploiting global input output access pattern classification. In *SC'97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pages 9–9. IEEE, 1997.

[30] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. vcacheshare: Automated server flash cache space management in a virtualization environment. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 133–144, Philadelphia, PA, June 2014.

[31] N. Mital, D. Gündüz, and C. Ling. Coded caching in a multi-server system with random topology. *IEEE Transactions on Communications*, 68(8):4620–4631, 2020.

[32] Ngoan Nguyn. Ram disk: an app to mount a folder directly into the ram. https://apkpure.com/ramdisk/com.yz.ramdisk, 2019.

[33] Pei Cao and Chengjie Liu. Maintaining strong cache consistency in the world wide web. *IEEE Transactions on Computers*, 47(4):445–457, 1998.

[34] pyropus technology. Memory test tool memtester. http://pyropus.ca/software/memtester/, 2017.

[35] 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. USENIX Association, February 2021.

[36] S. P. Shariatpanahi, S. A. Motahari, and B. H. Khalaj. Multi-server coded caching. *IEEE Transactions on Information Theory*, 62(12):7253–7271, 2016.

[37] Z. Shen, L. Han, R. Chen, C. Ma, Z. Jia, and Z. Shao. An efficient directory entry lookup cache with prefix-awareness for mobile devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2020.

[38] SoftPerfect. How to improve your computer performance and ssd life span with a ram disk. https://www.softperfect.com/articles/how-to-boost-computer-performance-with-ramdisk/, 2020.

[39] Statista. Number of smartphone users worldwide from 2016 to 2021. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, 2020.

[40] Statista. Replacement cycle length of smartphones worldwide. https://www.statista.com/statistics/786876/replacement-cycle-length-of-smartphones-worldwide/, 2020.

[41] Linus Torvalds and thousands of collaborators. The linux kernel archives. https://www.kernel.org/, 2020.

[42] Rik Van Riel. Page replacement in linux 2.4 memory management. 2001.

[43] Wetest. Fps test tool perfdog. https://perfdog.wetest.net/, 2020.

[44] Sangjin Yoo and Dongkun Shin. Reinforcement learning-based SLC cache technique for enhancing SSD write performance. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, July 2020.

[45] Tao Zhang, Aviad Zuck, Donald E. Porter, and Dan Tsafrir. Apps can quickly destroy your mobile's flash - why they don't, and how to keep it that way (poster). *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, page 207–221, 2019.