usenix
ASSOCIATION

Proceedings of the 18th USENIX Conference on File and Storage Technologies

Santa Clara, CA, USA   February 25–27, 2020

18th USENIX Conference on File and Storage Technologies

Santa Clara, CA, USA
February 25–27, 2020

Sponsored by

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

In cooperation with ACM SIGOPS

FAST '20 Sponsors

Gold Sponsors

NetApp

NSF

Silver Sponsors

amazon

Radian MEMORY SYSTEMS

vmware

Bronze Sponsors

FACEBOOK

FUTUREWEI Technologies

IBM

ORACLE

SAMSUNG

SNIA

Open Access Sponsor

NetApp

Industry Partners
DMTF
FreeBSD Foundation

Media Sponsor
ACM *Queue*

USENIX Supporters

USENIX Patrons

Bloomberg • Facebook • Google
Microsoft • NetApp

USENIX Benefactors

Amazon • Oracle
Thinkst Canary • Two Sigma • VMware

USENIX Partners

ProPrivacy • Restore Privacy • Top10VPN

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the
18th USENIX Conference on File and Storage
Technologies

February 24–27, 2020
Santa Clara, CA, USA

Conference Organizers

Program Co-Chairs

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

Brent Welch, *Google*

Program Committee

Nitin Agrawal, *ThoughtSpot*

George Amvrosiadis, *Carnegie Mellon University*

John Bent, *Seagate*

Pramod Bhatotia, *The University of Edinburgh*

Suparna Bhattacharya, *Hewlett Packard Enterprise*

William J. Bolosky, *Microsoft Research*

André Brinkmann, *Johannes Gutenberg-University Mainz*

Randal Burns, *Johns Hopkins University*

Ali Butt, *l Tech*

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

Angela Demke Brown, *University of Toronto*

Gary Grider, *Los Alamos National Laboratory*

Haryadi Gunawi, *University of Chicago*

Dean Hildebrand, *Google*

Yu Hua, *Huazhong University of Science and Technology*

H. Howie Huang, *The George Washington University*

Jian Huang, *University of Illinois at Urbana–Champaign*

Jooyoung Hwang, *Samsung Electronics*

Bill Jannen, *Williams College*

Kimberly Keeton, *HP Labs*

Geoff Kuenning, *Harvey Mudd College*

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

Sungjin Lee, *DGIST (Daegu Gyeongbuk Institute of Science and Technology)*

Darrell Long, *University of California, Santa Cruz*

Xiaosong Ma, *Qatar Computing Research Institute*

Umesh Maheshwari, *Hewlett Packard Enterprise*

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

Changwoo Min, *Virginia Tech*

Kiran-Kumar Muniswamy-Reddy, *Amazon*

Dalit Naor, *IBM Research*

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

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

Rob Ross, *Argonne National Laboratory*

Keith A. Smith, *MongoDB*

Vasily Tarasov, *IBM Research*

Devesh Tiwari, *Northeastern University*

Carl Waldspurger, *Carl Waldspurger Consulting*

Brent Welch, *Google*

Ric Wheeler, *Facebook*

Avani Wildani, *Emory University*

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

Gala Yadgar, *Technion—Israel Institute of Technology*

Jishen Zhao, *University of California, San Diego*

Poster Session Chair

Dean Hildebrand, *Google*

Work-in-Progress Reports (WiPs) Chair

Avani Wildani, *Emory University*

Test of Time Awards Committee

Jiri Schindler, *Tranquil Data*

Bianca Schroeder, *University of Toronto*

Tutorial Coordinators

Andy Klosterman, *NetApp*

John Strunk, *Red Hat*

Steering Committee

Nitin Agrawal, *ThoughtSpot*

Angela Demke Brown, *University of Toronto*

Greg Ganger, *Carnegie Mellon University*

Casey Henderson, *USENIX Association*

Kimberly Keeton, *HP Labs*

Geoff Kuenning, *Harvey Mudd College*

Arif Merchant, *Google*

Florentina Popovici, *Google*

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*

Ric Wheeler, *Facebook*

Erez Zadok, *Stony Brook University*

External Reviewers

Kang Chen	Ke Yang	Pradeep Kumar	Tao Zhang
Guanyu Feng	Guangyan Zhang	Ankush Jain	Jaehun Han
Cheng Li	Xiaowei Zhu	Michael Kuchnik	Simon Bertron

Message from the
FAST '20 Program Co-Chairs

Welcome to the 18th USENIX Conference on File and Storage Technologies, FAST '20. 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. We are pleased to present a diverse set of papers on topics such as cloud storage, key-value stores, consistency, reliability, caching, HPC systems, SSD, and traditional file systems. Submissions to the conference came from authors representing academia, industry, and the open source community.

FAST '20 received 138 submissions. Of these, we accepted 23 papers, for an acceptance rate of 17%. The Program Committee used a two-round online review process and then met in person to select the final program. In the first round, each paper received at least three reviews. For the second round, 79 papers received at least three more reviews. The Program Committee discussed 61 papers in an all-day meeting on December 6, 2019, at Google in Sunnyvale, CA. We used Eddie Kohler's excellent HotCRP software to manage all stages of the review process, from submission to author notification.

As in the previous years, we included a category of short papers. Short papers provide a vehicle for presenting completed research that does not require a full-length paper to describe and evaluate. We received 25 short paper submissions, of which 2 were accepted. Also in line with previous years, we included a category of deployed-systems papers, which address experience with the practical design, implementation, analysis or deployment of large-scale, operational systems. We received 6 deployed-systems submissions, of which we accepted 3.

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 '20. We would also like to thank the attendees of FAST '20 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, especially Casey Henderson, Jasmine Murcia, Sarah TerHune, Camille Mulligan, Olivia Vernetti, and Arnold Gatilao, 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 Poster and Work-in-Progress Session Chairs, Dean Hildebrand and Avani Wildani. 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. Some of the PC traveled halfway across the world for the one-day, in-person PC meeting. In total, the PC wrote 653 thoughtful and meticulous reviews. HotCRP recorded approximately 473,000 words in reviews and comments (excluding HotCRP boilerplate language). The reviewers' evaluations, and their thorough and conscientious deliberations at the PC meeting, contributed significantly to the quality of our decisions. Each paper had a shepherd that reviewed the final submission and provided additional feedback. In many cases this led to significant improvements in the final quality of the submissions. We look forward to an interesting and enjoyable conference!

Brent Welch, *Google*
Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology)*
FAST '20 Program Co-Chairs

FAST '20: 18th USENIX Conference on File and Storage Technologies February 25–27, 2020
Boston, MA, USA

Cloud Storage

File Systems

HPC Storage

SSD and Reliability

Performance

Key Value Storage

Caching

Consistency and Reliability

MAPX: Controlled Data Migration in the Expansion of Decentralized Object-Based Storage Systems

Li Wang
laurence.liwang@gmail.com
Didi Chuxing

Yiming Zhang
sdiris@gmail.com (Corresponding)
NiceX Lab, NUDT

Jiawei Xu
titan_xjw@cs.sjtu.edu.cn
SJTU

Guangtao Xue
xue-gt@cs.sjtu.edu.cn
SJTU

Abstract

Data placement is critical for the scalability of decentralized object-based storage systems. The state-of-the-art CRUSH placement method is a decentralized algorithm that deterministically places object replicas onto storage devices without relying on a central directory. While enjoying the benefits of decentralization such as high scalability, robustness, and performance, CRUSH-based storage systems suffer from *uncontrolled* data migration when expanding the clusters, which will cause significant performance degradation when the expansion is nontrivial.

This paper presents MAPX, a novel extension to CRUSH that uses an extra time-dimension mapping (from object creation times to cluster expansion times) for controlled data migration in cluster expansions. Each expansion is viewed as a new layer of the CRUSH map represented by a virtual node beneath the CRUSH root. MAPX controls the mapping from objects onto layers by manipulating the timestamps of the intermediate placement groups (PGs). MAPX is applicable to a large variety of object-based storage scenarios where object timestamps can be maintained as higher-level metadata. For example, we apply MAPX to Ceph-RBD by extending the RBD metadata structure to maintain and retrieve approximate object creation times at the granularity of expansions layers. Experimental results show that the MAPX-based migration-free system outperforms the CRUSH-based system (which is busy in migrating objects after expansions) by up to $4.25\times$ in the tail latency.

1 Introduction

Object-based storage systems have been widely used for various scenarios such as distributed file storage, remote block storage, small object (e.g., profile pictures) storage, blob (e.g., large videos) storage, etc. Compared to filesystem-based storage, object-based storage simplifies data layout by exposing an interface for reading and writing objects via unique object names, and thus reduces management complexity at the backend.

Objects are distributed among a large number of object storage devices (OSDs) possibly with various capacities and characteristics, making data placement critical for the scalability of object-based systems. Decentralized placement methods uniformly distribute objects among OSDs without relying on a central directory, and usually outperform centralized methods because their clients could directly access objects by calculating (instead of retrieving) the responsible OSDs. CRUSH [67] is the state-of-the-art placement algorithm that allows structured mapping from objects onto a hierarchical cluster map comprising nodes representing OSDs, machines, racks, etc. Currently, CRUSH has been widely adopted in large-scale storage systems (like Ceph [66] and Ursa [44]) owing to its simplicity and generality.

While enjoying the benefits of decentralization such as high scalability, robustness, and performance, CRUSH-based storage systems suffer from uncontrolled data migration after expanding the clusters and/or adding more intermediate placement groups (PGs). Although the migration could re-balance the load of the entire system right after the expansion, it also causes significant performance degradation when the expansion is nontrivial (e.g., adding several racks of storage machines).

In practical deployment of distributed storage systems, it is preferred to avoid large-scale data migration after cluster expansions [15], even at the cost of temporary load imbalance. Ceph [66] is a CRUSH-based object storage system which mitigates CRUSH's migration problem via implementation-level optimizations. It limits the migration rate to a relatively-low level, performing writes to the old OSDs if the written object is waiting for migration. However, all object replicas will be *eventually* migrated to the target OSDs calculated by the CRUSH algorithm, making Ceph experience degraded performance for a long period of time.

In contrast, traditional centralized placement methods could easily control data migration for cluster expansions. For example, Haystack [15] and HDFS [9] maintain a central directory recording object positions, so as to keep existing objects unaffected during expansions and place only new

objects onto the newly-added OSDs.

In this paper we present MAPX, a novel extension to CRUSH that uses an extra dimensional mapping (from object creation times to cluster expansion times) for controllable data migration in the expansion of decentralized object-based storage systems. Each expansion is viewed as a new layer of the CRUSH map represented by a virtual node beneath the CRUSH root. MAPX controls the mapping from objects onto layers by manipulating the timestamps of the intermediate PGs.

The time-dimension mapping cannot support *general* object storage where the maintenance overhead of per-object timestamps might be overwhelming. However, MAPX is applicable to a large variety of object-based storage scenarios (such as block storage and file storage), where the object creation timestamps can be maintained as higher-level storage metadata. We apply MAPX to Ceph-RBD (Reliable-autonomic-distributed-object-store Block Device) [3] and CephFS (Ceph File System) [4] with minimum modifications to the original CRUSH algorithm in Ceph (Luminous) [5]. For Ceph-RBD, we extend the *rbd_header* metadata structure to maintain and retrieve approximate object creation times at the granularity of expansion layers; while for CephFS, we extend the *inode* metadata structure to take the files' creation times, which could also be maintained at the granularity of layers, as the creation times of the files' objects. More complex applications of MAPX could be built based on block storage (Ceph-RBD) or file storage (CephFS). Experimental results show that the MAPX-based migration-free system outperforms the CRUSH-based system (which is busy in migrating objects after expansions) by up to $4.25\times$ in the tail latency.

The rest of this paper is organized as follows. Section 2 introduces the background and problem of CRUSH. Section 3 presents the design of MAPX. Section 4 evaluates the performance of MAPX and compares it with CRUSH. Section 5 introduces related work. And finally Section 6 concludes the paper and discusses future work.

2 Background

2.1 CRUSH Overview

CRUSH uses a logical cluster map to abstract the storage cluster's hierarchical structure. Fig. 1 illustrates a three-level storage hierarchy, where the entire cluster (root) is composed of cabinets (representing racks), which are filled with shelves (representing storage machines) each installing many OSDs (disks). The internal nodes (root, cabinet, and shelf) in the hierarchy are referred to as buckets (the types of which are *straw2* throughout this paper as discussed in detail in Section 5.1). The hierarchy is flexible for extension. For example, cabinets might be further grouped into "row" buckets for larger clusters.
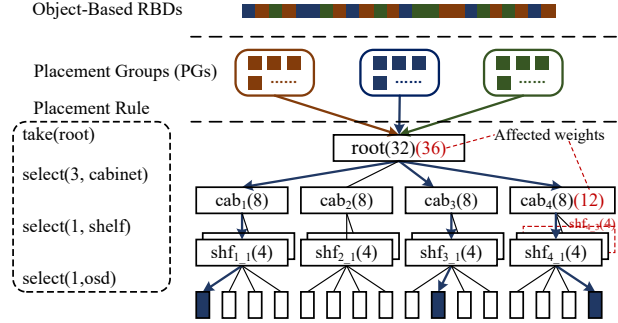
Figure 1: Example of CRUSH placement algorithm. An RBD is mapped to a PG which is subsequently mapped to a list of OSDs. The second operation (select(3,cabinet)) realizes three-way replication with three different cabinets. For simplicity each leaf OSD has the same weight of one.

Each OSD has a *weight* assigned by the administrator to control the OSD's relative amount of stored data, so that the load of an OSD is on average proportional to its weight. The weight of an internal bucket is (recursively) calculated as the sum of the weights of its child items. There are mainly two steps for CRUSH to place object replicas onto OSDs, which are briefly introduced below and will be discussed in more details in Section 5.1.

First, the objects are categorized into PGs by computing the modulo of the hashing of object names, i.e., *pgid* = HASH(*name*) mod PG_NUM. Second, the objects in a PG are mapped to a list of OSDs following the CRUSH algorithm. The first step is similar to traditional hashing and in the rest of this section we will briefly introduce the second step.

The CRUSH algorithm supports flexible constraints for reliable replica placement by (i) encoding the information of failure domains (like shared power source or network) into the cluster map, and (ii) letting the administrator define the *placement rules* that specify how replicas are placed by recursively selecting bucket items.

Fig. 1 demonstrates a typical placement procedure of CRUSH (for the dark blue PG) beginning at the root, where the values in the buckets' parentheses represent the weights. The first operation (take(root)) of the rule selects the root of the storage hierarchy and uses it as an input to subsequent operations. The second operation (select(3,cabinet)) repeatedly computes the following Eq. (1) to choose $x = 3$ items (cabinets at this level) for three-way replication, from totally $|\vec{i}| = 4$ items $\in \vec{i}$ beneath the root:

$$C(pgid, \vec{i}, r) = \underset{i \in \vec{i}}{\arg\max} HASH(pgid, r, ID(i)) \times W(i), \quad (1)$$

where *pgid* is the ID of the input PG, $r = 1, 2, \cdots$ is a parameter for the argmax computation, *HASH* is a three-input hash function, and $ID(i)$ and $W(i)$ are the ID and weight of an item $i \in \vec{i}$, respectively. To choose x distinct items, it is
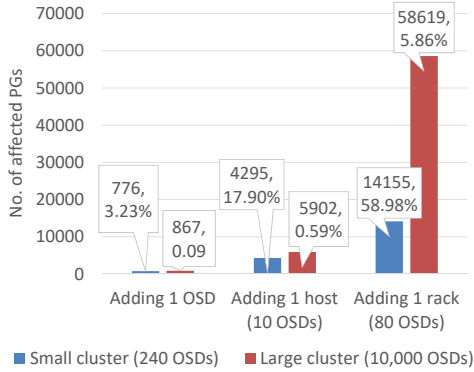
Figure 2: Data migration of two simulated CRUSH clusters during expansions.

possible to perform Eq. (1) more than x times because the output of Eq. (1) may have already been chosen in previous computation or the chosen item may be failed/overloaded.

Similarly, the subsequent operations (`select(1,shelf)` and `select(1,osd)`) follow Eq. (1) to choose $x = 1$ shelf and OSD beneath each of the three cabinets. The final result of the placement rule is the three darkblue OSDs in Fig. 1.

2.2 The Main Drawback of CRUSH

CRUSH achieves statistical load balancing without a central directory, and could automatically re-balance the load when the storage cluster map changes. On the downside, however, it also causes *uncontrollable* data migration in cluster expansions. For instance, adding a new shelf (shf_{4_3}) with 4 OSDs beneath a cabinet (cab_4) in Fig. 1 will affect the weights (labeled in the second red parentheses) of all items along the path from the newly-added shelf up to the root, and thus will lead to data movement not only from other shelves in cab_4 to the newly-added shf_{4_3} but also from other cabinets to cab_4. The amount of data migration can be as high as $h\frac{\Delta w}{W}$ if Δw is small relative to W [67], where h is the number of levels in the hierarchy, and Δw and W are the increased weight of the expansion and the total weight of all OSDs, respectively.

To demonstrate the severity of the problem, we measure the amount of data movement in two simulated CRUSH-based three-level Ceph clusters, which adopt three-way replication taking a rack as a failure domain. One rack consists of 8 hosts each containing 10 OSDs. The first small cluster has a total of 3 racks, 24 hosts, and 240 OSDs, and stores 24,000 PGs; while the second large cluster has 125 racks, 1000 hosts, and 10,000 OSDs, and stores 1,000,000 PGs. We respectively add one OSD, one machine, and one rack to the two clusters. The result (Fig. 2) shows that the migration is significant when the expansion is nontrivial, e.g., almost 60% of the PGs will be affected when adding one rack to the small cluster, which will inevitably cause performance degradation during the entire migration period.
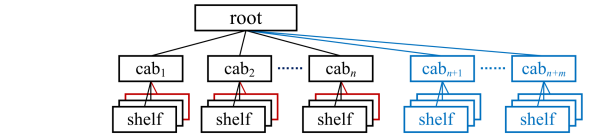
(a) The composite cluster map after two expansions
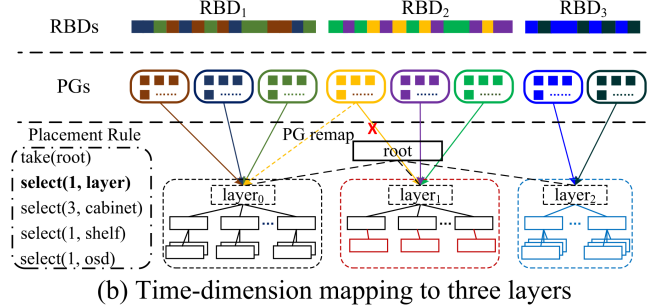
(b) Time-dimension mapping to three layers

Figure 3: MAPX records each expansion as a layer. MAPX implicitly adds a *select* operation (`select(1,layer)`) to the placement rule.

3 MAPX Design

Compared to moderate load imbalance, large-scale data migration often has much more negative impact on I/O performance in the expansion of distributed storage systems. The CRUSH placement algorithm suffers from data migration after each cluster expansion because it "crushes" the differences between the new and the old objects/OSDs. To address this problem, MAPX extends the original CRUSH algorithm with an extra time-dimension mapping.

3.1 Migration-Free Expansion

Storage systems usually prefer to avoid data migration after cluster expansion even at the cost of temporary load imbalance. For instance, Haystack and HDFS leverage a central directory to keep existing objects unaffected during cluster expansions. As new objects are stored onto the new OSDs, the available capacity of them decreases over time and thus eventually the entire system will achieve approximate load balancing. Data migration can be performed (with metadata modification) at any time as needed.

Inspired by the centralized placement methods, our goal is to achieve controlled data migration for cluster expansions. To achieve this, we design MAPX on top of CRUSH by introducing an extra time-dimension mapping to distinguish the new and the old objects/OSDs, while still preserving the benefits of randomness and uniformness of CRUSH.

Fig. 3(a) depicts an example of two expansions to the original cluster which consists of n cabinets each having two shelves. The first expansion adds a shelf (represented by a red rectangle) to each of the n cabinets and the second expansion adds m cabinets (represented by blue rectangles).

Algorithm 1 Extended `select` Procedure of MAPX

```
 1: procedure SELECT(number, type)
 2:     if type ≠ "layer" then
 3:         return CRUSH_SELECT(number, type)
 4:     end if
 5:     layers ← layers beneath currently-processing bucket
            ▷ each layer represents an expansion
 6:     num_layers ← number of layers in layers
 7:     pg ← current Placement Group
 8:     ⃗o ← Φ                              ▷ output list
 9:     for (i = num_layers − 1; i ≥ 0; i − −) do
10:         layer ← layers[i]
11:         if layer.timestamp ≤ pg.timestamp then
12:             if layer was chosen by previous select then
13:                 continue
14:             end if
15:             ⃗o ← ⃗o + {layer}
16:             number ← number − 1
17:             if number == 0 then
18:                 break
19:             end if
20:         end if
21:     end for
22:     return ⃗o
23: end procedure
```

Unlike CRUSH which *monolithically* updates the cluster map, MAPX views each expansion, as well as the original cluster, as a separate *layer* which contains not only the new leaf OSDs but also all the internal buckets (shelves, cabinets, etc.) from the leaf OSDs up to the root.

To support the time-dimension mapping with minimum modifications to CRUSH, we insert a virtual level beneath the common CRUSH root (Fig. 3(b)), where each virtual node represents a layer of expansion. The virtual level enables MAPX to realize migration-free expansion by mapping new objects to the new layer before further processing of the CRUSH algorithm. Since the new layer will not affect the weights of the old ones, the placement of old objects within old layers will not change.

Mapping objects to PGs. In each expansion, the new layer is assigned with a certain number of newly-created PGs each having a timestamp (t_{pgs}) equal to the layer's expansion time (t_l). When writing/reading an object O (with creation timestamp t_o), we first compute the ID (*pgid*) of O's PG by

$$pgid = Hash(name) \bmod \text{INIT\_PG\_NUM}[j]$$
$$+ \sum_{i=0}^{j-1} \text{INIT\_PG\_NUM}[i], \quad (2)$$

where *name* is the object name, INIT_PG_NUM[i] is the initial number of PGs of the ith layer, and the jth layer has the latest timestamp $t_l ≤ t_o$ among all layers. Note that although

PGs might be remapped to other layers for, e.g., load rebalancing (Section 3.2), INIT_PG_NUM is a layer's constant and thus the mapping from objects to PGs is *immutable*. Consequently, each object is mapped to a responsible PG during creation, which has the latest timestamp $t_{pgs} ≤ t_o$ among all PGs. For instance, suppose that the three RBD_1, RBD_2, and RBD_3 in Fig. 3(b) are created respectively after the expansions of $layer_0$, $layer_1$, and $layer_2$. The objects of RBD_1, RBD_2, and RBD_3 will use the three layers' INIT_PG_NUM to calculate their PGs respectively within $layer_0$, $layer_1$, and $layer_2$.

Mapping PGs to OSDs. Similar to CRUSH, MAPX maps a PG onto a list of OSDs following a sequence of operations in a user-defined placement rule. As shown in Fig. 3(b), MAPX implicitly adds a *select* operation (`select(1,layer)`) to the placement rule, so as to realize the time-dimension mapping from PGs to layers without disturbing the administrators. Internally, MAPX extends CRUSH's original *select* operation to support the *layer*-type `select()`, as shown in Algorithm 1. If *type* is not "layer", then the processing is the same as the original CRUSH (Lines 2 ~ 4). Otherwise, we initialize an array of *layers* which stores all layers beneath the currently-processing bucket (usually the root) in an ascending order of the layers' timestamps (Line 5). We also initialize *num_layers* (the number of layers), *pg* (the placement group), and ⃗o (the output list) at Lines 6 ~ 8. Then the loop (Lines 9 ~ 21) adds *number* layers in the array of *layers* to the output list ⃗o. In most cases *number* = 1 so that the PG could be mapped to OSDs in one layer, but it is also possible to specify a larger *number* for, e.g., mirroring between two layers of expansions.

Note that the replicas of an object are not necessarily all placed on the newest layer. For example, suppose that the last expansion ($layer_2$) adds only two cabinets in Fig. 3(a) (i.e., $m = 2$) but the second `select()` function (`Select(3, cabinet)`) requires three cabinets. This will cause the first `select()` function (`select(1, layer)`) to be invoked twice to satisfy the rules following the backtracking mechanism of CRUSH: when a `select()` function cannot select enough items beneath a "layer" bucket, MAPX will retain (rather than abandon) the selected items and backtrack to the root to select the lacking items beneath a previous layer. Lines 12 ~ 14 check whether *layer* has been chosen by previous `select()` and if so we continue to the next loop, so as to avoid duplicate layer selection when performing backtracking. The double check ensures Algorithm 1 to correctly handle this situation, respectively returning $layer_2$ and $layer_1$ for the first and second `select()` functions.

3.2 Migration Control

The MAPX-based migration-free placement algorithm provides (statistical) load balancing within each layer, owing to the randomness and uniformness of the original CRUSH

algorithm, and achieves approximate load balancing among different layers by timely expanding the cluster when the load of the current layer increases to the same level as previous layers.

However, the load of a layer might change because of, e.g., removals of objects, failures of OSDs, or unpredictable workload changes. In Fig. 3, for example, it is possible that the cluster performs the second expansion ($layer_2$) when the load of the first expansion ($layer_1$) is as high as that of the original cluster ($layer_0$), but afterwards a large number of objects of $layer_1$ are removed and consequently the loads of the first two layers may get imbalanced.

To address the potential load imbalance problem, we design three flexible strategies for dynamically managing the load in MAPX, namely, placement group remapping, cluster shrinking, and layer merging.

PG remapping. MAPX supports to control object data migration by dynamically remapping the PGs. Each PG has two timestamps, namely, a static timestamp (t_{pgs}) that is equal to the expansion time of the PG's initial layer, and a dynamic timestamp (t_{pgd}) that could be set to any layer's expansion time. Different from the mapping from objects to PGs which uses static timestamps (Section 3.1), the mapping from PGs to layers is performed by comparing the PGs' dynamic timestamps to the layers' timestamps (Line 11 in Algorithm 1). Consequently, a PG can be easily remapped to any layer by manipulating the dynamic timestamp (as illustrated in Fig. 3(b)), which will be notified to all OSDs and clients via incremental map updates. The storage overhead for PGs' timestamps is moderate. For example, if we use a one-byte index for each PG timestamp (pointing to the corresponding layer's timestamp) which supports a maximum of $2^8 = 256$ layers), and suppose that one machine has 20 OSDs each responsible for 200 PGs, then the memory overhead of timestamps for a 1000-machine cluster is $1000 \times 20 \times 200 \times 2 \times 1B = 8MB$.

Cluster shrinking. When the load of a layer becomes lower than a threshold, MAPX shrinks the cluster by removing the layer's devices (such as OSDs, machines, and racks) from the cluster, as an inverse operation of cluster expansions. Given a layer Ω to be removed from the cluster, we first assign all PGs in Ω to the remaining layers according to their aggregated weights (for simplicity the reassignment does not consider the actual loads of the layers), and then migrate the PGs to the target layers through remapping (as discussed above). After shrinking the layer Ω is logically preserved (with no physical devices or PGs) and its `INIT_PG_NUM` will not change, so as not to affect the mapping from objects to PGs (following Eq. (2)).

Layer merging. MAPX balances the loads of two layers (Ω and Ω') via layer merging, which could be easily realized by setting the expansion time of one layer (Ω') to be the same as that of the other (Ω).

3.3 Implementing MAPX in Ceph

We have implemented the MAPX structure in Ceph by augmenting the original CRUSH algorithm with an extra time-dimension mapping. As shown in Fig. 3(b), the internal buckets (like shelves, cabinets, and rows, but not leaf OSDs) may belong to multiple layers. Therefore, we assign an internal device in a particular layer (i.e., beneath a particular virtual node) with a virtual device ID by concatenating the physical device ID and the layer's timestamp. We use the weight fields of the virtual nodes to record the layers' timestamps, which will be compared with the PGs' dynamic timestamps for layer selection.

MAPX is not suitable for general object stores, mainly because it is nontrivial to maintain and retrieve the timestamps of arbitrary objects. The overhead of per-object timestamp maintenance is similar to that of the maintenance of a central directory, and thus should be avoided in decentralized placement methods like CRUSH and MAPX. However, MAPX is applicable to a large variety of object-based storage systems such as block storage (Ceph-RBD [3]) and file storage (Ceph-FS [4]), where the object timestamps can be maintained as higher-level metadata.

Ceph-RBD. We have implemented the metadata-based time-stamp retrieval mechanism for Ceph-RBD (RADOS Block Device). Ceph stores the metadata (such as the prefix of data object names, and the information of volume, snapshot, striping, etc.) of an RBD in its *rbd_header* structure, which will be retrieved when a client mounts the RBD via *rbd_open*. Since an object of an RBD can be created after any expansions, we inherit the timestamp of the current layer (when an object is created) as the object's timestamp. Therefore, we add a per-object index (named *object_timestamp*) to the *rbd_header* structure which points to each layer's expansion time. The storage overhead for the extra metadata is moderate. For example, if we use one byte for the per-object index and each object is 4MB, then the storage overhead of the *object_timestamp* array for a 4TB RBD is at most $\frac{4TB}{4MB} \times 1B = 1MB$.

CephFS. We have also (partially) implemented the time-stamp retrieval mechanism for CephFS (Ceph Filesystem). Ceph stores the file metadata (including file creation times) in the *inode* structure. A client reads *inode* when opening a file and gets the file creation time. Currently we let all the objects of a file inherit the file's timestamp, so that we could control the time-dimension mapping at the granularity of files. We also plan to support finer-grained object timestamp maintenance. If the size of a file exceeds a threshold T (e.g., $T = 100$ MB), we could divide it into subfiles each smaller than 100 MB. The file's metadata maintains both the mapping from the file to its sub-files and the creation timestamp of each subfile, so that we could control the time-dimension mapping at the granularity of subfiles.
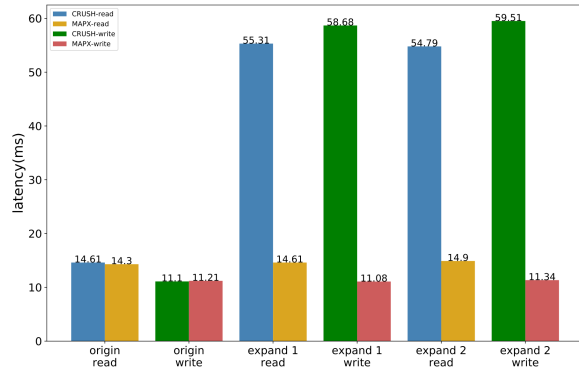
Figure 4: 99th percentile I/O latency of MAPX and CRUSH (during cluster expansions).

Figure 5: IOPS of MAPX and CRUSH (during cluster expansions).

4 Evaluation

In this section we evaluate the performance of the MAPX-based Ceph and compare it with that of the original CRUSH-based Ceph. Our testbed consists of four machines, of which three machines run the Ceph OSD storage servers and the other machine runs the client. Each machine has dual 20-core Xeon E5-2630 2.20GHz CPU, 128GB RAM, and one 10GbE NIC, running CentOS 7.0. Each storage machine, installs four 5.5TB HDDs, and runs Ceph 12.2 (Luminous) with the BlueStore backend. In all experiments every storage machine is viewed as a failure domain. The Ceph monitor is co-located with one of the storage servers. The client runs the `fio` benchmark.

4.1 I/O Performance during Expansions

We compare the I/O performance of MAPX and CRUSH during expansions, respectively being used as the object placement methods for Ceph.

We use the default values of all parameters of Ceph except $OSD\_max\_backfills$. As discussed in Section 1, Ceph mitigates the migration problem of CRUSH via implementation-level optimizations. It uses the parameter $OSD\_max\_backfills \geq 1$ to trade off between the severity and duration of performance degradation caused by data migration.

By default Ceph sets the parameter $OSD\_max\_backfills = 1$, which makes migration have the lowest priority so that objects in PGs could be migrated with an extremely-low speed. Although partially mitigating the degradation problem, setting $OSD\_max\_backfills = 1$ will significantly extend the migration period and largely increase the write load before the migration completes: writes to a PG waiting for migration will first be performed to the origin OSD and then be asynchronously migrated to the target OSD. Clearly, this makes Ceph experience *less severe* performance degradation but for a *longer* period of time. We set
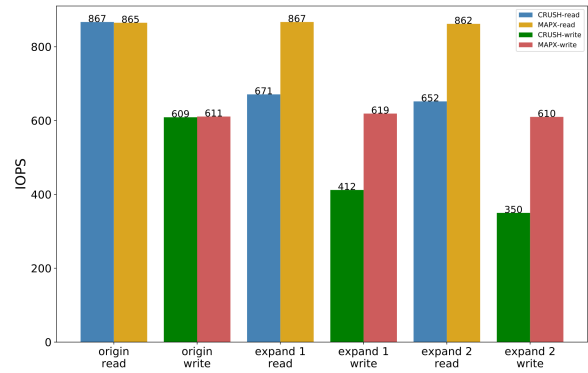
$OSD\_max\_backfills = 10$, which is more reasonable in this experiment so that migration could get a higher priority to demonstrate the algorithm-level difference between MAPX and CRUSH. We will discuss more on the impact of migration priority in Section 5.2.

The initial Ceph cluster has three storage machines each of which has two OSDs. We create 128 PGs, and the three-way replication results in (on average) $128 \times 3 \div 3 \div 2 = 64$ PGs for which each OSD will be responsible. We create 40 RBD images (each with 20GB data) in the initial cluster. We expand the storage cluster by respectively adding one and two OSDs to each machine in the cluster. We evaluate the performance (including I/O latency and IOPS) of Ceph running the migration-free MAPX, and compare it with the performance of Ceph running the original CRUSH algorithm. The I/O size is 4KB. The iodepth is 1 and 128 in the latency and IOPS tests, respectively.

Fig. 4 shows the evaluation result for the 99th percentile tail latencies. Note that cloud storage scenarios usually care about the (99th, 99.9th, or 99.99th percentile) tail latency rather than the mean or median latency, so as to guarantee SLA. MAPX outperforms CRUSH by up to 4.25×, mainly because the migration in CRUSH severely contends with the normal I/O requests. In this experiment, MAPX always uses six OSDs of the initial cluster to serve I/O requests because it does not migrate existing RBDs to the new OSDs. In contrast, CRUSH respectively uses six, nine, and twelve OSDs, but the CRUSH-induced data migration severely degrades the performance, which is unacceptable for latency-sensitive applications.

Fig 5 shows the evaluation result for IOPS respectively in MAPX and CRUSH. Each result is the mean of 20 runs, and we omit the error bars because the variances to the mean are relatively small (less than 5%). Similar to the latency test, MAPX significantly outperforms CRUSH by up to 74.3% in the IOPS test, because CRUSH's data migration contends with the normal I/O requests.
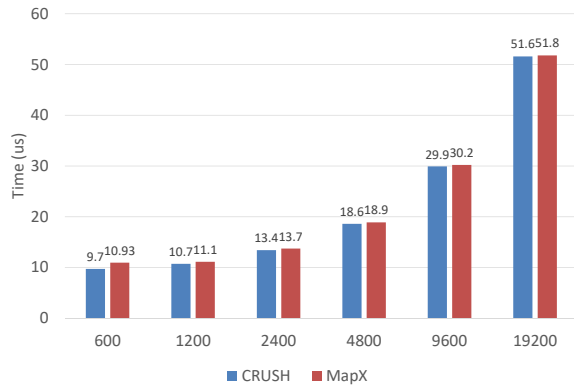
Figure 6: Computation overhead of MAPX and CRUSH.
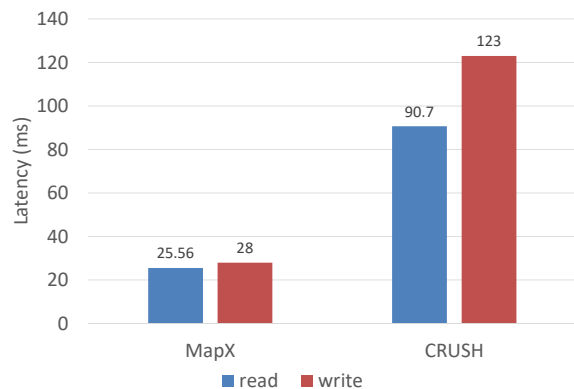
Figure 7: 99th percentile I/O latency of MAPX and CRUSH (during cluster shrinking).

4.2 Computational Overhead

We compare the computation times of MAPX and CRUSH by simulating a Ceph cluster of different numbers of OSDs (varying from 600 to 19,200). The result (Fig 6) shows that both MAPX and CRUSH can map an object to an OSD in tens of microseconds. The small extra times of MAPX compared to CRUSH come from the computation of the time-dimension mapping beneath the root.

4.3 I/O Performance during Shrinking

We evaluate the I/O performance of MAPX (used as the object placement methods for Ceph) in shrinking. The Ceph cluster has three storage machines each initially having three OSDs, and we expand the cluster by adding one OSD to each of the three machines using the same configurations as that in Section 4.1. We then remove the newly-added layer (i.e., removing one OSD from each of the three machines), following the shrinking method (introduced in Section 3.2). We control the migration speed by setting the number of concurrently migrated PGs to eight.

Fig. 7 depicts the 99th percentile I/O latency of MAPX
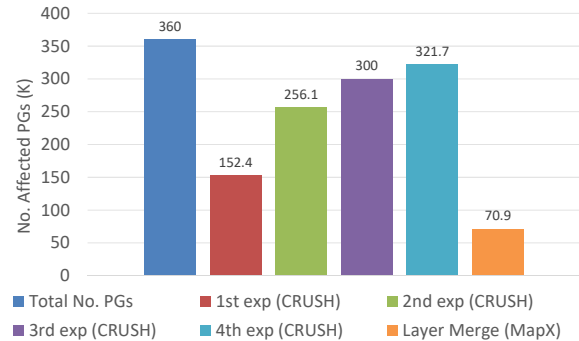
Figure 8: Number of affected PGs in layer merging in MAPX (after four expansions). Since CRUSH does not support merging, for reference we measure the number of affected PGs after each expansion in CRUSH.

during cluster shrinking. For reference, Fig. 7 also shows the 99th percentile latency of CRUSH in shrinking by removing one OSD from each of the three machines. Ceph shrinks the cluster by directly modifying the cluster map. Note that the result does not necessarily mean that MAPX has lower latency than CRUSH in shrinking, because they adopt different throttling mechanisms. However, MAPX outperforms CRUSH during cluster shrinking in that MAPX requires less migration than CRUSH. For instance, removing an OSD in CRUSH will lower the entire subtree's weight and thus may result in unnecessary data migration. In contrast, MAPX never causes migration between preserved OSDs because shrinking occurs at the granularity of layers. We omit the result for IOPS during shrinking due to lack of space, which has similar trends with that for I/O latency.

4.4 Layer Merging

We use CrushTool [6] to emulate layer merging in MAPX. We adopt three-way replication where each object has three replicas stored on three OSDs. Initially the storage cluster consists of 5 racks each having 20 machines. One machine has 20 OSDs. There are totally 100 machines and 2000 OSDs, storing 200,000 PGs. We expand the cluster four times. In each expansion, we add a new layer of one rack (of 20 machines and 400 OSDs), and add 40,000 new PGs to the new layer. Clearly, MAPX maps all the new PGs onto the newly-added OSDs and thus no migration happens. After the four expansions, there are totally 9 racks, 180 machines, and 3600 OSDs, storing 360,000 PGs. We then merge the 40 machines of the first and second expansions (as introduced in Section 3.2), and measure how many PGs are affected by the merging in MAPX.

The result is depicted in Fig. 8, where layer merging in MAPX affects 70,910 PGs among all the 80,000 PGs of the two merged layers. The relatively high ratio of affected PGs in layer merging of MAPX is decided by the

nature of CRUSH. For reference, we also emulate the four expansions in CRUSH, where we let the cluster initially have 360,000 PGs and do not add new PGs during expansions, because otherwise CRUSH will change the mapping from objects to PGs causing many more PGs to be migrated. Fig. 8 also shows how many PGs are affected by each expansion in CRUSH. For instance, almost 90% of all the PGs are affected in the fourth expansion when the number of machines increases from 160 to 180.

5   Related Work

5.1   CRUSH in Ceph

Ceph [66] is a widely-used object-based storage system supporting block storage [3], file storage [4], and simple object storage [8] (like S3 [1]). To deterministically and uniformly maps data objects onto OSDs without relying on a central directory, Ceph applies CRUSH by taking the following two steps.

In the first step, Ceph computes the placement groups (PGs) of the objects. The actual computation of PGs is slightly more complicated than simple hashing and modulo (discussed in Section 2.1) when the PG number (PG_NUM) is not a power of two: it computes the *pgid*s with *double-modulo* by using two values of 2^{nth} power near PG_NUM, so as to minimize *pgid* changes when changing the numbers of PGs. For instance, consider two objects *A* and *B* with HASH(A) $= 25$ and HASH(B) $= 29$. Suppose that at first the PG has PG_NUM$_1 = 8$, which results in $pgid_A = 1$ and $pgid_B = 5$. Then, suppose that we increase the PG number to PG_NUM$_2 = 12$. Since $2^3 < 12 < 2^4$, Ceph first computes the modulo for *A* and *B* using $2^4 = 16$, and respectively gets $pgid_A = 9$ and $pgid_B = 13$. For $pgid_A < $ PG_NUM$_2$, Ceph will take $pgid_A = 9$ as the final *pgid* of *A*. In contrast, for $pgid_B > $ PG_NUM$_2$, Ceph will compute the modulo again using $2^3 = 8$ and get $pgid_B = 5$ as the final *pgid* of *B*. Clearly, the double-modulo mechanism makes the *pgid*s not to change when the first modulo is between PG_NUM$_2 = 12$ and $2^4 = 16$.

In the second step, Ceph maps *pgid*s onto OSDs in the storage cluster, where the hierarchy is composed of OSDs and buckets. Buckets can contain any number of OSDs or other buckets. OSDs are always at the leaves and are assigned weights by the administrator to control the relative amount of data they are responsible for. Bucket weights are the sum of the weights of its items. Currently CRUSH has five types (uniform, list, tree, straw, and straw2) of buckets, and different bucket types use different formulas to choose a given number of items beneath the bucket. The straw2 buckets are the most popular because they have the smallest migration overhead when changing the cluster map or the number of PGs. By default all buckets in Ceph have the *straw2* type.

5.2   Load Balancing & Migration Overhead

Ceph developers have realized the performance degradation problem due to expansion-caused migration. They alleviate this problem through implementation-level optimizations by lowering the priority of migration tasks to avoid bursty migration after the expansion [7]. However, the PGs calculated by CRUSH have to be *eventually* migrated. Further, the conservative migration settings significantly extend the migration period during which a large fraction of PGs are waiting for migration. This complicates their write procedure (first being written to the origin OSDs and then to the target OSDs), unnecessarily increasing the load.

In contrast, MAPX provides administrators with the ability to *control* the migration at the algorithm level: the migration may never happen if (as in most cases) there is not severe imbalance between the loads of different layers. Further, sometimes CRUSH needs to increase the number of PGs, for example to reduce the per-OSD load, which causes a large fraction of objects to be migrated even using the double-modulo method (Section 5.1), while MAPX could smoothly add PGs during expansions without migration.

Focusing on OSD failure caused data migration, Ref. [36] proposes to use *cluster device flags* to selectively label failed OSDs for reducing data transfer. However, it is not clear how to use the flags to address/alleviate the migration problem when expanding the storage clusters.

Consistent distributed hash tables (DHTs) [63, 57, 74, 59, 60, 38, 73] are widely used for decentralized *overlay* storage. Early DHTs require multi-hop routing to locate the data and thus are not suitable for distributed object storage. For example, Chord [63] uses hashing to map both the IDs of storage nodes and the keys of data onto a ring. A node is responsible for a key if it is the nearest node after the key on the ring. Each node only has routing information about a subset of nodes on the ring, and it takes $O(\log N)$ time to locate a key in an *N*-node Chord network. Later DHT networks (like OneHop [18]) support direct key locating by maintaining all routing information on each node in the system, and have been adopted in some decentralized object stores including Amazon Dynamo [28], S3 [1], and OpenStack Swift [11].

Compared to CRUSH, most DHTs cannot express the storage hierarchy including OSDs, machines, racks, etc. DHT-based storage systems have to use additional mechanisms to model the hierarchy (e.g., Cassandra [41] and CubeX [71] respectively adopt virtual nodes and multi-level cubic ring [70], and hierarchy-aware DHTs[33, 51, 29, 39, 69] adopt hierarchical routing tables), which are inflexible compared to CRUSH. Further, load assignment in DHTs is decided by the positions of the nodes and keys on the ring, and thus adding a new node will only make a portion of the load of its successor move to it, which inevitably causes imbalance (although introducing less migration).

5.3 Storage Systems

Decentralized Object storage systems. In recent years, decentralized object storage has been widely used in various scenarios. For example, Twitter uses virtual buckets to store its photos [2], LinkedIn designs Ambry [54] which adopts logical grouping and asynchronous replication to realize geo-distributed object storage [61], and Facebook designs F4 [52] which adopts erasure coding [45] to reduce replication factors for its *warm* objects. Key-value (KV) storage systems [10, 20, 28, 40, 47] could be viewed as generalized object stores that provide an interface for reading, writing, deleting and modifying the values associated with keys. Unlike general object stores, their values are often relatively small.

Centralized Object storage systems. Some object stores adopt a centralized metadata directory to simplify data placement. Haystack [15] is a centralized object store for Facebook's large amounts of small objects like photos, audio/video pieces, H5 files, etc. Haystack places object data (packed into needles) in large files stored in data servers, and stores object positions (i.e., on which machines) in a central directory. Similar to Haystack, Lustre [16] and HDFS [9] leverage a central directory to maintain object positions which helps keep existing objects unaffected during cluster expansions. The central directory based placement methods are inefficient in scalability and robustness. Further, the multi-phase I/O of metadata and data leads to poor performance and complicates consistency issues [23, 22, 55, 34] and thus cannot satisfy the requirement of the emerging OLDI (online data-intensive) applications [25, 68]. Compared to the centralized placement methods, MAPX preserves the benefits of decentralized CRUSH placement algorithm while providing flexible control over data migration in expanding the storage clusters.

Block storage systems. Large-scale block storage systems [65, 49, 42, 35] adopt distributed protocols [12, 17] to provide block interface to remote clients. For example, Ursa [44] designs a hybrid block store for optimizing SSD-based storage [46, 14, 27, 26, 13]. Salus [64] provide virtual disk service based on HBase [31]. Blizzard [50] realizes high-performance parallel I/O based on FDS [53]. PARIX [45, 72] performs speculative partial writes to alleviate the inability of erasure coding (EC) [19, 62, 37] and efficiently support random small writes.

File systems. Distributed file systems spread the data of a file across many storage servers [22, 24, 30, 32, 35, 43, 48, 58]. For instance, GFS [30] is a large-scale fault-tolerant file system for data-intensive cloud applications. Zebra [32] uses striping on RAID [21] and logs for high disk parallelism. BPFS [24] focuses on persistent memory hardware and uses epoch barrier to provide an in-memory file system with ordering guarantees. OptFS [22] improves the journaling file system [56] by decoupling durability from ordering.

6 Conclusion

The contention between decentralized and centralized data placement methods has been long lived in the design of large-scale object storage systems. The decentralized CRUSH method achieves high scalability, robustness, and performance, but suffers from uncontrollable data migration in cluster expansions. This paper presents MAPX, a novel extension to CRUSH that embraces the best of both decentralized and centralized methods. MAPX controls data migration by introducing an extra time-dimension mapping from object creation times to cluster expansion times, while still preserving the randomness and uniformness of CRUSH. We have applied MAPX to Ceph-RBD and CephFS, respectively by extending the *rbd_header* and *inode* metadata structures. In our future work, we will study how to reduce the maintenance overhead of object timestamps, so as to apply MAPX to a broader range of object-based storage scenarios.

Acknowledgement

References

[1] `https://aws.amazon.com/s3/`.

[2] `https://blog.twitter.com/engineering/en_us/a/2012/blobstore-twitter-s-in-house-photo-storage-system.html`.

[3] `https://ceph.com/ceph-storage/block-storage/`.

[4] `https://ceph.com/ceph-storage/file-system/`.

[5] `https://docs.ceph.com/docs/master/releases/luminous/`.

[6] `https://docs.ceph.com/docs/mimic/man/8/crushtool/`.

[7] `https://docs.ceph.com/docs/mimic/rados/configuration/osd-config-ref/`.

[8] `https://github.com/ceph/ceph/tree/master/src/rgw`.

[9] `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

[10] `https://rocksdb.org/`.

[11] `https://www.swiftstack.com/product/open-source/openstack-swift/`.

[12] AIKEN, S., GRUNWALD, D., PLESZKUN, A. R., AND WILLEKE, J. A performance analysis of the iscsi protocol. In *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on* (2003), IEEE, pp. 123–134.

[13] ANAND, A., MUTHUKRISHNAN, C., KAPPES, S., AKELLA, A., AND NATH, S. Cheap and large cams for high performance data-intensive networked systems. In *NSDI* (2010), USENIX Association, pp. 433–448.

[14] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAY-EE, A., TAN, L., AND VASUDEVAN, V. Fawn: a fast array of wimpy nodes. In *SOSP* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 1–14.

[15] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., AND VAJGEL, P. Finding a needle in haystack: facebook's photo storage. In *Usenix Conference on Operating Systems Design and Implementation* (2010), pp. 47–60.

[16] BRAAM, P. The lustre storage architecture. *arXiv preprint arXiv:1903.01955* (2019).

[17] CASHIN, E. L. Kernel korner: Ata over ethernet: putting hard drives on the lan. *Linux Journal 2005*, 134 (2005), 10.

[18] CASTRO, M., COSTA, M., AND ROWSTRON, A. I. T. Debunking some myths about structured and unstructured overlays. In *NSDI* (2005).

[19] CHAN, J. C., DING, Q., LEE, P. P., AND CHAN, H. H. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (2014), pp. 163–176.

[20] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *Acm Transactions on Computer Systems 26*, 2 (2008), 1–26.

[21] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR) 26*, 2 (1994), 145–185.

[22] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 228–243.

[23] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012* (2012), p. 9.

[24] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 133–146.

[25] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[26] DEBNATH, B., SENGUPTA, S., AND LI, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2011), SIGMOD '11, ACM, pp. 25–36.

[27] DEBNATH, B. K., SENGUPTA, S., AND LI, J. Flashstore: High throughput persistent key-value store. *PVLDB 3*, 2 (2010), 1414–1425.

[28] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. *Acm Sigops Operating Systems Review 41*, 6 (2007), 205–220.

[29] GANESAN, P., GUMMADI, P. K., AND GARCIA-MOLINA, H. Canon in g major: Designing dhts with hierarchical structure. In *ICDCS* (2004), pp. 263–272.

[30] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *SOSP* (2003), pp. 29–43.

[31] HARTER, T., BORTHAKUR, D., DONG, S., AIYER, A., TANG, L., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis of hdfs under hbase: A facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (2014), pp. 199–212.

[32] HARTMAN, J. H., AND OUSTERHOUT, J. K. The zebra striped network file system. *ACM Transactions on Computer Systems (TOCS) 13*, 3 (1995), 274–310.

[33] HARVEY, N. J. A., JONES, M. B., SAROIU, S., THEIMER, M., AND WOLMAN, A. Skipnet: A scalable overlay network with practical locality properties. In *USENIX Symposium on Internet Technologies and Systems* (2003).

[34] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[35] HILDEBRAND, D., AND HONEYMAN, P. Exporting storage systems in a scalable manner with pnfs. In *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)* (2005), IEEE, pp. 18–27.

[36] HUANG, M., LUO, L., LI, Y., AND LIANG, L. Research on data migration optimization of ceph. In *2017 14th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)* (2017), IEEE, pp. 83–88.

[37] JIN, C., FENG, D., JIANG, H., AND TIAN, L. Raid6l: A log-assisted raid6 storage architecture with improved write performance. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)* (2011), IEEE, pp. 1–6.

[38] KAASHOEK, M. F., AND KARGER, D. R. Koorde: A simple degree-optimal distributed hash table. In *IPTPS* (2003), pp. 98–107.

[39] KARGER, D. R., AND RUHL, M. Diminished chord: A protocol for heterogeneous subgroup formation in peer-to-peer networks. In *IPTPS* (2004), pp. 288–297.

[40] LAKSHMAN, A., AND MALIK, P. Cassandra:a structured storage system on a p2p network. In *Proc Acm Sigmod International Conference on Management of Data* (2009).

[41] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[42] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed virtual disks. In *ACM SIGPLAN Notices* (1996), vol. 31, ACM, pp. 84–92.

[43] LEUNG, A. W., PASUPATHY, S., GOODSON, G. R., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *USENIX annual technical conference* (2008), vol. 1, pp. 2–5.

[44] LI, H., ZHANG, Y., LI, D., ZHANG, Z., LIU, S., HUANG, P., QIN, Z., CHEN, K., AND XIONG, Y. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Proceedings of the Fourteenth EuroSys Conference 2019* (2019), ACM, p. 15.

[45] LI, H., ZHANG, Y., ZHANG, Z., LIU, S., LI, D., LIU, X., AND PENG, Y. Parix: speculative partial writes in erasure-coded systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), USENIX Association, pp. 581–587.

[46] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 1–13.

[47] LU, L., GOPALAKRISHNAN, H., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Wisckey: Separating keys from values in ssd-conscious storage. *Acm Transactions on Storage 13*, 1 (2017), 5.

[48] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS) 2*, 3 (1984), 181–197.

[49] MEYER, D. T., AGGARWAL, G., CULLY, B., LEFEBVRE, G., FEELEY, M. J., HUTCHINSON, N. C., AND WARFIELD, A. Parallax: virtual disks for virtual machines. In *ACM SIGOPS Operating Systems Review* (2008), vol. 42, ACM, pp. 41–54.

[50] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., KHAN, O., AND NAREDDY, K. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), pp. 257–273.

[51] MISLOVE, A., AND DRUSCHEL, P. Providing administrative control and autonomy in structured peer-to-peer overlays. In *IPTPS* (2004), pp. 162–172.

[52] MURALIDHAR, S., LLOYD, W., ROY, S., HILL, C., LIN, E., LIU, W., PAN, S., SHANKAR, S., SIVAKUMAR, V., AND TANG, L. f4: Facebook's warm blob storage system. In *Usenix Conference on Operating Systems Design and Implementation* (2014), pp. 383–398.

[53] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O., HOW-ELL, J., , AND SUZUE, Y. Flat datacenter storage. In *OSDI* (2012).

[54] NOGHABI, S. A., SUBRAMANIAN, S., NARAYANAN, P., NARAYANAN, S., HOLLA, G., ZADEH, M., LI, T., GUPTA, I., AND CAMPBELL, R. H. Ambry:linkedin's scalable geo-distributed object store. In *International Conference on Management of Data* (2016), pp. 253–265.

[55] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J. K., AND ROSENBLUM, M. Fast crash recovery in ramcloud. In *SOSP* (2011), pp. 29–41.

[56] PIERNAS, J., CORTES, T., AND GARCÍA, J. M. Dualfs: a new journaling file system without meta-data duplication. In *Proceedings of the 16th international conference on Supercomputing* (2002), ACM, pp. 137–146.

[57] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R. M., AND SHENKER, S. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA* (2001), pp. 161–172.

[58] REN, K., ZHENG, Q., PATIL, S., AND GIBSON, G. 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* (2014), IEEE Press, pp. 237–248.

[59] ROWSTRON, A. I. T., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware* (2001), pp. 329–350.

[60] SHEN, H., XU, C.-Z., AND CHEN, G. Cycloid: A constant-degree and lookup-efficient p2p overlay network. *Perform. Eval. 63*, 3 (2006), 195–216.

[61] SPIROVSKA, K., DIDONA, D., AND ZWAENEPOEL, W. Optimistic causal consistency for geo-replicated key-value stores. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on* (2017), IEEE, pp. 2626–2629.

[62] STODOLSKY, D., GIBSON, G., AND HOLLAND, M. Parity logging overcoming the small write problem in redundant disk arrays. In *ACM SIGARCH Computer Architecture News* (1993), vol. 21, ACM, pp. 64–75.

[63] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (2001), 149–160.

[64] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANAN-DAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the salus scalable block store. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (2013), pp. 357–370.

[65] WARFIELD, A., ROSS, R., FRASER, K., LIMPACH, C., AND HAND, S. Parallax: Managing storage for a million machines. In *HotOS* (2005).

[66] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), pp. 307–320.

[67] WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. Crush: Controlled, scalable, decentralized placement of replicated data. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (2006), IEEE, pp. 31–31.

[68] ZAHARIA, M., CHOWDHURY, M., DAS, T., AND DAVE, A. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI* (2012), pp. 1–14.

[69] ZHANG, Y., CHEN, L., LU, X., AND LI, D. Enabling routing control in a dht. *IEEE Journal on Selected Areas in Communications 28*, 1 (2009), 28–38.

[70] ZHANG, Y., LI, D., GUO, C., WU, H., XIONG, Y., AND LU, X. Cubicring: Exploiting network proximity for distributed in-memory key-value store. *IEEE/ACM Transactions on Networking 25*, 4 (2017), 2040–2053.

[71] ZHANG, Y., LI, D., AND LIU, L. Leveraging glocality for fast failure recovery in distributed ram storage. *ACM Transactions on Storage (TOS) 15*, 1 (2019), 1–24.

[72] ZHANG, Y., LI, H., LIU, S., XU, J., AND XUE, G. Pbs: An efficient erasure-coded block storage system based on speculative partial writes. *ACM Transactions on Storage (TOS) 15* (2020), 1–26.

[73] ZHANG, Y., AND LIU, L. Distributed line graphs: A universal technique for designing dhts based on arbitrary regular graphs. *IEEE Transactions on Knowledge and Data Engineering 24*, 9 (2011), 1556–1569.

[74] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*, 1 (2004), 41–53.

Lock-Free Collaboration Support for Cloud Storage Services with Operation Inference and Transformation [*]

Jian Chen[1*], Minghao Zhao[1*], Zhenhua Li[1✉], Ennan Zhai[2]
Feng Qian[3], Hongyi Chen[1], Yunhao Liu[1,4], Tianyin Xu[5]

[1]*Tsinghua University,* [2]*Alibaba Group,* [3]*University of Minnesota,* [4]*Michigan State University,* [5]*UIUC*

Abstract

This paper studies how today's cloud storage services support collaborative file editing. As a tradeoff for transparency/user-friendliness, they do not ask collaborators to use version control systems but instead implement their own heuristics for handling conflicts, which however often lead to unexpected and undesired experiences. With measurements and reverse engineering, we unravel a number of their design and implementation issues as the root causes of poor experiences. Driven by the findings, we propose to reconsider the collaboration support of cloud storage services from a novel perspective of *operations* without using any locks. To enable this idea, we design intelligent approaches to the inference and transformation of users' editing operations, as well as optimizations to the maintenance of files' historic versions. We build an open-source system UFC2 (User-Friendly Collaborative Cloud) to embody our design, which can avoid most (98%) conflicts with little (2%) overhead.

1 Introduction

Computer-supported collaboration allows a group of geo-distributed people (*i.e.,* collaborators) to cooperatively work online. To enable this, the most common technique is Version Control Systems (VCSes) like Git, SVN and Mercurial, which require the mastery of complex operations and thus are not suited to non-technical users [58]. In contrast, dedicated online editors, such as Google Docs and Overleaf, provide web-based easy-to-use collaboration support, but with limited functions and "walled-garden" concerns [8, 10, 13, 68]. As an alternative approach, cloud storage services (*e.g.,* Dropbox, OneDrive, Google Drive, and iCloud) have recently evolved their functionality from simple file backup to online collaboration. For example, over 300,000 teams have adopted Dropbox for business collaboration, submitting ∼4000 file edits per second [62]. For ease of use, collaboration is made transparent by almost every service today through *automatic file synchronization*. When a user modifies a file in a "sync folder" (a local directory created by the service), the changed file will be automatically synchronized with the copy maintained at

Pattern 1: Losing updates	
Alice is editing a file. Suddenly, her file is overwritten by a new version from her collaborator, Bob. Sometimes, Alice can even lose her edits on the older version.	All studied cloud storage services
Pattern 2: Conflicts despite coordination	
Alice coordinates her edits with Bob through emails to avoid conflicts by enforcing a sequential order. Every edit is saved instantly. Even so, conflicts still occur.	All studied cloud storage services
Pattern 3: Excessively long sync duration	
Alice edits a shared file and confirms that the edit has been synced to the cloud. However, Bob does not receive the updates for an excessively long duration.	Dropbox, OneDrive, SugarSync, Seafile, Box
Pattern 4: Blocking collaborators by opening files	
Alice simply opens a shared Microsoft Office file without making any edits. This mysteriously disables Bob's editing the file.	Seafile (only for Microsoft Office files)

Table 1: Common patterns of unexpected and undesired collaborative editing experiences studied in this paper.

the cloud side. Then, the cloud will further distribute the new version of the file to the other users sharing the file.

Collaboration inevitably introduces *conflicts* – simultaneous edits on two different copies of the same file. However, it is non-trivial to automatically resolve conflicts, especially if the competing edits are on the same line of the file. Instead of requiring users to learn complex diff-and-merge instructions to solve conflicts in VCSes, all of today's cloud storage services opt for transparency and user-friendliness – they devise different approaches to preventing conflicts or automatically resolving conflicts. Unfortunately, these efforts do not work well in practice, often resulting in unexpected results. Table 1 describes four common patterns of unexpected/undesirable collaborative experiences caused by cloud storage services.

To "debug" these patterns from the inside out, we study eight widely-used cloud storage services based on traffic analysis with trace-driven experiments and reverse engineering. The studied services include Dropbox, OneDrive, Google Drive, iCloud Drive, Box [2], SugarSync [20], Seafile [16], and Nutstore [11]. Also, we collect ten real-world collaboration traces, among which seven come from the users of different services and the other three come from the contributors of well-known projects hosted by Github. Our study results reveal a number of design issues of collaboration support in today's cloud storage services. Specifically, we find:

---

*[*]Co-primary authors. Zhenhua Li is the corresponding author.
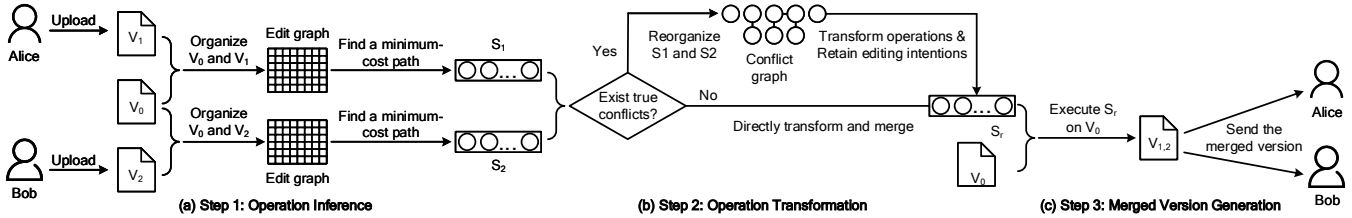
Figure 1: Working principle for merging two versions of the same file at the cloud side: (a) inferring the operation sequences S_1 and S_2 that respectively change V_0 to V_1 and V_2 using edit graphs; (b) transforming and merging S_1 and S_2 into S_r with the minimal conflict, based on a conflict graph and topological sorting when necessary; (c) executing S_r on V_0 to generate the merged version $V_{1,2}$.

- Using file-level locks to prevent conflicts is difficult due to the unpredictability of users' real-time editing behavior (as cloud storage services can neither designate nor monitor the editor) and the latency between clients and the server.

- Existing conflict-resolution solutions are too coarse-grained and do not consider user intention – they either keep the latest version based on the server-side timestamp or distribute all the conflicting versions to the users.

Most surprisingly, we observe that the majority of "conflicts" reported by these cloud storage services are not *true conflicts* but are artificially created. In those false-positive conflicts (or false conflicts), the collaborators were editing different parts of a shared file. This is echoed by the common practice of mitigating false conflicts in cloud storage service-based collaborative editing by intentionally dividing an entire text file into multiple separate files [18, 23]. Such false conflicts can be automatically resolved at the server side without user intervention.

In this paper, we show that it is feasible to provide effective collaboration support in cloud storage services by intelligently merging conflicting file versions using the *three-way merge* method [54, 63], where two conflicting versions are merged based on a common-context version. This is enabled by the inference and transformation of users' editing operations; meanwhile, no lock is used so as to achieve the transparency and user-friendliness. As depicted in Figure 1, our basic idea is to first infer the collaborators' operation sequences [1(a)], and then transform these sequences based on their true conflicts (if any) [1(b)] to generate the final version [1(c)]. Compared to a file-level or line-level conflict resolution (*e.g.,* adopted by Dropbox or Git), our solution is more fine-grained: modifications on different parts of the same file or even the same line can be automatically merged.

Building a system with the above idea, however, requires us to address two technical challenges. First, inferring operation sequences in an efficient way is non-trivial, since it is a computation-intensive task for cloud storage services[1]. As illustrated in Figure 1(a), when two versions V_1 and V_2 emerge, we need to first find the latest *common-context* version V_0

hosted at the cloud, and then infer two operation sequences S_1 and S_2 that convert V_0 to V_1 and V_2, respectively. The common approach using dynamic programming [33, 44, 57] may take excessive computing time in our scenario, *e.g.,* ∼30 seconds for a 500-KB file. To address the issue, we leverage an *edit graph* [4, 55] to organize V_0 and V_1, and thus essentially reduce the inference time, *e.g.,* ∼200 ms for a 500-KB file.

The second challenge is how to transform and merge S_1 and S_2 into S_r with *minimal conflict*, *i.e.,* 1) simplifying manual conflict resolution of text files by sending only one merged version ($V_{1,2}$) to the collaborators; and 2) retaining the collaborators' editing intentions while minimizing the amount of conflicts to be manually resolved in $V_{1,2}$. As illustrated in Figure 1(b), it is easy to directly transform and merge S_1 and S_2, via *operation transformation* [39], if there is no true conflict. To address the challenging case (of true conflicts), we utilize a *conflict graph* [53] coupled with *topological sorting* to reorganize all operations, so as to prioritize the transformation of real conflicting operations and minimize their impact on the transformation of other operations.

Besides solving the above challenges, we facilitate conflict resolution by maintaining each shared file's historic versions at the cloud with CDC (content-defined chunking [59]) deduplication. For a user-uploaded version, we adopt full-file sync for small files and delta sync for larger files to achieve the shortest upload time. For a server-merged version, we design *operation-based CDC* (OCDC) which exploits the implicit operations inferred during conflict resolution to accelerate CDC – only the boundaries of those chunks affected by the operations need recalculation.

We build UFC2 (User-Friendly Collaborative Cloud) on top of Amazon EFS (Elastic File System) and S3 to implement our design. Our evaluation using real-world traces indicates that conflicts generated during collaboration are significantly reduced by 98% on average (the remainder are true conflicts). Meanwhile, the incurred time overhead by a conflict resolution is usually between 10 and 80 ms, which is merely 0.6%–4% (2% on average) of the delivery time for a file update. In addition, our designed OCDC optimization outpaces the traditional CDC by ∼3 times, thus reducing the data chunking time from 30–400 ms to 10–120 ms for a common file. Finally, we have made all the source code and measurement data publicly available at https://UFC2.github.io.

---

[1]In contrast, it is straightforward and lightweight to acquire a user's operation sequences in Google Docs [7], Overleaf [15], and similar services, where a dedicated editor is used and monitored in real time.

Trace	Timespan	# Col-s	# Files	# Versions	Avg. File Size	Major File Types
Dropbox-1	11/2/2018–2/6/2019	5	305	3527	86 KB	tex (52%), pdf (16%), Matlab src (24%) & fig (4%)
Dropbox-2	4/3/2019–5/14/2019	6	216	2193	67 KB	tex (57%), pdf (21%), Matlab fig (9%)
OneDrive	3/15/2019–5/31/2019	5	253	2673	83 KB	tex (61%), pdf (15%), Matlab fig (7%)
iCloud Drive	2/1/2019–4/30/2019	6	301	3211	59 KB	tex (53%), pdf (22%), Matlab fig (12%)
Box	3/21/2019–5/2/2019	8	273	2930	60 KB	tex (66%), pdf (27%)
SugarSync	4/11/2019–5/26/2019	9	325	3472	89 KB	tex (49%), pdf (25%), Matlab src (19%) & fig (3%)
Seafile	2/17/2019–4/30/2019	7	251	2823	71 KB	tex (55%), pdf (19%), Matlab fig (10%)
Spark-Git	1/15/2018–3/27/2019	58	15181	129957	4 KB	Scala (78%), Java (6%), py (5%)
TensorFlow-Git	7/24/2018–3/27/2019	86	16754	246016	9 KB	py (30%), C header (14%) & src (29%), txt (20%)
Linux-Git	9/9/2018–3/30/2019	87	63865	901167	13 KB	C header (31%) & src (42%), txt (16%)

Table 2: Statistics of the ten real-world collaboration traces. "Col-s" means collaborators, "src" means source code, and "py" means python.

2 Design Challenges

In this section, we employ trace-driven experiments, special benchmarks, and reverse engineering to deeply understand the design challenges of collaborative support in today's cloud storage services. In particular, we analyze the root causes of poor experiences listed in Table 1.

2.1 Study Methodology

In order to quantitatively understand how today's cloud storage services behave under typical collaborative editing workloads, we first collected ten real-world collaboration traces as listed in Table 2. Among them, seven are provided by users (with informed consent) that collaborate on code/document writing using different cloud storage services. The other three are extracted from well-known open-source GitHub projects. Each trace contains all the file versions uploaded by every involved user during the collection period.

For the first seven traces, relatively few (*i.e.,* 5–9) collaborators work on a project for a couple of months. Each of their workloads is unevenly distributed over time: during some periods collaborators frequently edit the shared files, whereas during the other periods there are scarcely any edits to the shared files. By contrast, in the last three traces, a large number of collaborators constantly submit their edits for quite a few months, and thus generate many more file versions. In addition, the collaborators involved in all the ten traces are located across multiple continents.

Using these traces, we conducted a comparative measurement study of eight mainstream cloud storage services: Dropbox, OneDrive, Google Drive, iCloud Drive, Box, SugarSync, Seafile, and Nutstore. For each service, we ran its latest PC client (as of Jul. 2019) on Windows-10 VMs rented from Amazon EC2; these VMs have the same hardware configuration (a dual-core CPU@2.5 GHz, 8 GB memory, and 32 GB SSD storage) and network connection (whose downlink/uplink bandwidth is restricted to 100 / 20 Mbps by WonderShaper to resemble a typical residential network connection [1, 19]).

We deployed puppet collaborators on geographically distributed VMs across five major regions to replay a trace, with one client software and one puppet collaborator running on one VM. Specifically, we rented AWS VMs in South America, North America, Europe, the Middle-East, and the Asia-Pacific (including East Asia and Australia). We instructed the puppet collaborators to upload different file versions (as recorded in the trace) to the cloud. To safely reduce the duration of the replay, we skipped the "idle" timespan in the trace during which no file is edited by any collaborator. In addition, we strategically generated some "corner cases" that seldom appear in users' normal editing, so as to make a deeper and more comprehensive analysis. For example, we edited fix-sized small (KB-level) files to measure cloud storage services' sync delay, so as to avoid the impact of file size variation; we edited a random byte on a compressed file to figure out their adoption of delta sync mechanisms; and we performed specially controlled edits to investigate their usage of locks, as well as their delivery time of lock status.

We captured all the IP-level sync traffic in the trace-driven and benchmark experiments via Wireshark [25]. From the traffic, we observe that almost all the communications during the collaboration are carried out with HTTPS sessions (using TLS v1.1 or v1.2). By analyzing the traffic size and occurrence time of respective HTTPS sessions, we can understand the basic design of these eight mainstream cloud storage services, *e.g.,* using full-file sync or delta sync mechanisms to deliver a file update.

To reverse engineer the implementation details, we attempted to reverse HTTPS by leveraging man-in-the-middle attacks with Charles [3], and succeeded with OneDrive, Box, and Seafile. For the three services, we are able to get the detailed information of each synced file (including its ID, creation time, edit time, and to our great surprise the concrete content), as well as the delivered lock status and file update. Furthermore, since Seafile is open source, we also read the source code to understand the system design and implementation, *e.g.,* its adoption of FIFO message queues and the CDC delta sync algorithm.

For the remaining five cloud storage services, we are unable to reverse their HTTPS sessions, as their clients do not accept the root CA certificates forged by Charles. For these services, we search the technical documentation (including design documents and engineering blogs) to learn about their designs, such as locks and message queues [5, 9, 12, 14, 21, 22, 31].

Cloud Storage Service	Lock Mechanism	Conflict Resolution	Message Queue	File Update Method
Dropbox	No lock	Keep all the conflicting versions	LIFO	`rsync`
OneDrive	No lock	Keep all the conflicting versions	Queue	Full-file sync
Google Drive	No lock	Keep only the latest version	-	Full-file sync
iCloud Drive	No lock	Ask users to choose among multiple versions	-	`rsync`
Box	Manual locking	Keep all the conflicting versions	Queue	Full-file sync
SugarSync	No lock	Keep all the conflicting versions	Queue	`rsync`
Seafile	Automatic/manual*	Keep all the conflicting versions	FIFO	CDC
Nutstore	Automatic locking	Keep all the conflicting versions	-	Full-file & `rsync`

Table 3: A brief summary of the collaboration support of the eight mainstream cloud storage services in our study. "∗": Seafile only supports automatic locking for Microsoft Office files. "-": we do not observe obvious queuing behavior.

2.2 Results and Findings

Our study quantifies the occurrence of conflicts in different cloud storage services, and uncovers their key design principles as summarized in Table 3.

Occurrence probability of conflicts. When the ten traces are replayed with each cloud storage service, we find considerable difference (ranging from 0 to 4.8%) in the *ratio* of conflicting file versions (generated during a replay) over all versions, as shown in Table 4. Most notably, Google Drive appears to have never generated conflicts, because once it detects conflicting versions of a file (at the cloud) it only keeps the latest version based on their server-side timestamps. In contrast, the most conflicting versions are generated with iCloud Drive, because its *sync delay* (*i.e.,* the delivery time of a file update) is generally longer than that of the other cloud storage services (as later indicated in Figure 3 and Table 5). In comparison, for each trace Nutstore generates the fewest conflicting versions (with Google Drive not considered), as its automatic locking during collaboration can avoid a portion (7.6%–19.1%) of conflicts.

Locks. We observe that the majority of the studied cloud storage services (Dropbox, OneDrive, Google Drive, iCloud Drive, and SugarSync) never use any form of locks for files being edited. As a consequence, collaboration using these products can easily lead to conflicts. Box, Seafile, and Nutstore use coarse-grained file-level locks; unfortunately, we find that their use of locks is either too early or too late[2], leading to undesired experiences. This is because cloud storage services are unable to acquire users' real-time editing behaviors and thus cannot accurately determine when to request/release locks. Specifically, locking too early leads to Pattern 4 in Table 1, locking too late (locking after editing) leads to Pattern 1, and unlocking too early leads to Pattern 2.

Box only supports manual locks on shared files. When Alice attempts to lock a shared file f and Bob has not opened it, f is successfully locked by Alice and then Bob cannot edit it (until it is manually unlocked by Alice). However, if Bob

---

[2]Ideally, a file should be locked right before the user starts editing, and unlocked right after the user finishes the editing.

has already opened f when Alice attempts to lock it, he can still edit it but cannot save it, because when Bob attempts to save his edit the file editor (*e.g.,* MS Word) will re-check the permission of f. *In essence, Box implements locks by creating a process on Bob's PC, which attempts to "lock" a file by changing the file's permission as read-only.* In this case, if Bob is using an *exclusive* editor (not allowing other applications to write the file it opened), Alice's edits cannot be synced to Bob, thus leading to Pattern 3; otherwise, Bob's edits will be overwritten by Alice's, leading to Pattern 1.

Seafile automatically locks a shared file f when f is opened by an MS Office application, and f will not be unlocked until it is closed. This locking mechanism is coarse-grained and may lead to Pattern 4. For non-MS Office files, Seafile supports manual locks in the same way as Box, and thus they have the same issue in collaboration.

Nutstore attempts to lock a shared file f automatically, when Alice saves her edit. At this time, if Bob has not opened f, f is successfully locked by Alice and Bob cannot edit it; after Alice's saved edit is propagated to Bob, f is automatically unlocked. However, if Bob opened the shared file before Alice saves the file, Nutstore has the same problems as Box and Seafile (Patterns 1 and 3 in Table 1).

Finally, we are concerned with the delivery time of a *lock status* (*i.e.,* whether a file is locked). According to our measurements, the lock status is delivered in real time with ∼100% success rates. As in Figure 2, the delivery time ranges from 0.7 to 1.6 seconds, averaging at 1.0 second. This indicates that today's cloud storage services implement dedicated infrastructure (*e.g.,* queues) for managing locks.

In summary, implementing desirable locks in cloud storage services is not only complex and difficult but also somewhat expensive. Therefore, we feel it wiser to give up using locks.

Conflict resolution. We find three different strategies for resolving the conflicts. *First*, Google Drive only keeps the latest version (defined by the timestamp each version arrives at the cloud). All the older versions are discarded and can hardly be recovered by the users (Google Drive does not reserve a version history for any file). Note that this notion of "latest" may not reflect the absolute latest (which depends on the client-side time), *e.g.,* when the real latest version
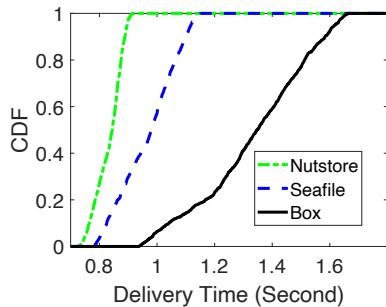
Figure 2: CDF of the delivery time of a lock status. Note that among all the studied services, only three of them use locks.
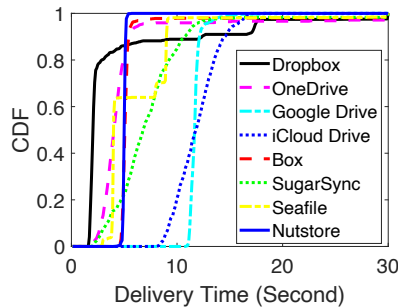
Figure 3: CDF of the delivery time of a file update, where the file is several KBs in size.
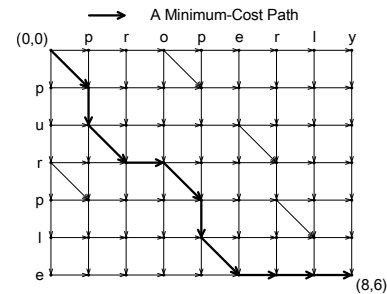
Figure 4: A simple edit graph for reconciling V_0 (the horizontal word "properly") and V_1 (the vertical word "purple").

Trace	DB	OD	GD	ID	Box	SS	SF	NS
DB1	4.4%	4.4%	0	4.5%	4.3%	4.3%	4.3%	3.6%
DB2	4.7%	4.7%	0	4.8%	4.6%	4.7%	4.6%	3.8%
OD	4.1%	4.1%	0	4.2%	4.0%	4.0%	4.1%	3.5%
ID	4.1%	4.0%	0	4.1%	4.1%	4.1%	4.1%	3.4%
Box	4.3%	4.3%	0	4.4%	4.2%	4.3%	4.3%	3.7%
SS	4.2%	4.1%	0	4.2%	4.2%	4.1%	4.2%	3.7%
SF	4.5%	4.5%	0	4.6%	4.5%	4.5%	4.5%	3.8%
SG	1.3%	1.3%	0	1.3%	1.3%	1.3%	1.3%	1.2%
TG	3.5%	3.5%	0	3.5%	3.5%	3.5%	3.5%	3.2%
LG	4.0%	4.0%	0	4.0%	4.0%	4.0%	4.0%	4.0%

Table 4: Ratio of conflicting file versions (over all versions) when the ten traces are replayed with each of the studied cloud storage services. DB=Dropbox, OD=OneDrive, GD=Google Drive, ID=iCloud Drive, SS=SugarSync, SF=Seafile, NS=Nutstore, SG=Spark-Git, TG=TensorFlow-Git, and LG=Linux-Git.

Cloud Service	Min	Median	Mean	P99	Max
Dropbox	1.6	2.0	141.2	**312**	**17751**
OneDrive	1.6	4.0	33.4	**106**	**4415**
Google Drive	10.9	11.7	11.7	12.9	18.1
iCloud Drive	8.1	11.8	11.9	11.9	16.9
Box	4.4	5.1	41.8	**115**	**6975**
SugarSync	2.0	6.8	51.3	**124**	**7094**
Seafile	2.7	4.0	53.8	**99**	**9646**
Nutstore	4.2	5.0	5.0	5.0	5.6

Table 5: Statistics (in unit of second) of the delivery time of a file update, where the file is several KBs in size.

arrives earlier due to network latency. *Second*, iCloud Drive asks the user to choose one version from all the conflicting versions. The user has to compare them by hand, and then make a decision (which is often not ideal). *Third*, a more common solution is to keep all the conflicting versions in the shared folder, and disseminate them to all the collaborators. This solution is more conservative (which does not cause data loss), but leaves all burdens to users. Moreover, given the distributed nature, merging efforts from the collaborators could cause further conflicts if not coordinated well.

Given the difficulties in resolving conflicts, we advocate that cloud storage services should make more effort to proactively avoid, or at least significantly reduce, the conflicts.

Delivery latency and message queue. Delivery latency of a file (update) prevalently exists in cloud storage at both infrastructure (*e.g.,* S3 and Azure Blob) and service (*e.g.,* Dropbox) levels [34, 35, 43, 64, 67, 74]. It stems from multiple factors such as network jitter, system I/O, and load balancing in the datacenter [43, 50]. We measure the delivery time of a file update regarding the eight cloud storage services. As in Figure 3 and Table 5, some services always have reasonable delivery time. On the other hand, in a few services, the maximum

delivery time reaches several hours for a KB-level file, and the 99-percentile (P99) delivery time can reach hundreds of seconds. The unpredictability and long tail latency can sometimes break the time order among file updates, which is the main root cause of Patterns 2 and 3.

Additionally, we find that the implementation of message queues in some cloud storage services aggravates the delivery latency. Specifically, different services have very different message queue implementations, leading to different queueing behaviors. For a FIFO queue (used by Seafile), when the server is overloaded, many requests for file/fetch updates are processed by the server but not accepted by the client due to client-side timeout, thus wasting the server's processing resources. This problem can be mitigated by using LIFO queues (used by Dropbox). However, for a LIFO queue, the requests from "unlucky" users (who encounter the server's being overloaded after issuing fetch update requests) wait for a long duration. We suspect that the services with excessively long delivery time are using big shared queues with no QoS consideration, and may benefit from using a dedicated queue like QJUMP [41].

File update methods. Collaboration results in frequent, short edits to files. *Delta sync* is known to be efficient in updating short edits, compared with *full-file sync* where the whole file has to be transferred [49]. To understand the file update method, we let Alice modify a *Z*-byte highly compressed file,

where $Z \in \{1, 1K, 10K, 100K, 1M\}$, and observed the traffic usage in delivering the file update. By comparing the traffic usages in uploading and downloading an update, we find that OneDrive, Google Drive, and Box adopt full-file sync, and the others adopt delta sync (`rsync` [72] or CDC [59]). Especially, we confirm Seafile's adoption of CDC from its source code [17]. In terms of Nutstore, it adopts a hybrid file update method: full-file sync for small ($\leq$64 KB) files and delta sync for the other files, so as to achieve the highest update speed, because small and large files are more suitable for full-file and delta sync, respectively (full-file sync requires fewer rounds of client-server message exchanges).

2.3 Implications

Our study results show that today's cloud storage services either do not use any locks or use coarse-grained file-level locks to prevent conflicts. The former would inevitably lead to conflicts. The latter, however, is hard to prevent conflicts in practice for two reasons: 1) it is hard to accurately predict user's editing behaviors in real time and therefore to determine the timing of applying the lock, and 2) the latency between the client and the server can vary significantly, so file-level conflicts are generally inevitable. Furthermore, the study shows that full-file and delta sync methods can be combined to accelerate the delivery of a file update. To address the revealed issues, we explore the possibility of developing lock-free conflict resolution by inferring fine-grained user intentions. We also explore a hybrid design of full-file and delta sync methods for efficient file update and synchronization.

3 Our Solution

This section aims to address the challenges uncovered in §2. Our key idea is to model file editing events as `insert` or `delete` *operations* (§3.2). Based on the operation model, we *infer* the collaborators' operation sequences (§3.3), and then *transform* these sequences (§3.4) based on their conflicts to generate the final version. We explain the above procedure with a simple case of two file versions, and demonstrate its applicability to the complex case of multiple versions (§3.5). We also design optimizations to the maintenance of shared files' historic versions (§3.6),

3.1 True and False Conflicts

We examine the conflicting file versions as listed in Table 4 in great detail. We find that ∼1/3 of them come from non-text (*e.g.,* PDF or EXE) files, which, as mentioned in §1, are typically generated based on text files and thus can be simply deleted or regenerated from text files for pretty easy conflict resolution. The remainder relate to text files, the vast majority of which, to our surprise, only contain "false positive" con-

flicts as the collaborators in fact operated on different parts of a shared file.

Take the Dropbox-1 collaboration trace as an example. When it is replayed with Dropbox or OneDrive, among the 3,527 file versions hosted at the cloud side, 154 text files are considered (by Dropbox and OneDrive) to be conflicting versions and then distributed to all the collaborators. Actually, 152 out of the 154 *apparently* conflicting versions can be correctly merged at the cloud side. The remaining two cannot be correctly merged as two collaborators happen to edit the same part of the shared file in parallel, thus generating 9 *true* conflicts. In other words, the vast majority of the (coarse-grained) file-level conflicts are *false* (positive) conflicts when seen at the (fine-grained) operation level.

3.2 Explicit and Implicit Operations

We model *operation* as the basic unit in collaborative file editing. A shared file can be regarded as a sequence of characters, and an *explicit* operation is a user action that has truly occurred to the shared file, modifying some of its characters. In detail, an explicit operation O consists of seven properties:

- There are two possible *operation types*: `insert` and `delete`; *O.type* represents the operation type of O.

- The *targeted string* is the string that will be inserted or deleted by O, which is denoted by *O.str*.

- The *length* of O is the (character) length of *O.str*, which is denoted by *O.len*.

- The *position* of O is where *O.str* will be inserted to or deleted from in the shared file, which is denoted by *O.pos*.

- O must be performed on a context (file version), which is called the *base context* of O, or denoted as *O.bc*.

- O is performed on *O.bc* to generate a new context, which is called the *result context* of O, or denoted as *O.rc*.

- The range of characters impacted by O in *O.bc* is the *impact region* of O, denoted as *O.ir*. It is calculated as:
$$O.ir = \begin{cases} [O.pos, O.pos+1) & \text{if } O.type = \text{insert}; \\ [O.pos, O.pos+O.len) & \text{if } O.type = \text{delete}. \end{cases}$$

This formula tells that when a string is inserted to *O.bc*, the insert operation only affects the position (in *O.bc*) where the string is inserted; but when a string is deleted from *O.bc*, the positions where all the characters of the string formerly appear at *O.bc* are affected.

Automatically acquiring a user's explicit operations is trivial and lightweight when the editor can be monitored, *e.g.,* in Google Docs [7] and Overleaf [15]. In these systems, users are required to use a designated online file editor, by monitoring which all the collaborators' explicit operations can be directly captured in real time.

In contrast, our studied cloud storage services are supposed to work independently with any editors and support any types of text files, thus bringing great convenience to their users

(especially non-technical users). Therefore, we do not attempt to monitor any editors or impose any restrictions on the file types, and thus cloud storage services cannot capture users' *explicit operations*. Instead, we choose to analyze users' *implicit operations* based on the numerous file versions hosted at the cloud side. For a shared file f, implicit operations represent the cloud-perceived content changes to f (*i.e.,* the eventual result of a user's editing actions), rather than the user's editing actions that have actually happened to f. Obviously, implicit operations, as well as their various properties, have to be indirectly *inferred* from the different versions of f. Since we focus on implicit operations in this work, we simply use "operations" to denote "implicit operations" hereafter.

3.3 Operation Inference (OI)

When no conflict happens, inferring the operations from two consecutive versions of a file is intuitive, so in this part we only consider the OI when two conflicting versions emerge at the cloud. Note that when there are more than two conflicting versions, our described algorithms below still apply.

When two conflicting versions of a file, V_1 and V_2 (of n_1 and n_2 bytes in length) are uploaded to the cloud by two collaborators, the cloud first pinpoints their latest *common-context* version V_0 (of n_0 bytes in length) hosted in the cloud. Generally, the cloud knows which version is consistent with a collaborator's local copy during her last connection to the cloud. When the collaborator uploads a new version, this "consistent" version is regarded as the base context (version) of the new version, so that all versions of a shared file constitute a *version tree*, in which the parent of a version is its base context. Therefore, to pinpoint V_0 is to find the latest common ancestor of V_1 and V_2 in the version tree.

After pinpointing V_0, the cloud starts to infer the operation sequences (S_1 and S_2) that change V_0 to V_1 and V_2, respectively. To infer S_1, the common approach is to first find the longest common subsequence (LCS) between V_0 and V_1 using *dynamic programming* [33, 44, 57]. Then, by comparing the characters in V_0 and the LCS one by one, a sequence of delete operations can be acquired, which changes V_0 to the LCS; in a similar manner, a sequence of insert operations that changes the LCS to V_1 can be acquired. After that, the acquired delete and insert operations are combined to constitute S_1 (S_2 is constituted in a similar manner). Unfortunately, this common approach requires $O(n_0 * n_1)$ computation complexity, which may require considerable time for a large file, *e.g.,* ~30 seconds for a 500-KB file.

To address this problem, we leverage an *edit graph* [4, 55] to organize V_0 and V_1. Figure 4 exemplifies how to calculate the LCS between two words "properly" (V_0, on the horizontal axis) and "purple" (V_1, on the vertical axis) using an edit graph, where a diagonal edge has weight 0 and a horizontal or vertical edge has weight 1. Accordingly, finding the LCS between V_0 and V_1 is converted to finding a minimum-cost

path that goes from the start point (*i.e.,* $(0,0)$ in Figure 4) to the end point (*i.e.,* $(8,6)$ in Figure 4). With an edit graph, the problem can be solved with $O((n_0 + n_1) * d)$ complexity [55], where $d = n_0 + n_1 - 2l$ is the number of horizontal and vertical edges (*i.e.,* the length of difference between V_0 and V_1) and l is the number of diagonal edges (*i.e.,* the length of the LCS). Note that d is usually much smaller than n_0 and n_1 in practice: in our collected traces, the median and mean values of $\frac{d}{n_0+n_1}$ are merely 0.12% and 2.19%. Thus, the cloud can infer S_1 and S_2 efficiently using the edit graph, *e.g.,* for a 500-KB file the inference time is typically optimized from ~30 seconds to ~200 ms, resulting in a 150× reduction.

3.4 Operational Transformation (OT)

After the operation sequences S_1 and S_2 are inferred, which contain s_1 and s_2 operations respectively (all operations in a sequence are sorted by their position and have the same base context V_0), the cloud first detects whether there exist true conflicts, and then constructs a *conflict graph* [53] (as shown in Figure 5) if there are any. A conflict graph is a directed acyclic graph that has $s_1 + s_2$ vertices representing the aforementioned $s_1 + s_2$ operations. After that, *operation transformation* (OT) [39] is adopted to transform and merge S_1 and S_2 into a result sequence S_r, which can be executed on V_0 to generate the merged file version $V_{1,2}$.

Detecting true conflicts. In order to detect true conflicts between S_1 and S_2, the cloud first merges S_1 and S_2 into a temporary sequence S_{temp} sorted by the operations' position, and initializes the conflict graph G with $s_1 + s_2$ vertices and 0 edges. Then, for each operation in S_{temp}, the cloud checks whether the operations behind it conflict with it – this is achieved by checking whether the impact regions of two operations overlap each other. If two operations $S_{temp}[i]$ and $S_{temp}[j]$ are real conflicting operations, an edge $e_{i,j}$ connecting v_i to v_j (denoted by solid arrows in Figures 5a and 5b) is added to G to represent a true conflict. If there are no true conflicts between any two operations, G is useless and simply discarded. The detection, in the worst case (where each operation in S_1 conflicts with each operation in S_2), bears $O((s_1 + s_2)^2)$ complexity. However, in common cases there exist only a few conflicts, and thus the detection can be quickly carried out with $O(s_1 + s_2)$ complexity.

Basics of OT. As the *de facto* technique for conflict resolution in distributed collaboration, OT [39] has been well studied [40, 61] and used (*e.g.,* Google Docs [7], Overleaf [15], Wave [24], and Etherpad [6]). It resolves conflicts by transforming parallel operations on a shared file to equivalent sequential operations (if possible). A very simple example of OT is shown in Figure 6. More details and examples of OT can be found at https://UFC2.github.io

OT when there are no true conflicts. According to our detection results on the ten collaboration traces (cf. Table 2),
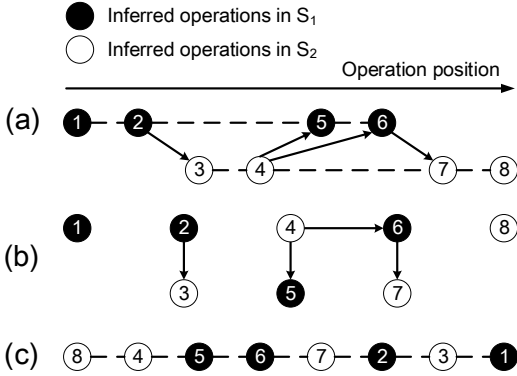
Figure 5: Reordering conflicting operations with a conflict graph. (a) In the two operation sequences S_1 and S_2, a dashed line denotes a sequence, while a solid arrow represents a true conflict. (b) S_1 and S_2 are reorganized into a conflict graph, where conflicting operations are linked with directed edges. (c) In the result sequence S_r, operations are sorted by their topological order in the conflict graph.
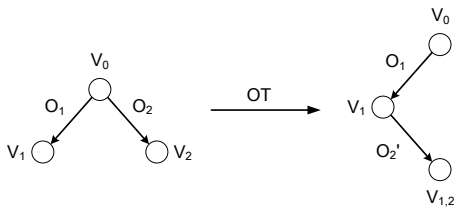
Figure 6: An example of OT that merges V_1 and V_2, in which O_2 is transformed to O_2' to resolve the conflict between O_1 and O_2.

when a file-level conflict occurs there are no true conflicts with a very high (>95%) probability, which is consistent with the results of our manual examination in §3.1. When there are no true conflicts detected, the cloud directly applies OT on S_1 and S_2 to generate S_r and $V_{1,2}$. Traditionally, the computation complexity of OT is deemed as $O((s_1 + s_2)^2)$. In our case, since there are no true conflicts and S_{temp} are already sorted by the operations' position, we choose to transform the operations in S_{temp} in their descending order of position, thus achieving a much lower complexity of $O(s_1 + s_2)$. After the transformation, we get S_r and execute S_r on V_0 to generate the merged version.

OT in the presence of true conflicts. If there are true conflicts detected, it is impossible to directly and correctly resolve the conflicts as in the above case. Consequently, we choose to prioritize the mitigation of user intervention while preserving potentially useful information, so as to facilitate users' manual conflict resolution. Specifically, two principles should be followed: 1) the cloud should send only one merged version $V_{1,2}$ to the collaborators for easy manual conflict resolution; and 2) users' editing intentions should be retained as much as possible, while the number of conflicts that have to be manually resolved in $V_{1,2}$ had better be minimized.

To realize the two principles, our first step is to utilize *topological sorting* [46] to reorganize and help transform S_1

and S_2 (via their conflict graph G) following two rules. First, real conflicting operations should be transformed and put into S_r in the ascending order of their position, so that their conflicts can be resolved at one time and thus do not negatively impact the transformation of other operations. Second, non-conflicting operations should be put into S_r in the descending order of their position, so that they can be quickly transformed like in the case of no true conflicts.

After S_1 and S_2 are topologically sorted and put into S_r (see Figure 5c), we apply our customized OT scheme to embody the aforementioned two principles for resolving true conflicts. First of all, we classify true conflicts into different categories that are suited to different processing strategies. Given two conflicting operations O_1 and O_2 working on the same base context (V_0), there seem to be four different categories of conflicts in the form of "$O_1.type/O_2.type$": 1) `delete/delete`, 2) `delete/insert`, 3) `insert/delete`, and 4) `insert/insert`. Here "/" means $O_1.pos \leq O_2.pos$. However, by carefully examining the impact regions of O_1 and O_2 ($O_1.ir$ and $O_2.ir$) in each category, we find that `insert/delete` conflicts are never true conflicts, because an insert operation only affects the targeted string at the position it appears, and never affects a to-be-deleted string that starts behind this position. Thus, we only need to deal with the other three categories as follows.

- For a `delete/delete` conflict, all the characters deleted by the users (say, Alice and Bob) are $O_1.str \cup O_2.str$. To retain both users' editing intentions as much as possible, we choose to delete only the characters both users want to delete (*i.e.*, $O_1.str \cap O_2.str$), while preserving the other characters with related information. For example, let $V_0 =$ "We need foods, water, clothes, and books."; O_1 made by Alice is to `delete` "foods, water, " at position 8, whereas O_2 made by Bob is to `delete` "water, clothes, " at position 15. In this case, O_1 is transformed to `insert` "[Alice delete:foods,]" at position 8, and O_2 is transformed to `insert` "[Bob delete:clothes,]" at position 30 (= 8+ the length of "[Alice delete:foods,]"). After the two transformed operations are executed on V_0, the merged version $V_{1,2}$ is "We need [Alice delete:foods,][Bob delete:clothes,]and books." This is not a perfect result, but is pretty easy to be manually resolved by Alice and Bob.

- For a `delete/insert` conflict, we notice that the characters deleted by Alice might be the literal context of the characters inserted by Bob. Thus, the deleted characters should be preserved to facilitate (mostly Bob's) manual conflict resolution. For example, let $V_0 =$ "There is a cat in the courtyard."; O_1 is to `delete` " in the courtyard" at position 14, changing V_0 to V_1 ("There is a cat."), whereas O_2 is to `insert` "spacious " at position 22, changing V_0 to V_2 ("There is a cat in the spacious courtyard."). Without appropriate transformation, the merged version is "There is a catspacious .", which is obviously confusing. In this

Trace	# File Versions	# Conflicting Versions	# MV Conflicts	# Conflicts	# True Conflicts	Reduction of Conflicts
Dropbox-1	3527	154	8	501	9	98.2%
Dropbox-2	2193	104	12	257	5	98.1%
OneDrive	2673	109	10	284	7	97.5%
iCloud Drive	3211	133	9	402	8	98.0%
Box	2930	125	5	374	8	97.9%
SugarSync	3472	147	13	523	11	97.9%
Seafile	2823	126	11	411	9	97.8%
Spark-Github	129957	1728	133	6724	167	97.5%
TensorFlow-Github	246016	8621	845	66231	1097	98.3%
Linux-Github	901167	36048	3210	216584	2882	98.7%

Table 6: Measurement statistics when the ten collaboration traces are replayed with UFC2. "MV Conflicts" denote the conflicts of multiple versions, *i.e.,* ≥ 3 conflicting versions are generated from the same base version.

case, O_1 is split into two operations: one is to `insert` "[Alice delete: in the]" at position 14, and the other is to `insert` "[Alice delete: courtyard]" at position 37 ($= 14+$ the length of "[Alice delete: in the]"); and O_2 is transformed to `insert` "[Bob insert: spacious]" at position 37. Afterwards, $V_{1,2}$ is "There is a cat[Alice delete: in the][Bob insert: spacious][Alice delete: courtyard].", which is also imperfect but easy to be manually resolved.

- For an `insert`/`insert` conflict, except when $O_1.str = O_2.str$ (which rarely happens), we choose to preserve both $O_1.str$ and $O_2.str$ by inserting $O_2.str$ after $O_1.str$, meanwhile adding the related information. For example, let $V_0 =$ "We need foods and books." O_1 is to `insert` ", water," at position 13, whereas O_2 is to `insert` ", clothes," at the same position. In this case, $V_{1,2}$ is "We need foods, [Alice insert:water][Bob insert:clothes], and books."

3.5 Merging Conflicts of Multiple Versions

Our above-designed scheme, despite being described with a simple case of two versions, is also applicable to solving conflicts between multiple versions. Multi-version conflicts do not often happen in practice, *e.g.,* we can calculate from Table 6 that they only account for 9% of the total conflicts.

In this complex case, suppose multiple collaborators (say $n \geq 3$) simultaneously edit the same base version V_0 and then generate n conflicting versions $V_1, V_2, V_3, ... , V_n$. To resolve such conflicts, we first figure out the operation sequences (*i.e.,* $S_1, S_2, S_3, ..., S_n$) for each version using edit graphs, which represent the changes in each version relative to their common base version V_0. Afterwards, with our devised operation transformation method, all the operation sequences are merged one by one, so as to generate the result operation sequence $S_{r_{1,2,3,...,n}}$. Specifically, S_1 and S_2 are first merged to generate $S_{r_{1,2}}$, and then S_3 are merged with $S_{r_{1,2}}$ to generate $S_{r_{1,2,3}}$. This procedure is repeated until all the operation sequences are merged, resulting in $S_{r_{1,2,3,...,n}}$. Finally, similar to the simple case of two versions , $S_{r_{1,2,3,...,n}}$ is executed on V_0 to generate the final version $V_{1,2,3,...,n}$.
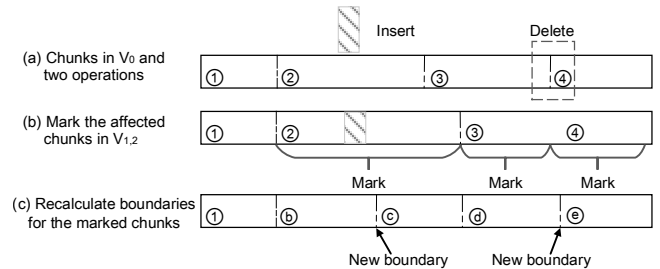
Figure 7: Boundary recalculation in OCDC. Chunk ② is split into ⓑ and ⓒ as its size exceeds the size limitation of a single chunk after the characters are added. Chunks ③ and ④ are re-partitioned as ⓓ and ⓔ as the total sizes of their remaining parts exceed the size limitation of a single chunk (otherwise, they will be combined).

3.6 Maintenance of Historic Versions

The merged version $V_{1,2}$ of a shared file, as well as the previous versions, should be kept in the cloud so that 1) users can retrieve any previous versions as they wish, and 2) the cloud can pinpoint V_0 from historic versions in future conflict resolutions. To save the storage space for hosting historic versions, we break each version into variable-sized data chunks using CDC [59] for effective chunk-level deduplication.

For a user-uploaded file version, guided by the findings in §2.2, we adopt full-file sync for small ($\leq$64 KB) files and CDC delta sync for larger files to achieve the (expected) shortest upload time. Here we adopt CDC delta sync rather than the more fine-grained `rsync` to make our delta sync strategy compatible with the aforementioned CDC-based version data organization. In other words, we allow a little extra network traffic to save expensive computation cost.

For a server-merged version $V_{1,2}$, we exploit the implicit operations inferred during the aforementioned conflict resolution to accelerate CDC, which is referred to as *operation-based CDC* (OCDC). Specifically, for each operation in the result sequence S_r, we examine whether its impact region overlaps the boundaries of any chunks of V_0 (see Figure 7 (a)); if yes, we mark the boundary (or boundaries) as "changed" (see Figure 7 (b)). After examining all operations in S_r, we use the unchanged boundaries to split $V_{1,2}$ into multiple parts, and recalculate the block boundaries of those parts that contain
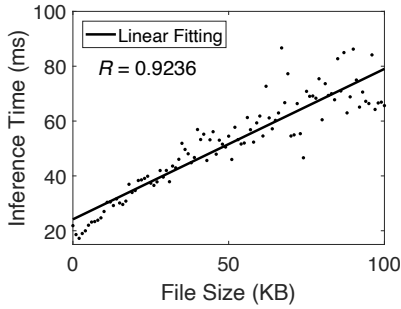
Figure 8: Time overhead incurred by our devised operation inference. Here R is the correlation coefficient between the measurements and linear fitting.
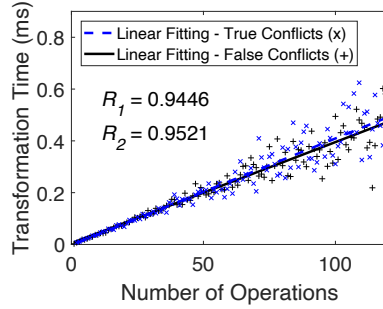
Figure 9: Time overhead incurred by our devised operation transformation. Here R_1 and R_2 are the correlation coefficients with and without true conflicts.
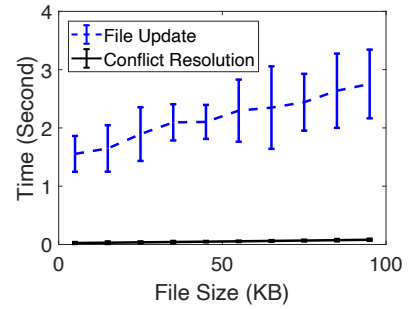
Figure 10: Total time overhead of a conflict resolution *vs.* the delivery time of a file update (using the hybrid full-file/delta sync method).

"changed" boundaries (see Figure 7 (c)). OCDC is especially effective when there is only small difference between V_1/V_2 and V_0 (which is the usual case in practice).

4 Implementation and Evaluation

To implement our design, we build a prototype system UFC2 (User-Friendly Collaborative Cloud) on top of Amazon Web Services (AWS) with 5,000 lines of Python code, and evaluate UFC2 using real-world workloads in multiple aspects.

4.1 Implementation

At the infrastructure level of UFC2, we host the (hierarchical) metadata of historic versions in Amazon EFS for efficient file system access, and the (flat) data chunks in Amazon S3 for economic content storage – note that the unit storage price of EFS ($\sim$\$0.3/GB/month) is around $10\times$ higher than that of S3 [38]. Besides, the web service of UFC2 runs on a standard VM (with a dual-core CPU @2.5 GHz, 8-GB memory, and 32-GB SSD storage) rented from Amazon EC2. Moreover, the employed EFS storage, S3 storage, and EC2 VM are located at the same data center in Northern Virginia, so there is no bottleneck among them. At the client side, we deploy puppet collaborators on geo-distributed VMs rented from Amazon EC2 to replay our collected ten real-world collaboration traces (cf. Table 2). Details of these VMs and the replay processes are the same as those described in §2.1.

4.2 Experiment Results

Ratio of conflicts resolved. Our first metric to evaluate the collaboration support of cloud storage services is the number of conflicts. We replay the ten traces with UFC2, and observe that the file versions generated by UFC2 (at the cloud side) are slightly different from those generated by Dropbox/OneDrive/iCloud Drive/Box/SugarSync/Seafile (cf. §2.2) due to the variation (*esp.,* in latency) of network environments; also, the resulting conflicts are slightly different. Notably, all the false conflicts are automatically resolved by UFC2. The

remaining conflicts are all true conflicts that should be manually resolved by the collaborators, assisted with the helpful information automatically added by UFC2. As listed in Table 6, the ratio of conflicts is reduced by 97.5%–98.7% for different traces, *i.e.,* an average reduction by 98%.

Time overhead of conflict resolution. Conflict resolution in UFC2 consists of two steps: operation inference (OI, §3.3) and operation transformation (OT, §3.4). Thus, we first examine the time overhead incurred by the two steps separately, and then analyze the total time of conflict resolution (compared to the delivery time of a file update).

First, we record the time of OI in every conflict resolution when replaying the ten traces with UFC2. The results are plotted as a scatter diagram shown in Figure 8, together with a linear fitting. The correlation coefficient (R) between the measurements and the linear fitting results is as large as 0.9236, indicating that the time of OI is generally proportional to the file size. This is because by leveraging an edit graph, we reduce the computation complexity of OI from $O(n_0 * n_1)$ to $O((n_0 + n_1) * d)$ (refer to §3.3 for the details).

Second, we record the time of OT in every conflict resolution, and find it is very small (<1 ms) compared to the time of OI. As shown in Figure 9, the time of OT is highly proportional to the number of operations; in addition, the performance is quite similar with or without true conflicts. According to §3.4, the complexity of our devised OT is $O(s_1 + s_2)$, which explains the experiment results.

Further, we calculate the total time of a conflict resolution, and record the delivery time of the corresponding file update (using the hybrid full-file/delta sync method). As shown in Figure 10, the total time of a conflict resolution is 10–80 ms, while the delivery time of a file update is 1.5–3 seconds. The former is merely 0.6%–4% (on average 2%) of the latter, showing that our conflict resolution brings negligible performance overhead to the collaboration in cloud storage.

Time overhead of OCDC *vs.* traditional CDC. We record the time spent in breaking a merged file version into data
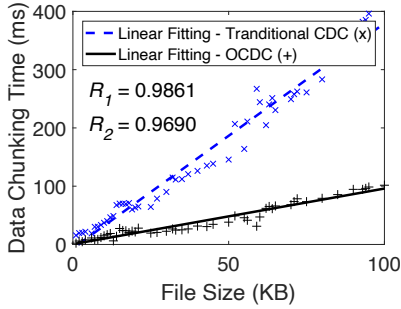
Figure 11: Data chunking time for a common file, using OCDC *vs.* traditional CDC.
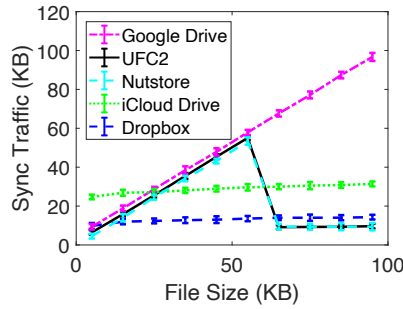
Figure 12: Sync traffic of UFC2 and representative cloud storage services for a file update when there are no file-level conflicts.
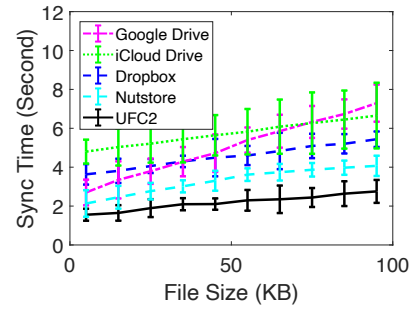
Figure 13: Sync time of UFC2 and representative cloud storage services for a file update when there are no file-level conflicts.
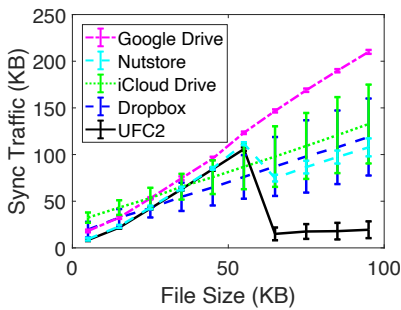
Figure 14: Sync traffic of UFC2 and representative cloud storage services for a file update when there exist file-level conflicts.
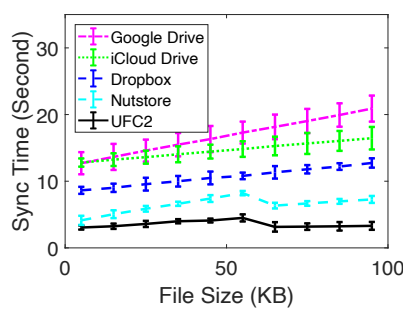
Figure 15: Sync time of UFC2 and representative cloud storage services for a file update when there exist file-level conflicts.
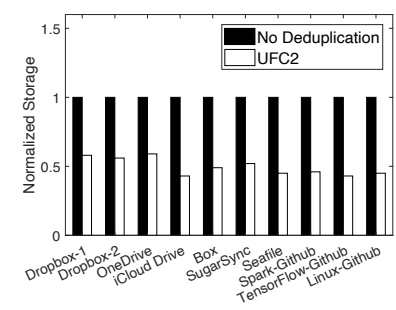
Figure 16: Normalized storage overhead of historic file versions for the ten real-world collaboration traces.

chunks with OCDC when replaying the ten traces with UFC2. For comparison, we also break the same merged file version into data chunks with traditional CDC.

As shown in Figure 11, for both OCDC and traditional CDC, the data chunking time is highly proportional to the file size. This is quite intuitive because a larger file is usually broken into more chunks. Additionally, we notice that OCDC outperforms traditional CDC by ∼3 times, reducing the data chunking time from 30–400 ms to 10–120 ms.

Network overhead. We compare the sync traffic of UFC2 with those of Dropbox, Google Drive, iCloud Drive, and Nutstore, for a file update. We only select the four cloud storage services since Dropbox, Google Drive, and iCloud Drive each represent a typical strategy for conflict resolution adopted by existing cloud storage services (*i.e.,* keep all conflicting versions, only keep the latest version, and force users to choose one version, cf. §2.2) while Nutstore is the only service that combines full-file sync and delta sync to enhance the file update speed.

As shown in Figure 12, when there are no file-level conflicts, the sync traffic of Google Drive is close to the file size, as Google Drive adopts full-file sync. In contrast, Dropbox and iCloud Drive always consume nearly 10 KB and 30 KB of sync traffic respectively due to their adoption of delta sync;

we infer that the sync granularity of Dropbox is finer than that of iCloud Drive. In contrast, Nutstore and UFC2 resemble Google Drive for small (≤64 KB) files and Dropbox for larger files, as they both adopt full-file sync for small files and delta sync for larger files to achieve the shortest sync time (see Figure 13). This hybrid sync method results in substantial savings of sync traffic for Nutstore and UFC2 after the turning point (64 KB) in Figures 12 and 14.

As shown in Figure 14, when there exist file-level conflicts, the sync traffic of Google Drive is nearly twice of the file size. This is because (the client of) Google Drive first uploads the local version, and then downloads the cloud-hosted newer version to overwrite the local version. In contrast, the sync traffic consumed by Dropbox or iCloud Drive is close to the file size; this is because the client of Dropbox (or iCloud Drive) renames one of the conflicting versions, and the renamed one is uploaded as a newly-created file using full-file sync (which usually consumes more traffic than necessary since delta sync can still be applied).

The case of Nutstore in Figure 14 is a bit complex: for small files, its sync traffic is nearly twice of the file size (similar to Google Drive); for larger files, the traffic is slightly larger than the file size (similar to Dropbox/iCloud Drive). This is because Nutstore renames one of the conflicting versions

when a file-level conflict occurs – if the file is small ($\leq$64 KB), the two files are both uploaded to the cloud using full-file sync; otherwise, the renamed file is uploaded using full-file sync (which usually consumes unnecessary traffic) whereas the original file is uploaded using delta sync.

Finally, we examine the case of UFC2 in Figure 14. Its client first uploads a conflicting version and then downloads the merged version from the cloud. For a small file, the two versions are both delivered using full-file sync, so the sync traffic is nearly twice of the file size; for a larger file, the two versions are both delivered using delta sync (which is more traffic-saving than what Nutstore does for a larger file), so the sync traffic is always as small as $\sim$20 KB. This is why UFC2 achieves the shortest sync time, as shown in Figure 15.

Storage Overhead. For the maintenance of a file's historic versions, the straightforward approach is to store all versions separately without data deduplication; its storage overhead is taken as the baseline and normalized as 1.0, as shown in Figure 16. Utilizing CDC-based deduplication, the storage overhead of UFC2 is normalized between 0.43 and 0.59 (0.49 on average) with respect to the ten traces. In comparison, the storage overhead of Google Drive is normalized as small as 0.05–0.1, because Google Drive only stores the latest version and discards all previous versions. We do not quantify the storage overhead of the other mainstream cloud storage services since we do not know their cloud-side storage organization.

5   Related Work

Various schemes have been proposed to address the collaboration conflicts in distributed file systems (DFS) and version control systems (VCSes). In this section, we survey the typical schemes and compare them to our design choices.

Conflict resolution in DFSes. LOCUS [73], Coda [47] and InterMezzo [32] mark files with unresolved conflicts as inconsistent, so that these files are inaccessible until users manually rename and merge them. These schemes prevent users from accessing the conflicting files before conflicts are resolved, and the idea of restrictive access is inherited by some recent cloud-backed file systems such as SCFS [30].

In contrast, Ficus [60] and Rumor [42] attempt to design specific conflict resolvers (using semantic knowledge of certain file types or user-made rules), so as to automatically merge conflicts of specific kinds. Bayou [69] preserves all conflicting files and allows users to access them. Similar approaches are adopted by recent large-scale systems like Dynamo [36], TierStore [37], Depot [52], and COPS [51], where all conflicting file versions are preserved, and users are forced to manually resolve all file-level conflicts. In fact, the above described strategies are also adopted (in part) by our studied popular cloud storage services.

Our work essentially differs from the aforementioned schemes by providing not only effective but also transparent and user-friendly collaboration support for replicated files in distributed environments. The desired features are enabled by our novel perspective and intelligent technical approaches in addressing the concurrent conflicts.

Conflict resolution in VCSes. Popular VCSes, such as SVN, CVS, Git, RCS [71], and SunPro [26], generally operates at a (text) line level. To resolve the conflicts between two versions of a shared file, they use delta algorithms like `bdiff` [70] and UNIX `diff` [45] to find the modified lines, which are then simply combined to form a merged version. However, if two users' modifications are made on the same line, they have to manually pick which line to retain. Recently, a more advanced approach called *structured merge* [27,28,48,75] has emerged in the software engineering community, which takes the syntactic structure of a program into account and thus can resolve very detailed conflicts happening to non-essential elements (*e.g.,* comments, tabs, and blanks) of a program. Different from VCSes' line-level or syntactic approaches that is mostly designed for developers, our work studies conflict resolution for general-purpose cloud storage services designed for regular end users.

6   Conclusion

Despite a rich body of techniques for resolving conflicts in collaborative systems [29,40,56,65,66], today's mainstream cloud storage services still use the simplest form, *i.e.,* coarse-grained file-level conflict detection and resolution. Given that collaboration has become a major use case of cloud storage services, existing mechanisms, as revealed in this paper, are deficient, inconvenient, and sometimes frustrating.

To address the issue, we make a series of efforts towards understanding and improving collaboration in cloud storage services from a novel perspective of operations without using any locks. We find that the vast majority of conflicts reported by today's cloud storage services are false conflicts, and design intelligent approaches to efficient operation inference, user-friendly operation transformation, and judicious maintenance of historic versions. We implement all the approaches in an open-source prototype system that can significantly reduce collaboration conflicts and meanwhile preserve the transparency and user-friendliness of cloud storage services.

References

[1] Average U.S. Internet Speeds More Than Double Global Average. https://www.ncta.com/whats-new/average-us-internet-speeds-more-double-global-average.

[2] Box – Secure File Sharing, Storage, and Collaboration. https://www.box.com/.

[3] Charles Web Debugging Proxy. https://www.charlesproxy.com/.

[4] Diff Match Patch is a High-performance Library in Multiple Languages that Manipulates Plain Text. https://github.com/google/diff-match-patch.

[5] Dropbox Tech Blog. https://blogs.dropbox.com/tech/.

[6] Etherpad: Really Real-time Collaborative Document Editing. https://github.com/ether/etherpad-lite.

[7] Google Docs: Free Online Documents for Personal Use. https://www.google.com/docs/about/.

[8] Google Drive privacy warning – could yours have leaked data? https://www.welivesecurity.com/2014/07/11/google-drive-privacy-warning/.

[9] Meet Bandaid, the Dropbox Service Proxy. https://blogs.dropbox.com/tech/2018/03/meet-bandaid-the-dropbox-service-proxy/.

[10] Never-Googlers: Web Users Take the Ultimate Step to Guard Their Datae. https://www.washingtonpost.com/technology/2019/07/23/never-googlers-web-users-take-ultimate-step-guard-their-data/.

[11] Nutstore – Share Your Files Anytime, Anywhere, with Any Device. https://www.jianguoyun.com/.

[12] Nutstore Help Center. http://help.jianguoyun.com/.

[13] Online Discussion – Those who refuse to use Google for privacy reasons. https://www.reddit.com/r/apple/comments/9ed07l/those_who_refuse_to_use_google_for_privacy/.

[14] Optimizing Web Servers for High Throughput and Low Latency. https://blogs.dropbox.com/tech/2017/09/optimizing-web-servers-for-high-throughput-and-low-latency/.

[15] Overleaf, Online LaTeX Editor. https://www.overleaf.com/.

[16] Seafile - Open Source File Sync and Share Software. https://www.seafile.com/en/home/.

[17] Seafile Source Code. https://github.com/haiwen/seafile.

[18] Simultaneous Collaborative Editing of a LaTeX File (Online Forum Discussion). https://tex.stackexchange.com/questions/27549/simultaneous-collaborative-editing-of-a-latex-file.

[19] Speedtest Global Index – Global Speeds August 2019. https://www.speedtest.net/global-index.

[20] SugarSync – Cloud File Sharing, File Sync & Online Backup From Any Device. https://www2.sugarsync.com/.

[21] SugarSync Help Center. https://support.sugarsync.com/hc/en-us/.

[22] The SugarSync Blog. https://www.sugarsync.com/blog/.

[23] Tool for the (collaborative) job. https://blogs.ams.org/phdplus/2016/09/11/tool-for-the-collaborative-job/.

[24] Wave | Real-time Collaboration and Coediting Service. https://www.codox.io/.

[25] Wireshark Network Protocol Analyzer. http://www.wireshark.org/.

[26] E. Adams, W. Gramlich, S. S. Muchnick, and S. Tirfing. SunPro: Engineering a Pratical Program Development Environment. In *Proceedings of International Workshop on Advanced Programming Environments*, pages 86–96. Springer-Verlag, 1986.

[27] S. Apel, O. Leßenich, and C. Lengauer. Structured Merge with Auto-tuning: Balancing Precision and Performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 120–129. ACM, 2012.

[28] T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A Differencing Technique and Tool for Object-oriented Programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[29] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski. Specification and Complexity of Collaborative Text Editing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 259–268. ACM, 2016.

[30] A. Bessani, R. Mendes, T. Oliveira, et al. SCFS: A Shared Cloud-backed File System. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 169–180, 2014.

[31] R. Bhargava. Evolution of Dropbox's Edge Network, 2017. https://blogs.dropbox.com/tech/2017/06/evolution-of-dropboxs-edge-network/.

[32] P. Braam, M. Callahan, P. Schwan, et al. The InterMezzo File System. In *Proceedings of the 3rd of the Perl Conference, O'Reilly Open Source Convention*, 1999.

[33] G. Canfora, L. Cerulo, and M. Di Penta. Ldiff: An Enhanced Line Differencing Tool. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 595–598. IEEE Computer Society, 2009.

[34] Z. Chen, Q. He, Z. Mao, H.-M. Chung, and S. Maharjan. A Study on the Characteristics of Douyin Short Videos and Implications for Edge Caching. In *Proceedings of the ACM Turing Celebration Conference - China (TURC)*, page 13:1–13:6. ACM, 2019.

[35] Y. Cui, N. Dai, Z. Lai, M. Li, Z. Li, Y. Hu, K. Ren, and Y. Chen. Tailcutter: Wisely Cutting Tail Latency in Cloud CDNs under Cost Constraints. *IEEE/ACM Transactions on Networking*, 27(4):1612–1628, 2019.

[36] 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 *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220. ACM, 2007.

[37] M. Demmer, B. Du, and E. Brewer. TierStore: A Distributed Filesystem for Challenged Networks in Developing Regions. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 35–48. USENIX, 2008.

[38] J. E, Y. Cui, M. Ruan, Z. Li, and E. Zhai. HyCloud: Tweaking Hybrid Cloud Storage Services for Cost-efficient Filesystem Hosting. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, pages 1–9. IEEE, 2019.

[39] C. Ellis and S. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 399–407. ACM, 1989.

[40] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 337–350. USENIX, 2010.

[41] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2015.

[42] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile Data Access Through Optimistic Peer-to-peer Replication. In *Advances in Database Technologies*, pages 254–265. Springer, 1999.

[43] B. Hou and F. Chen. GDS-LC: A Latency-and Cost-aware Client Caching Scheme for Cloud Storage. *ACM Transactions on Storage*, 13(4):40, 2017.

[44] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta Algorithms: An Empirical Analysis. *ACM Transactions on Software Engineering and Methodology*, 7(2):192–214, 1998.

[45] J. W. Hunt and M. D. MacIlroy. *An Algorithm for Differential File Comparison*. Bell Laboratories Murray Hill, 1976.

[46] A. B. Kahn. Topological Sorting of Large Networks. *Communications of the ACM*, 5(11):558–562, Nov. 1962.

[47] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 213–225. ACM, 1991.

[48] O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund. Renaming and Shifted Code in Structured Merging: Looking Ahead for Precision and Performance. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 543–553. IEEE, 2017.

[49] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang. Towards Network-level Efficiency for Cloud Storage Services. In *Proceedings of the Conference on Internet Measurement Conference (IMC)*, pages 115–128. ACM, 2014.

[50] G. Liang and U. C. Kozat. Fast Cloud: Pushing the Envelope on Delay Performance of Cloud Storage with Coding. *IEEE/ACM Transactions on Networking*, 22(6):2012–2025, 2014.

[51] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 401–416. ACM, 2011.

[52] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud Storage with

Minimal Trust. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 307–322. USENIX, 2010.

[53] D. Marx. Graph Colouring Problems and Their Applications in Scheduling. *Periodica Polytechnica Electrical Engineering (Archives)*, 48(1-2):11–16, 2004.

[54] T. Mens. A State-of-the-art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[55] E. W. Myers. An *O(ND)* Difference Algorithm and Its Variations. *Algorithmica*, 1(1):251–266, Nov. 1986.

[56] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the Annual ACM Symposium on User Interface and Software Technology (UIST)*, pages 111–120. ACM, 1995.

[57] Y. S. Nugroho, H. Hata, and K. Matsumoto. How Different Are Different Diff Algorithms in Git? *Empirical Software Engineering*, pages 1–34, 2019.

[58] S. Perez De Rosso and D. Jackson. What's Wrong with Git?: A Conceptual Design Analysis. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, pages 37–52. ACM, 2013.

[59] C. Policroniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 73–86. USENIX, 2004.

[60] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving File Conflicts in the Ficus File System. In *Proceedings of the USENIX Summer Technical Conference*, pages 183–195. USENIX, 1994.

[61] B. Shao, D. Li, T. Lu, and N. Gu. An Operational Transformation Based Synchronization Protocol for Web 2.0 Applications. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*, pages 563–572. ACM, 2011.

[62] C. Smith. 33 Staggering Dropbox Statistics and Facts (2019) | By the Numbers, 2019. https://expandedra mblings.com/index.php/dropbox-statistics/.

[63] M. Sousa, I. Dillig, and S. K. Lahiri. Verified Three-way Program Merge. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[64] Y. Su, D. Feng, Y. Hua, and Z. Shi. Understanding the Latency Distribution of Cloud Object Storage Systems. *Journal of Parallel and Distributed Computing*, 128:71–83, 2019.

[65] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, Mar. 1998.

[66] D. Sun and C. Sun. Context-Based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 20(10):1454–1470, Oct. 2009.

[67] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 513–527, 2015.

[68] D. Svantesson and R. Clarke. Privacy and Consumer Risks in Cloud Computing. *Computer law & security review*, 26(4):391–397, 2010.

[69] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 172–182. ACM, 1995.

[70] W. F. TICHY. The String-to-String Correction Problem with Block Moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[71] W. F. Tichy. RCS – A System for Version Control. *Software: Practice and Experience*, 15(7):637–654, 1985.

[72] A. Tridgell and P. Mackerras. The Rsync Algorithm. Technical report, 1996. https://openresearch-repository.anu.edu.au/bit stream/1885/40765/3/TR-CS-96-05.pdf.

[73] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 49–70. ACM, 1983.

[74] Z. Wu, C. Yu, and H. V. Madhyastha. Costlo: Cost-effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 543–557, 2015.

[75] F. Zhu and F. He. Conflict Resolution for Structured Merge via Version Space Algebra. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):166, 2018.

POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database

Wei Cao[†], *Yang Liu*[‡], *Zhushi Cheng*[†], *Ning Zheng*[‡], *Wei Li*[†], *Wenjie Wu*[†], *Linqiang Ouyang*[‡],
Peng Wang[†], *Yijing Wang*[†], *Ray Kuan*[‡], *Zhenjun Liu*[†], *Feng Zhu*[†], *Tong Zhang*[‡]

[†] *Alibaba Group, Hang Zhou, Zhejiang, China*
[‡] *ScaleFlux Inc., San Jose, CA, USA*

Abstract

This paper reports the deployment of computational storage drives in Alibaba Cloud to enable cloud-native relational database cost-effectively support analytical workloads. With its compute-storage decoupled architecture, cloud-native relational database should pushdown data-intensive tasks (e.g., table scan) from front-end database nodes to back-end storage nodes in order to adequately support analytical workloads. This however makes it a challenge to maintain the cost effectiveness of storage nodes. The emerging computational storage opens a new opportunity to address this challenge: By replacing commodity SSDs with computational storage drives, storage nodes can leverage the in-storage computing power to much more efficiently perform table scans. Practical implementation of this simple idea is non-trivial and demands cohesive innovations across the software (i.e., database, filesystem and I/O) and hardware (i.e., computational storage drive) layers. This paper presents such a holistic implementation for Alibaba cloud-native relational database POLARDB. To the best of our knowledge, this is the first real-world deployment of cloud-native databases with computational storage drives ever reported in the open literature.

1 Introduction

Relational database is an essential building block in modern information technology infrastructure. Therefore, all the cloud vendors have invested significant efforts to grow their relational database service (RDS) business. Not surprisingly, some cloud vendors have developed their own cloud-native relational database systems, e.g., Amazon Aurora [28] and Alibaba POLARDB [9]. In order to achieve sufficient scalability and fault resilience, cloud-native relational databases naturally follow the design principle of decoupling compute from data storage [4, 17]. Meanwhile, they typically aim to be compatible with mainstream open-source relational databases (e.g., MySQL and PostgreSQL) and achieve high performance for OLTP (online transaction processing) workloads at a much lower cost than their on-premise counterparts.

It is highly desirable for cloud-native relational databases to adequately support analytical workloads. As pointed out by the authors of [28], because cloud-native relational databases decouple compute from data storage, the network bandwidth between database nodes and storage nodes becomes a scarce resource. This however does not match well to analytical workloads that involve intensive data access. To best serve OLTP workloads, cloud-native relational databases typically employ the row-store model (or the hybrid-row/column model [5]). This could make the network bandwidth an even bigger bottleneck for analytical workloads. In order to better serve analytical workloads, the almost only viable option is to off-load data-access-intensive tasks (in particular table scan) from database nodes to storage nodes. This concept is certainly not new and has been adopted by both proprietary database appliances (e.g., Oracle Exadata) and open-source databases (e.g., MySQL NDB Cluster). In spite of the simple concept, its practical implementation in the context of cloud-native databases is particularly non-trivial. On one hand, each storage node must be equipped with sufficient data processing power to handle table scan tasks. On the other hand, to maintain the cost effectiveness of cloud-native databases, we cannot significantly (or even modestly) increase the cost of storage nodes. By complementing CPUs with special-purpose hardware (e.g., GPU and FPGA), heterogeneous computing architecture appears to be an appealing option to address this data processing power vs. cost dilemma.

This work applies heterogeneous computing in POLARDB storage nodes to efficiently support table scan pushdown. The key idea is simple: Each POLARDB storage node off-loads and distributes table scan tasks from its CPU to its data storage devices. Under this framework, each data storage device becomes a *computational storage drive* [1] that can carry out table scan on the I/O path. Compared with off-loading table scan to a dedicated stand-alone computing device (e.g., FPGA/GPU-based PCIe card), distributing table scan across all the storage drives can minimize the data traffic across the storage/memory hierarchy and obviate data processing hot-spot. This simple concept is not new and has been discussed

---

(e.g., see [11, 14]). However, its practically viable implementation and real-world deployment remain completely missing, at least in the open literature. This is mainly due to the difficulty of addressing two challenges: (1) how to practically support the table scan pushdown across the entire software hierarchy, and (2) how to implement low-cost computational storage drives with sufficient table scan processing capability.

Over the course of materializing this simple idea in the context of POLARDB on Alibaba Cloud, we developed a set of software/hardware techniques to cohesively address the two challenges. To reduce the product development cycle and meanwhile ensure cost effectiveness, computational storage drives use an FPGA-centric host-managed architecture. Inside each computational storage drive, a single mid-range low-cost Xilinx FPGA chip handles both flash memory control and table scan. With highly optimized software and hardware design, each computational storage drive can support high-throughput (i.e., over 2GB/s) table scan on compressed data and meanwhile achieve storage I/O performance comparable to leading-edge NVMe SSDs. We developed a variety of techniques that enable POLARDB storage nodes fully exploit the capability of computational storage drives. This paper presents these design techniques and elaborates on their implementation, and further presents evaluation results to demonstrate their effectiveness. Based on the TPC-H queries, we extracted six individual table scan tasks and ran these scan tasks on one storage node. Such node-level evaluation shows that the computational storage drives can largely reduce both scan latency and CPU utilization of the storage node. We further carried out system-level evaluations on a POLARDB cloud instance over 7 database nodes and 3 storage nodes. Results show that this solution can noticeably reduce the TPC-H query latency. To the best of our knowledge, this is the first application of emerging computational storage in production database ever reported in the open literature.

2 Background and Motivation

2.1 POLARDB: Basic Architecture

POLARDB is a new cloud-native OLTP database designed by Alibaba Cloud. Its design goals come from our cloud customers' real needs: large per-instance storage capacity (tens of TB), high TPS (transactions per second), high and scalable QoS and high availability. POLARDB provides enterprise-level cloud database services and is compatible with MySQL and PostgreSQL. Fig. 1 illustrates the compute-storage decoupled architecture of Alibaba POLARDB. Database computing nodes and storage nodes are connected through high-speed RDMA network. In each POLARDB instance, there is only one read/write database node that handles both the read and write requests, and the other database nodes handle only read requests. All the nodes in an instance, including read/write nodes and read-only nodes, are able to access the same copy of data on a storage node. To ensure the high availability, POLARDB uses the Parallel-Raft protocol to write three copies of data across the storage nodes [9].
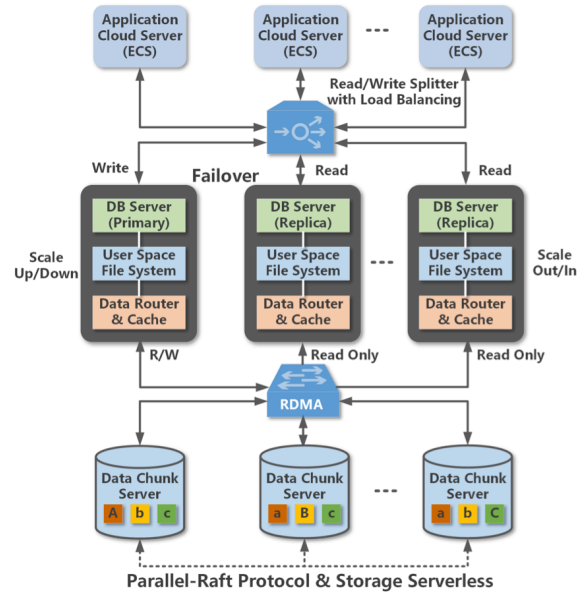
Figure 1: Illustration of POLARDB architecture.

2.2 POLARDB: Table Scan Pushdown

Off-loading table scan from database nodes to storage nodes is important for cloud-native relational database to effectively handle analytical workloads. This concept trades heavier data processing load on storage nodes for significantly reduced network traffic between database nodes and storage nodes. Moreover, since POLARDB employs the row-store model to better serve OLTP workloads, the column-oriented nature of table scan tends to demand even higher data processing power in storage nodes. Therefore, the key design issue is how to cost-effectively equip storage nodes with sufficient data processing power to handle the additional table scan tasks.

The most straightforward option is to simply scale up each storage node, which nevertheless is not practically desirable mainly due to the cost overhead. Table scan over row-store data does not fit well to modern CPU architecture and tends to largely under-utilize CPU hardware resources (e.g., cache memory, and SIMD processing resource) [2]. As a result, we have to more aggressively scale up the storage nodes to compensate for the inefficiency of CPU-based implementation. Hence, this straightforward option is economically unappealing and even unacceptable, especially as the classical CMOS technology scaling is quickly approaching its end [8].

An alternative is to complement storage node CPUs with special-purpose hardware (e.g., FPGA or GPU) that can carry out table scan with much better cost effectiveness. Under

this heterogeneous computing framework, the conventional practice uses a *centralized* heterogeneous architecture where the special-purpose hardware is implemented in the form of a single stand-alone FPGA/GPU-based PCIe card (e.g., see [24, 26, 29]). Nevertheless, this approach has several drawbacks for our targeted systems: (1) *High data traffic*: All the raw data in their row-store format must be fetched from the storage devices into the FPGA/GPU-based PCIe card. Due to the data-intensive nature of table scan, this leads to a very heavy data traffic over the PCIe/DRAM channels. The high data traffic can cause significant energy consumption overhead and inter-workload interference. (2) *Data processing hot-spot*: Each storage node contains a large number of NVMe SSDs, each of which can achieve multi-GB/s data read throughput. As a result, analytical processing workloads could trigger very high aggregated raw data access throughput that is far beyond the I/O bandwidth of one PCIe card. This could make the FPGA/GPU-based PCIe card become the system bottleneck.

The above discussion suggests that a *distributed* heterogeneous architecture is a better option. As illustrated in Fig. 2, by distributing table scans directly into each storage drive, we can eliminate the high data traffic over the PCIe/DRAM channels, and obviate data processing hot-spot in the system. This intuition directly motivated us to develop and deploy computational storage drives in POLARDB storage nodes.

2.3 Computational Storage Drive

Loosely speaking, any data storage device that can carry out data processing tasks beyond its core storage duty can be called a computational storage drive. The simple concept of empowering storage devices with additional computing capability can trace back to over 20 years ago [3, 21, 22]. Computational storage complements with CPU to form a heterogeneous computing system. Compared with its CPU-only counterpart, a heterogeneous computing system not surprisingly can achieve higher performance and/or energy efficiency for many applications, as demonstrated by prior research (e.g., see [10,11,15,16,18,23,27]). However, it is apparently subject to two cost overheads: (1) the hardware cost of implementing computational storage drives, and (2) the development cost on developing all the necessary hardware and software solutions to enable its real-world deployment. In spite of the over two decades of research, computational storage has not yet entered the mainstream market, arguably because of the absence of a practically justifiable benefit vs. cost trade-off.

To overcome the cost barrier, we chose an FPGA-based host-managed computational storage drive design strategy. This can reduce the development cost from two aspects: (1) We use a single FPGA to realize both flash memory control and computation (i.e., table scan in this work) inside computational storage drives. Compared with ASIC-based approach, the circuit-level programmability of FPGA can significantly reduce the computational storage drive development cycle and

(a) Centralized heterogeneous computing architecture

(b) Distributed heterogeneous computing architecture
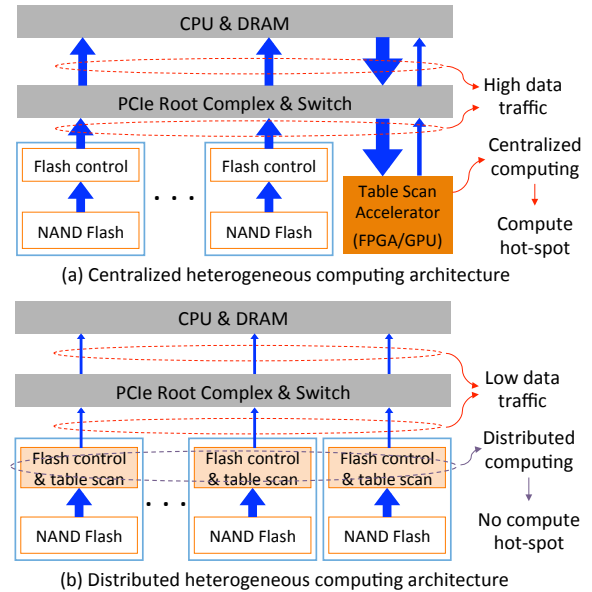
Figure 2: Illustration of (a) centralized heterogeneous computing architecture, and (b) distributed heterogeneous computing architecture.

cost. (2) The computational storage drive is fully managed by the host for the functions such as address mapping, request scheduling, and garbage collection. Its host-management nature can facilitate integrating computational storage drive into existing software stack. It enables a high flexibility to devise and optimize the computational storage drive's API through which applications can utilize its configurable computation capability. Meanwhile, the host-managed computational storage drive natively integrates into the Linux I/O stack as a storage block device to serve normal I/O requests.

However, in return for its circuit-level programmability, FPGA is expensive (e.g., modern high-end FPGA chip could cost few thousand dollars), leading to a higher hardware cost of computational storage drive. Meanwhile, the objective of this work is to deploy computational storage drive to cost-effectively support table scan pushdown. Therefore, one key issue is how to minimize the hardware cost overhead while achieving sufficiently high storage I/O and table scan processing performance, which will be discussed in the next section.

3 Design and Implementation

As pointed out above, although applying computational storage to support table scan pushdown is a very simple concept and has been well discussed in the open literature, its real-world implementation and deployment has remained missing. Our first-hand experience of implementing this concept for POLARDB reveals that transferring this simple idea into real product faces the following two major challenges:

1. *Support table scan pushdown across the entire software hierarchy*: Table scan pushdown is initiated by the user-space POLARDB storage engine that accesses data by specifying the offsets in files, while table scan is physically served by computational storage drive that operates as a raw block device and manages data with LBA (logical block address). The entire storage I/O stack sits in between POLARDB storage engine and computational storage drive. Hence, we have to cohesively enhance/modify the entire software/driver stack in order to create a path in support of table scan pushdown.

2. *Implement low-cost computational storage drive*: As discussed above in Section 2.3, although the FPGA-based design approach can significantly reduce the development cost, FPGA tends to be expensive. Moreover, since FPGA typically operates at only 200~300MHz (in contrast to 2~4GHz CPU clock frequency), we have to employ a large degree of circuit-level implementation parallelism (hence more silicon resource) in order to achieve sufficiently high performance. Therefore, we must develop solutions to enable the use of low-cost FPGA chip in our implementation.

The remainder of this section presents a set of design techniques across the software and hardware stacks that can address the above two major challenges.

3.1 Support Table Scan Pushdown Across the Entire Software Stack

To tackle the first challenge, we developed techniques to support the table scan pushdown across the entire software stack, as illustrated in Fig. 3. POLARDB database nodes incorporate a front-end analytical processing engine called *POLARDB MPP*. Being compatible with the MySQL protocol, this analytical processing engine can parse, optimize and rewrite SQL using the AST (abstract syntax tree) and a number of embedded optimization rules. It transforms each SQL query into a DAG (directed acyclic graph) execution plan consisting of operators and data flow topology. This analytical processing engine natively supports table scan pushdown to the underlying storage engine. Hence, we can keep the analytical processing engine intact in this work.

As illustrated in Fig. 3, in order to enable table scan pushdown, we have to appropriately enhance the entire storage stack underneath the analytical processing engine, including POLARDB storage engine, PolarFS (a distributed filesystem under POLARDB), and computational storage driver. In the following, we will elaborate on the implemented enhancements across these three layers.

3.1.1 Enhancement to POLARDB Storage Engine

POLARDB database storage engine follows the design principle of LSM-tree (log-structured merge-tree) [20]. Data in
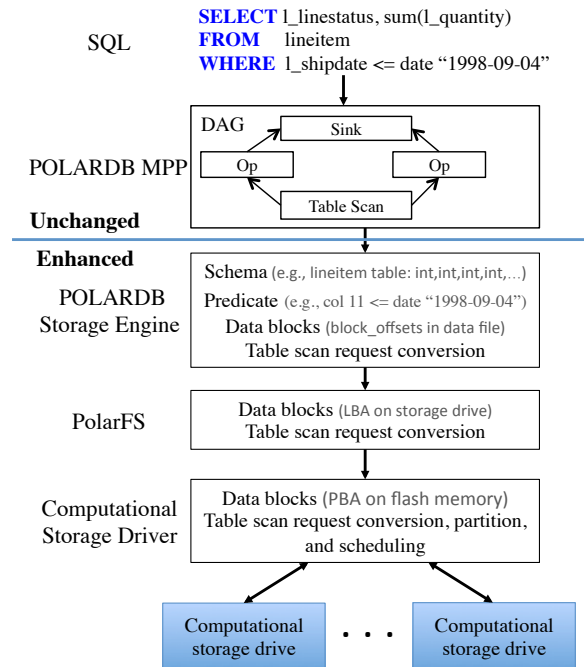
Figure 3: Illustration of the overall software stack.

each table are organized into many files (typical file size is few tens of MBs), and each file contains a large number of blocks (typical block size ranges from 4KB to 32KB). In its original implementation, POLARDB storage engine serves the table scan requests using the CPUs on storage nodes. Hence, the underlying storage I/O stack is oblivious to the table scan pushdown. Since this work aims to utilize computational storage drives to process table scan, we have enhanced POLARDB storage engine so that it can pass table scan requests to the underlying filesystem PolarFS. As illustrated in Fig. 3, storage engine accesses data blocks in terms of offsets in files. Each table scan request contains: (1) the location (i.e., offsets in files) of the to-be-scanned data, (2) the schema of the table onto which the table scan is applied, and (3) the table scan conditions to be evaluated. Meanwhile, POLARDB storage engine allocates a memory buffer for storing data returned from computational storage drives, and each table scan request contains the location of this memory buffer.

As discussed later, the implemented computational storage drives do not support all the possible scan conditions (e.g., *LIKE* is not supported in current implementation). Hence, upon receiving table scan pushdown from the analytical processing engine, the enhanced storage engine first analyzes the scan conditions, and if necessary it extracts and passes a subset of the scan conditions that can be served by the computational storage drives. After receiving the data returned from the computational storage drives, the storage engine always checks the data against the complete table scan conditions. Moreover, to improve the overall system efficiency, we should

exploit the computational parallelism across multiple computational storage drives within each storage node. Therefore, POLARDB storage engine is able to issue multiple table scan requests concurrently to the underlying computational storage devices through PolarFS.

3.1.2 Enhancement to PolarFS

As described in [9], POLARDB is deployed on the distributed filesystem PolarFS that manages the data storage across all the storage nodes. Each computational storage drive can only perform table scan on its own data and meanwhile data are scanned in the unit of storage engine data blocks. Meanwhile, due to the use of block-level compression, variable-length compressed blocks are contiguously packed in each file (i.e., each compressed block is not 4KB-aligned). Therefore, PolarFS employs a coarse-grained data striping (4MB stripe size) across the computational storage drives in order to ensure most data blocks entirely reside on one computational storage drive. In the rare case of one compressed block locates across two drives, the system will use storage node CPU to handle the corresponding scan operation.

As discussed in Section 3.1.1, POLARDB storage engine specifies the location of to-be-scanned data in the form of offsets in files. The to-be-scanned data may span over multiple files and hence multiple computational storage drives. Meanwhile, computational storage drives can only locate data in the form of LBAs. Therefore, upon receiving each table scan request from POLARDB storage engine, PolarFS must appropriately convert this request before forwarding it to the computational storage driver. Accordingly, we have enhanced PolarFS from the following aspects: (1) Suppose the to-be-scanned data span over *m* computational drives, the enhanced PolarFS decomposes this request into *m* scan requests, each of which scans the data on one computational storage drive. (2) For each scan request, it converts the data location information into offsets in LBAs. As illustrated in Fig. 3, the enhanced PolarFS subsequently passes the *m* scan requests with converted LBA-based location information to the underlying computational storage driver.

3.1.3 Enhancement to Computational Storage Driver

As discussed above in Section 2.3, our computational storage drive is fully managed by a host-side driver in the kernel space. The driver exposes each computational storage drive as a block device. Upon receiving each table scan request from PolarFS, the driver carries out the following operations. It first analyzes the scan conditions, and if necessary re-arranges the scan conditions in order to better streamline the hardware-based scan processing and hence improve the throughput. For example, suppose the table contains 16 fields (i.e., f_1, f_2, $\cdots$, f_{16}), and the scan condition involves two comparisons, where the first one compares f_{10} and a constant, and the second

one compares f_2 and f_5. Since hardware can pipeline the table record parsing, field selection, and comparison, if we re-arrange the scan condition by interchanging the position of the two comparisons, we can improve the hardware utilization efficiency and hence achieve higher processing throughput. The driver further converts the location information of the to-be-scanned data from the LBA domain into the physical block address (PBA) domain, where each PBA associates with a fixed location in NAND flash memory.

Moreover, the driver internally partitions each scan request into a number of (much) smaller scan sub-tasks, which can serve for two purposes: (1) A large scan task may occupy the flash memory bandwidth for a long time, which can cause other normal I/O request suffer from a longer latency. This problem can be mitigated by partitioning a large scan task into small sub-tasks and cohesively scheduling them with normal I/O requests. (2) By partitioning a large scan task into small sub-tasks, it helps to reduce the hardware resource usage for internal buffering and improve flash memory access parallelism. Moreover, storage device background operations, in particular garbage collection (GC), can severely interfere with table scan and hence cause significant latency penalty. Since all the flash management functions are handled by the host-side driver, we enhanced the driver so that it can cohesively schedule GC and table scan in order to minimize the GC-induced interference. In particular, in the case of heavy and bursty analytical processing workloads, the driver will adaptively reduce or even suspend the GC operation.

3.2 Reduce Hardware Implementation Cost

In order to tackle the challenge of computational storage drive implementation cost, the key is to maximize the FPGA hardware resource utilization efficiency. To achieve this objective, we further developed the following techniques across the software and hardware layers.

3.2.1 Hardware-Friendly Data Block Format

We first modified POLARDB storage engine data block format in order to facilitate the FPGA implementation of table scan. Table scan mainly involves various data comparison operations (e.g., $=$, $\geq$, $\leq$). In spite of the FPGA circuit-level programmability, it is difficult for FPGA to implement comparators that can efficiently support multiple different data types. In this work, we modified POLARDB storage engine so that it stores all the table data in the memory-comparable format, i.e., data can be compared using the function *memcmp*(). As a result, computational storage drives only need to implement a single type of comparator that can carry out the *memcmp*() function, regardless of the specific data types in different fields of a table. By enabling the implementation of type-oblivious comparators in FPGA, this can largely reduce the usage of FPGA resources for implementing table scan.

We further modified the storage engine data block structure in order to improve the hardware utilization efficiency. Fig. 4(a) illustrates the data block format being used in the original storage engine: One data block contains a number of sorted table entries, and ends with meta information (i.e., 1-byte data compression type and 4-byte CRC). Although such a block format can be easily handled by CPUs, it is not friendly to the hardware-based table scan in computational storage drives. We modified the data block format as illustrated in Fig. 4(b), where we add an additional block header including 1-byte block compression type, 4-byte number of key-value pairs, and 4-byte number of restart keys (note that restart key is used to facilitate key search in the presence of prefix compression). This modified block format is much more friendly to hardware-based table scan because: (1) Computational storage drive can decompress each block and check CRC without demanding POLARDB storage engine to pass the size information of each block. (2) By adding the "# of keys" and "# of restarts" fields at the beginning of each block, the hardware can more conveniently handle the restarts within each block and detect the end of each block. This is well suited to the sequential data processing flow of the hardware, and hence simplifies the FPGA-based hardware implementation.
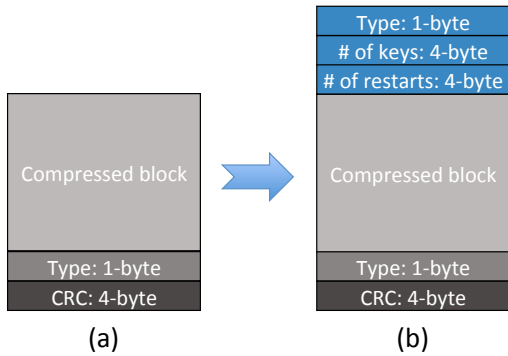
Figure 4: (a) Block structure in conventional practice, and (b) modified block structure to simplify hardware implementation of data scan.

3.2.2 FPGA Implementation

Fig. 5 shows the parallel and pipelined architecture of our FPGA implementation. To reduce the cost, we use a single mid-range FPGA chip for both flash memory control and table scan. The FPGA incorporates a powerful soft-decision LDPC (low-density parity-check) coding engine. This enables the use of low-cost 3D TLC (and QLC in the future) NAND flash memory, which helps to reduce the overall computational storage drive cost. We use a parallel and pipelined hardware architecture to improve the table scan processing throughput. As shown in Fig. 5, it contains two parallel data decompression engines and four data scan engines. Current implementa-

tion supports the Snappy decompression and following scan conditions: $=, \neq, >, \geq, <, \leq$, NULL, and !NULL.
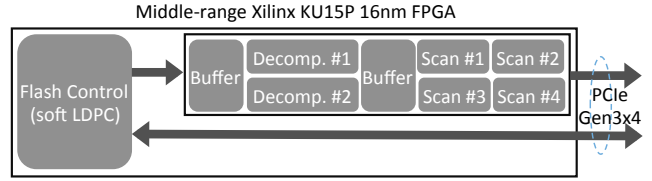
Figure 5: Parallel and pipelined FPGA implementation.

To further improve the hardware resource utilization efficiency, we applied a simple design technique described as follows. As pointed out above, all the fields are stored in the memory-comparable form, hence we only need to implement type-oblivious *memcmp* modules to evaluate each condition. Since the number of scan conditions varies among different table scan tasks, each scan engine employs a recursive architecture in order to maximize the FPGA resource utilization. Each scan engine contains one *memcmp* module and one *RE* (result evaluation) module. Let $P = \sum_{i=1}^{m}(\prod_{j=1}^{n_i} c_{i,j})$ denote the overall scan task, where each $c_{i,j}$ is one individual condition on one field. The symbols $\sum$ and $\prod$ represent the logic OR and AND operation, respectively. Using a single *memcmp* and *RE* module, we recursively evaluate the predicate with one condition $c_{i,j}$ at a time. The *RE* module checks whether the previous *memcmp* output (i.e., all the $c_{i,j}$'s that have been evaluated so far) is sufficient to determine the value of the result P. Once the value of P (i.e., either 1 or 0) can be determined, the scan engine can immediately finish the evaluation on current row, and start to work on another row. This recursive architecture can handle any arbitrary predicate with the optimal FPGA hardware resource utilization.

4   Evaluation

This section presents evaluation results to demonstrate the effectiveness of this deployed solution. The remainder of this section is organized as follows: Section 4.1 summarizes the experimental environment and basic storage performance of the computational storage drives. Section 4.2 evaluates and compares the table scan performance when using CPUs or computational storage devices to realize table scan. Section 4.3 presents the TPC-H evaluation results on a POLARDB instance in Alibaba Cloud, and Section 4.4 provides further concluding remarks.

4.1   Experimental Setup

In order to become practically viable products, besides providing in-storage computing capability, computational storage drives must have top-notch storage I/O performance (at least comparable with leading-edge commodity NVMe SSDs). The

storage performance of our computational storage drives is summarized as follows. Each drive uses 64-layer 3D TLC NAND flash memory chips. With PCIe Gen3×4 interface, each drive can sustain 2.2GB/s and 3.0GB/s sequential write and read throughput. Under 100% address span and fully triggered GC, each drive can achieve 160K and 590K random 4KB write and read IOPS, which are on par with the latest enterprise-grade NVMe SSDs. Each computational storage drive hosts a single mid-range Xilinx UltraScale+ KU15p FPGA chip that handles both flash memory control and computation. To maximize the error correction strength, each drive supports soft-decision LDPC code decoding with beyond-3GB/s decoding throughput. The performance evaluation is carried out on a POLARDB instance (with seven database nodes and three storage nodes) in Alibaba Cloud.

4.2 Table Scan Performance Evaluation

The FPGA inside each computational storage drive incorporates two Snappy decompression engines and four data scan engines. The decompression throughput varies with the data compressibility. Under compression ratio of 60% and 30%, the two decompression engines total can achieve 2.3GB/s and 2.8GB/s decompression throughput, respectively. The data scan engines also have variable throughput that depend on several runtime parameters, e.g., the size of each row in the table, table schema, and scan conditions.

We uses the *LINEITEM* table defined in TPC-H benchmark as a test vehicle to evaluate the effectiveness of moving table scan to computational storage drives. The *LINEITEM* table contains total 16 columns mixed with data types of identifier, integer, decimal, fixed-length and variable-length strings. To cover a wide range of processing complexity, we chose the following six table scan tasks (extracted from different TPC-H queries) to carry out evaluations on one storage node:

TS-1: *Select* L_PARTKEY, L_EXTENDEDPRICE, L_DISCOUNT
from LINEITEM
where L_SHIPDATE ≥ "1994-06-01" and L_SHIPDATE < "1994-07-01"

TS-2: *Select* L_PARTKEY, L_SUPPKEY, L_QUANTITY
from LINEITEM
where L_SHIPDATE ≥ "1993-01-01" and L_SHIPDATE < "1994-01-01"

TS-3: *Select* L_ORDERKEY, L_SUPPKEY, L_EXTENDEDPRICE, L_DISCOUNT, L_SHIPDATE
from LINEITEM
where L_SHIPDATE ≥ "1995-01-01" and L_SHIPDATE ≤ "1996-12-31"

TS-4: *Select* L_ORDERKEY, L_EXTENDEDPRICE, L_DISCOUNT
from LINEITEM
where L_SHIPDATE ≤ "1995-03-12"

TS-5: *Select* L_ORDERKEY
from LINEITEM
where L_COMMITDATE < L_RECEIPTDATE

TS-6: *Select* L_PARTKEY, L_SUPPKEY, L_QUANTITY
from LINEITEM

For the above six scan tasks, the data selectivity in terms of table entries is 1.25%, 15.17%, 30.34%, 54.04%, 63.22%, and 100.00%, respectively. We set the raw data compression ratio as 0.5 when generating the LINEITEM table, and use the Snappy compression library to compress each data block. For each table scan task, we measured the scan latency and PCIe data traffic when turning on and off the table scan pushdown. When we turn off the table scan pushdown, storage node treats each computational storage drive as a normal SSD and relies on CPU to carry out the table scan processing.

Fig. 6 shows the measured scan latency and CPU utilization, where each data point is obtained by averaging the results of 10 independent runs. As discussed above, each computational storage drive contains four hardware data scan engines. Hence, the storage node runs the scan tasks under two hardware configurations: (a) one computational storage drive with 4 CPU threads, and (b) two computational storage drives with 8 CPU threads. The notation *CPU-based Scan* and *CSD-based Scan* correspond to the cases when storage nodes use its CPU and computational storage drives to carry out table scan processing, respectively. As shown in Fig. 6, under each hardware configuration, we studied four cases: (1) *CPU-based scan* without data compression, (2) *CSD-based scan* without data compression, (3) *CPU-based scan* with Snappy compression, and (4) *CSD-based scan* with Snappy compression.

The results clearly show that, compared with *CPU-based scan*, its CSD-based counterpart can *simultaneously* reduce the scan latency and CPU utilization. For example, when we run the scan task TS-1 (with Snappy compression) on two drives with 8 threads, *CSD-based scan* can reduce the latency from 55s to 39s and meanwhile reduce the CPU utilization from 514% to 140%. Compared with other scan tasks, TS-6 can least benefit from CSD-based scan because its very simple scan condition largely under-utilizes the hardware resource in computational storage drives. Even for TS-6 (with Snappy compression), when using two drives with 8 threads, *CSD-based scan* can reduce the latency from 65s to 53s and meanwhile reduce the CPU utilization from 558% to 374%. Fig. 6 also shows that, although the CPU utilization of CPU-based scan remain relatively constant across all the six scan tasks, the CPU utilization of *CSD-based scan* noticeably increases as the data selectivity becomes larger. For example, TS-1 (with the selectivity of 1.25%) and TS-2 (with the selectivity of 15.17%) have less CPU utilization than others. This can be explained as follows: In the case of *CSD-based scan*, the CPU workload is proportional to the data selectivity. The smaller the data selectivity is, the less amount of data are transferred to and processed by the host CPU. In contrast, in the case of *CPU-based scan*, regardless of the data selectivity, host CPU has to fetch and process all the data from drives. The
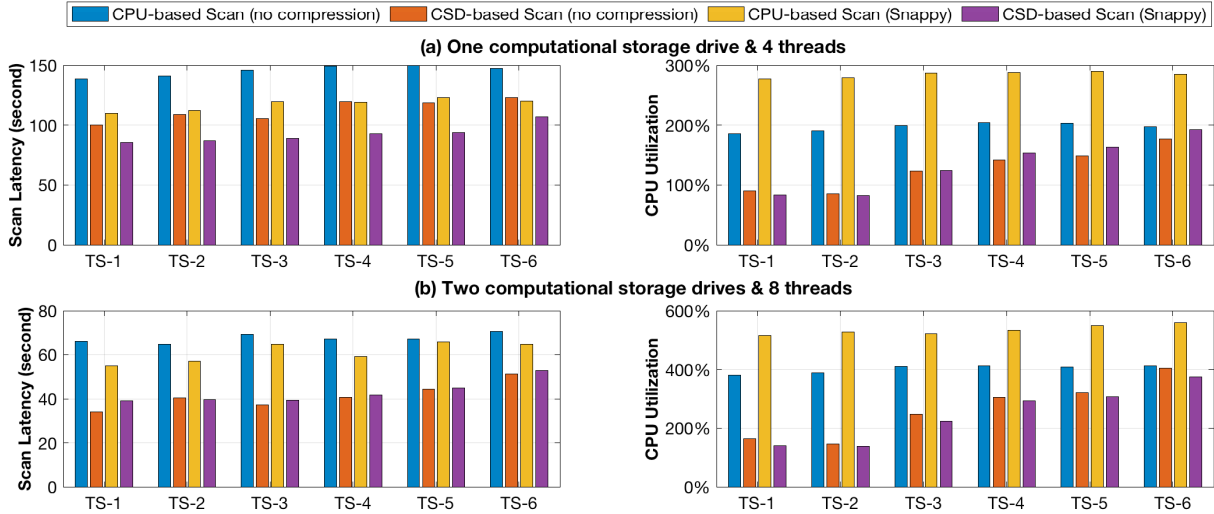
Figure 6: Measured scan latency and CPU utilization when the storage node runs the scan tasks on (a) one computational storage drive with 4 CPU threads, and (b) two computational storage drives with 8 CPU threads.

results also show that the effectiveness of *CSD-based scan* can readily scale with the number of computational storage drives. Finally, the results reveal that light-weight compression (i.e., Snappy in this study) can noticeably improve the performance of *CPU-based scan* at the cost of CPU utilization. In comparison, *CSD-based scan* is relatively insensitive to the use of compression.

To further reveal the benefit of using computational storage table scan pushdown to reduce data movement across the storage and memory hierarchy, Fig. 7(a) shows the measured volume of data being transferred from computational storage drives to host DRAM, and Fig. 7(b) shows the measured total host memory data transfer volume. The results show that
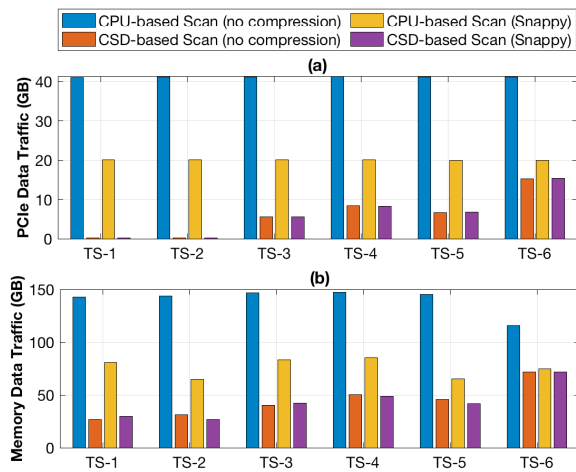
Figure 7: (a) PCIe data traffic and (b) memory data traffic inside the storage node.

CSD-based scan can significantly reduce the data transfer

volume across the storage and memory hierarchy. The benefit improves as the data selectivity becomes smaller. For example, in the case of scan task TS-1 (with the selectivity of 1.25%) , *CSD-based scan* can almost eliminate the PCIe data transfer traffic, and reduce the host memory data traffic by $5\times$ (without compression) and $3\times$ (with compression). The results also show that compression can very effectively reduce data traffic volume across the storage and memory hierarchy.

4.3 System-level Evaluation

We further ran TPC-H analytical workload benchmark on a POLARDB cloud instance with 32 SQL-engine containers distributed on 7 database nodes and 3 back-end storage nodes. Each storage node hosts 12 computational storage drives, and each drive has a capacity of 3.7TB. We considered the following three different scenarios:

1. *No pushdown*: In this baseline scenario, database nodes do not push the table scan down to storage nodes. As a result, storage nodes have to transfer all the data to database nodes for table scan.
2. *CPU-based pushdown*: We enable the table scan pushdown from database nodes to storage nodes, and the CPUs on the storage nodes are responsible for carrying out table scan.
3. *CSD-based pushdown*: We enable the table scan pushdown from database nodes to storage nodes, and the computational storage drives on the storage nodes are responsible for carrying out table scan.

For each one out of the total 22 TPC-H queries, we measured the POLARDB performance by splitting data into partitions and submitting *n* scan requests in parallel to the back-end storage cluster. In this study, we considered three different
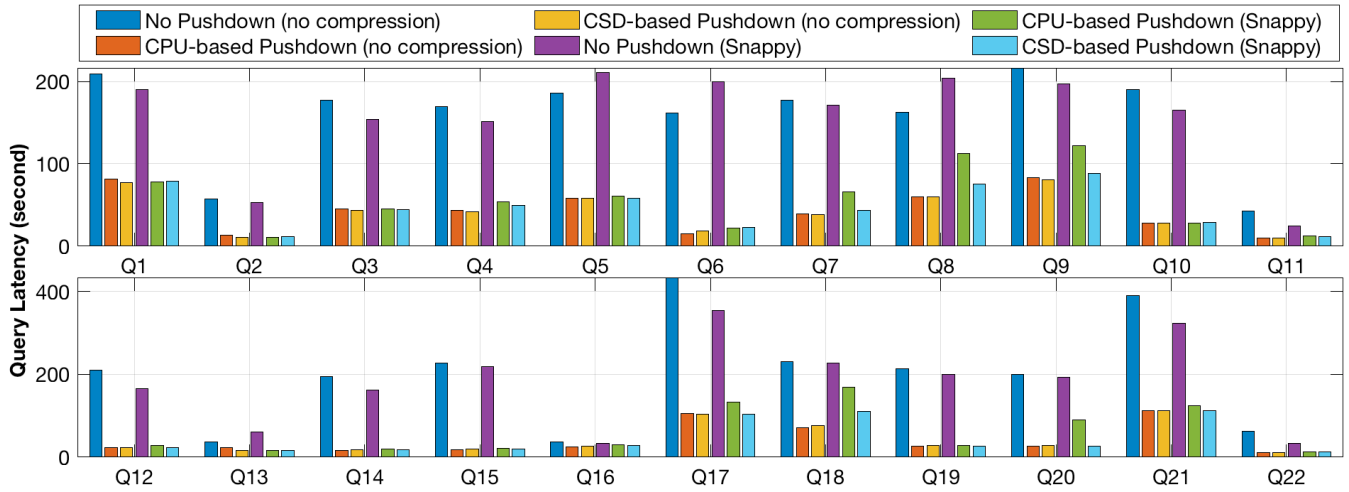
Figure 8: Measured TPC-H query latency under 32 parallel requests.
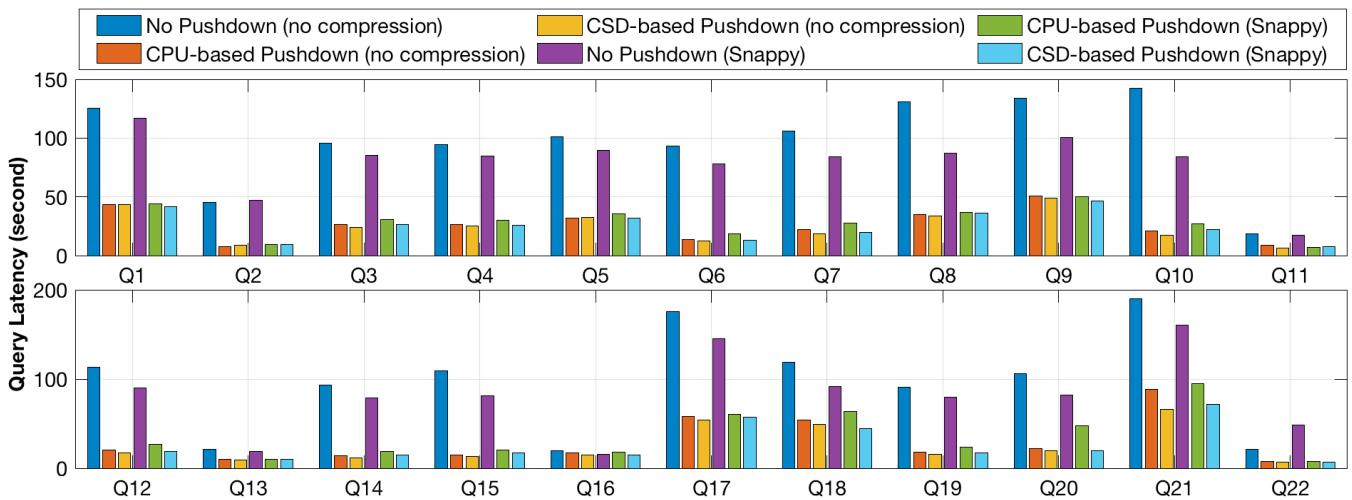
Figure 9: Measured TPC-H query latency under 64 parallel requests.
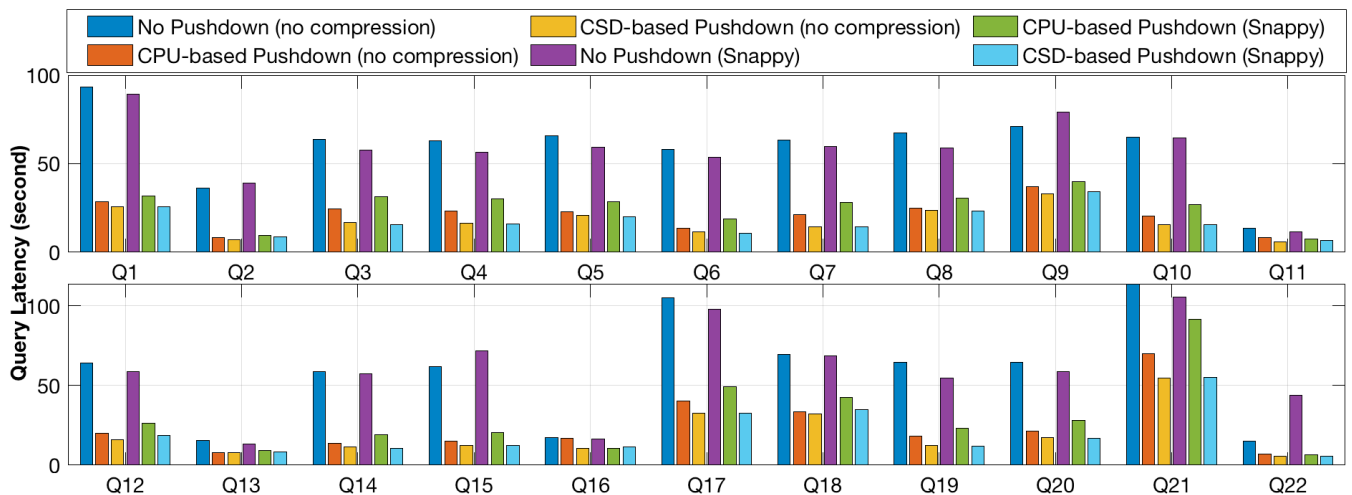
Figure 10: Measured TPC-H query latency under 128 parallel requests.
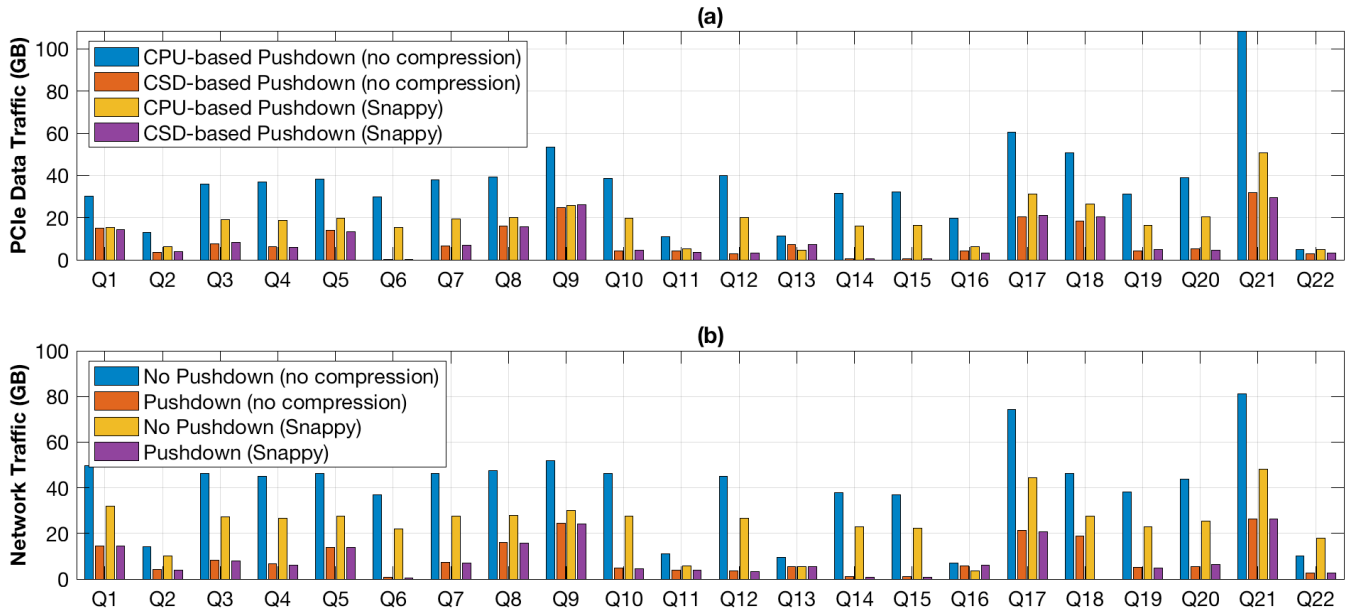
Figure 11: (a) PCIe data traffic inside storage nodes and (b) network data traffic in the POLARDB cluster.

values of *n*: 32, 64, and 128. Fig. 8, Fig. 9, and Fig. 10 show the measured latency of all the 22 TPC-H queries under 32, 64, and 128 parallel requests, respectively. Each evaluation point is obtained by averaging the results of 5 independent runs. The results clearly show the significant benefit of migrating table scan operations from database nodes to storage nodes, which can be intuitively justified given the compute-storage decoupled architecture of POLARDB. The results show that, as the number of requests increases, *CSD-based pushdown* on average can more noticeably outperform *CPU-based pushdown* in terms of scan latency. For example, in the case of 32 parallel requests (with Snappy compression), when switching from *CPU-based pushdown* to *CSD-based pushdown*, only 4 queries experience more than 30% latency reduction. In contrast, in the case of 128 parallel requests (with Snappy compression), when switching from *CPU-based pushdown* to *CSD-based pushdown*, 11 queries experience more than 30% latency reduction, where the maximum latency reduction is 50% for Q7. This is because, as the number of parallel requests increases, storage nodes will have more parallel table scan tasks to better utilize the hardware resource in the computational storage drives. Moreover, the results show that the benefit of *CSD-based pushdown* tends to improve when table data are compressed by Snappy. This can be explained as follows: When table data are compressed, *CPU-based pushdown* will consume more CPU resource in order to handle both data decompression and query processing. Hence a larger number of parallel requests will more likely make *CPU-based pushdown* CPU-bound. In contrast, *CSD-based pushdown* can readily leverage the hardware decompression engines in computational storage drives.

The results also show that *CPU-based pushdown* may even slightly outperform *CSD-based pushdown* in few cases under 32 or 64 requests (e.g., Q10 with 32 requests). This is most likely caused by the sub-optimal behavior of table scan pushdown scheduling, which leads to significant under-utilization of the hardware resource in the computational storage drives. Our future work will focus on improving the quality of table scan pushdown scheduling in order to avoid significant hardware resource under-utilization. Finally, Fig. 11 shows the measured total volume of PCIe data traffic inside storage nodes and total volume of network data traffic between database nodes and storage nodes. When switching from CPU-based pushdown to CSD-based pushdown, 7 TPC-H queries (with Snappy compression) experience more than 50% reduction on the PCIe data traffic volume, where the maximum PCIe data traffic volume reduction is 97% for Q6 followed by 94% for Q14. By moving table scan from database nodes to storage nodes, 12 TPC-H queries (with Snappy compression) experience more than 70% reduction on the total network data traffic volume. The above results clearly demonstrate the significant reduction in data traffic and scan latency of table scan pushdown in cloud-native database.

4.4 Summary

In-storage computing is a very simple concept and has been well discussed in the research community. Nevertheless, its practical implementation and deployment in real systems has remained elusive. Meanwhile, it is not uncommon that significant gain at the component level does not translate to noticeable benefit at the system level. Hence, commercializing the

simple idea of in-storage computing goes far beyond implementing a storage device that can do certain computation, and demands cohesive innovations across software and hardware hierarchy. Targeting at bringing in-storage table scan to cloud-native database systems, we have developed holistic solutions across the storage engine, filesystem, driver, and hardware stack. The component-level evaluation results in Section 4.2 show that our implemented computational storage drive can achieve high-throughput in-storage table scan, leading to significant reduction on host CPU usage and storage-to-memory data movement. The system-level evaluation results in Section 4.3 show that our holistic solution indeed can carry the component-level gain to the system level. The system-level evaluation also confirms the critical importance of realizing table scan pushdown from database nodes to storage nodes.

5 Related Work

Prior work has well studied the promise of accelerating databases using special-purpose hardware (in particular FPGA and GPU) to complement with CPUs. Many prior efforts focused on off-loading the table scan in analytical processing to dedicated accelerators (typically in the form of PCIe cards) built with either FPGA [24, 26, 29] or GPU [7, 25]. Beyond table scan, prior work also investigated the potential of off-loading more complicated query processing kernels [12, 19, 30]. Nevertheless, in spite of extensive prior efforts and impressive performance benefits being demonstrated over the years, IBM/Netezza [24] appears to be the only known commercially successful product on mainstream markets. It off-loads data compression and table scan into dedicated FPGA-based PCIe cards in IBM PureData Systems. Beyond using stand-alone accelerators to complement CPUs, Oracle even integrated special-purpose analytics acceleration units into its own SPARC CPU [6], which however apparently suffers from a very high development cost and has been discontinued by Oracle.

The emerging computational storage enables new opportunities to implement heterogeneous computing platforms for databases. The authors of [13] studied the design of computational storage drives that support key-value store. Prior work [11, 14] focused on leveraging computational storage drives to realize in-storage table scan. Although prior work [11, 14] share the same basic concept as this work, there are several distinct differences: (1) This work presents a holistic system solution in the context of cloud-native relational database, and demonstrates its effectiveness in real production environment. In comparison, prior work [11] ran synthetic queries inside one computational storage drive without integration with databases and system I/O stack. Prior work [14] implemented a prototype based on a modified MySQL running on a single server. It did not consider the integration with a database system with compute-storage decoupled architecture, and did not consider the use of multiple computational storage drives in one server. (2) The basic storage I/O performance metrics (i.e., sequential throughput and IOPS) of the computational storage drives being used in prior work are much worse than that of leading-edge commodity NVMe SSDs. As a result, the systems in prior work tend to be much more I/O-bound and hence more easily benefit from in-storage table scan. The benefits shown in prior work may largely diminish when being compared with systems that deploy leading-edge commodity NVMe SSDs. (3) Both prior work [11, 14] use embedded processors within SSD controllers to carry out the data processing, which however cannot match the multi-GB/s intra-SSD NAND flash memory access bandwidth and hence cannot achieve high-throughput predicate evaluation. (4) Data compression is widely used in databases to reduce the storage bit cost. As a result, computational storage drives must carry out data decompression in order to support predicate evaluation on the data read path. However, prior work [11, 14] did not consider the implementation of data decompression.

6 Conclusions

This paper reports a cohesive cross-software/hardware implementation that enabled Alibaba cloud-native relational database POLARDB to effectively support analytical workloads. The basic design concept is to dispatch the costly table scan operations in analytical processing from CPU into computational storage drives. Being well aligned with current industrial trend towards heterogeneous computing, the key idea is very simple and can trace back to over two decades ago. Nevertheless, it is non-trivial to practically materialize this simple idea with justifiable benefit vs. cost trade-off in the real world. Under the framework of Alibaba POLARDB, this work developed a set of design solutions across the entire software and hardware stacks to practically implement this simple idea in production cloud database environment. Experimental results on a POLARDB cloud instance over 7 database nodes and 3 storage nodes show that our implementation can achieve more than 30% latency reduction for 12 out of the total 22 TPC-H queries. Meanwhile, our implementation can reduce more than 50% storage-to-memory data movement volume for 12 TPC-H queries. It is our hope that this work will inspire much more research and development efforts to investigate how future cloud infrastructure can leverage the emerging computational storage drives.

References

[1] *SNIA Technical Work Group on Computational Storage*. https://www.snia.org/computational.

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. Row-stores: How different are they really? In

Proceedings of the International Conference on Management of Data (SIGMOD), pages 967–980, 2008.

[3] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 81–91, 1998.

[4] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 159–174, 2007.

[5] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, Nov. 2002.

[6] K. Aingaran, S. Jairath, and D. Lutz. Software in silicon in the Oracle SPARC M7 processor. In *IEEE Hot Chips Symposium (HCS)*, pages 1–31, 2016.

[7] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103, 2010.

[8] M. T. Bohr and I. A. Young. CMOS scaling trends and beyond. *IEEE Micro*, 37(6):20–29, November 2017.

[9] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proc. VLDB Endow.*, 11(12):1849–1862, Aug. 2018.

[10] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger. Active disk meets flash: A case for intelligent SSDs. In *Proc. of the International ACM Conference on Supercomputing*, pages 91–102, 2013.

[11] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1221–1230, 2013.

[12] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras. FPGA-based multithreading for in-memory hash joins. In *Proc. of Conference on Innovative Data Systems Research (CIDR)*, 2015.

[13] Z. István, D. Sidler, and G. Alonso. Caribou: Intelligent distributed storage. *Proc. VLDB Endow.*, 10(11):1202–1213, Aug. 2017.

[14] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. G. Lee, and J. Jeong. YourSQL: A high-performance database system leveraging in-storage computing. *Proc. VLDB Endow.*, 9(12):924–935, Aug. 2016.

[15] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. BlueDBM: An appliance for big data analytics. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2015.

[16] Y. Kang, Y.-S. Kee, E. Miller, and C. Park. Enabling cost-effective data processing with smart SSD. In *Proc. of IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, May 2013.

[17] J. J. Levandoski, D. B. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In *Proceedings of Biennial Conference on Innovative Data Systems Research (CIDR)*, 2015.

[18] D. Li, F. Wu, Y. Weng, Q. Yang, and C. Xie. HODS: Hardware object deserialization inside SSD storage. In *Proc. of IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 157–164, 2018.

[19] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Flexible query processor on FPGAs. *Proc. VLDB Endow.*, 6(12):1310–1313, Aug. 2013.

[20] 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.

[21] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, Mar 1997.

[22] E. Riedel, G. A. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, pages 62–73, 1998.

[23] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A user-programmable SSD. In *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 67–80, 2014.

[24] M. Singh and B. Leonhardi. Introduction to the IBM Netezza warehouse appliance. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 385–386, 2011.

[25] E. A. Sitaridi and K. A. Ross. Optimizing select conditions on GPUs. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN)*, pages 4:1–4:8, 2013.

[26] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad. Database analytics acceleration using FPGAs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 411–420, 2012.

[27] D. Tiwari, S. S. Vazhkudai, Y. Kim, X. Ma, S. Boboila, and P. J. Desnoyers. Reducing data movement costs using energy efficient, active computation on ssd. In *Proc. of the USENIX Conference on Power-Aware Computing and Systems (HotPower)*, 2012.

[28] A. 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 *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1041–1052, 2017.

[29] L. Woods, Z. István, and G. Alonso. Ibex: An intelligent storage engine with support for advanced SQL offloading. *Proc. VLDB Endow.*, 7(11):963–974, July 2014.

[30] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *proc. of Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 107–118, 2012.

Carver: Finding Important Parameters for Storage System Tuning

Zhen Cao,[1] Geoff Kuenning,[2] and Erez Zadok[1]

[1]*Stony Brook University*   *and*   [2]*Harvey Mudd College*

Abstract

Storage systems usually have many parameters that affect their behavior. Tuning those parameters can provide significant gains in performance. Alas, both manual and automatic tuning methods struggle due to the large number of parameters and exponential number of possible configurations. Since previous research has shown that some parameters have greater performance impact than others, focusing on a smaller number of more important parameters can speed up auto-tuning systems because they would have a smaller state space to explore. In this paper, we propose Carver, which uses (1) a variance-based metric to quantify storage parameters' importance, (2) Latin Hypercube Sampling to sample huge parameter spaces; and (3) a greedy but efficient parameter-selection algorithm that can identify important parameters. We evaluated Carver on datasets consisting of more than 500,000 experiments on 7 file systems, under 4 representative workloads. Carver successfully identified important parameters for all file systems and showed that importance varies with different workloads. We demonstrated that Carver was able to identify a near-optimal set of important parameters in our datasets. We showed Carver's efficiency by testing it with a small fraction of our dataset; it was able to identify the same set of important parameters with as little as 0.4% of the whole dataset.

1 Introduction

Storage systems are critical components of modern computer systems that have significant impact on application performance and efficiency. Most storage systems have many configurable parameters that control and affect their overall behavior. For example, Linux's Ext4 [22] offers about 60 parameters, representing over 10^{37} potential configuration states. The default settings are often sub-optimal; previous research has shown that tuning storage parameters can improve system performance by a factor of as much as $9\times$ [59].

To cope with the vast number of possible configurations, system administrators usually focus on using their domain expertise to tune a few frequently used and well-studied parameters that are *believed* to significantly impact system performance. However, this manual-tuning approach does not scale well in the face of increasing complexity. Modern storage systems use different file system types [21, 37, 56, 65], new hardware (SSDs [26, 46], SMR [1, 2], NVM [33, 73]), multi-tier and hybrid storage, and multiple virtualization layers (e.g., LVM, RAID). Storage systems range from one or a few identical nodes to hundreds of highly heterogeneous

configurations [23, 57]. Worse, tuning results depend heavily on hardware and the running workloads [10, 11, 70].

Recently, several optimization methods have been used to auto-tune storage systems, achieving good performance improvements within reasonable time frames [11, 40]. These auto-tuning techniques model the storage system as a black box, iteratively trying different configurations, measuring an objective function's value, and—based on previously learned information—selecting new configurations to try. However, many black-box auto-tuning techniques have difficulty scaling to high dimensions and can take a long time to converge on good solutions [61]. Therefore, the problem of dealing with the vast number of storage-parameter configurations remains largely unsolved.

In machine learning and information theory, dimensionality reduction is often applied to explosively sized datasets [5, 48]. We believe it can also be applied to storage-parameter selection. Previous research has reported that certain storage parameters have greater impact on performance than others [11]. By eliminating the less important parameters, and ordering parameters by importance, the parameter search space—and thus the number of configurations that need to be considered by either humans or algorithms—can be reduced significantly [28].

Evaluating a single storage configuration is time consuming, and a thorough analysis requires many configurations to be explored; these evaluations can span days or even months. One purpose of a storage parameter-selection algorithm is to be able to pick important parameters by evaluating only a small number of configurations, yet still select the important parameters with high accuracy.

In this paper, we propose Carver, which efficiently selects a subset of important storage parameters. Carver consists of three components: 1) a variance-based metric to quantify the importance of a storage parameter; 2) a sampling method to intelligently pick a small number of configurations representing the whole parameter space; and 3) a greedy algorithm to select important parameters. Carver outputs a set of selected important parameters; these can be used as pre-selected parameters for auto-tuning algorithms, as well as helping human experts better understand the behaviors of targeted storage systems. As shown in Section 5, the aforementioned three components give Carver the ability to select a near-optimal subset of important parameters by exploring relatively few configurations. With this efficiency, Carver could complete its parameter selection in a relatively short period of time in a real deployment.

Carver was thoroughly evaluated on (publicly available)

experimental data collected from our previous work [11], in which we conducted benchmarks on 7 file systems under 4 workloads over a time span of around four years. In that work, for each file system we picked 8–10 frequently tuned parameters and evaluated *all* possible storage configurations resulting from changing the values of these selected parameters. We collected I/O throughput and latency data throughout the evaluation. The data set consists of more than 500,000 benchmark runs (data points) in total. One advantage of having collected the datasets from the whole configuration space is that they can be used as the ground truth when testing Carver with only a small subset of configurations.

With the collected datasets, we first confirmed that certain parameters have more impact on system throughput or latency than other parameters, using Carver's proposed importance metric. We found that in all datasets there is always a small set of parameters that have significantly more impact on throughput than all the others. For example, under a *Fileserver* workload, the two most important parameters for Ext4 were *Journal Option* and *I/O Scheduler*. We also observed that the set of important parameters varies with different workloads. In the same Ext4 example, the two most important parameters became *Block Size* and *Inode Size* when the workload changed to *Dbserver*. We also demonstrated that our variance-based metric can always find a near-optimal set of important parameters in these datasets.

We then demonstrated Carver's efficiency in identifying important parameters by applying it to different measurements, such as I/O throughput and latency. Carver can easily be extended and applied equally well to other quantifiable objectives such as energy consumption, and even composite cost functions [41]. In our evaluation, Carver uses Latin Hypercube Sampling (LHS) as the sampling method. LHS allows Carver to identify the set of important parameters using a small number of experimental runs that explore only a fraction of all configurations. For instance, among all 1,000 repeated runs, Carver was able to find the two most important parameters for Ext4 using only 0.4% of the evaluation results. We believe Carver's efficiency in finding the most important parameters quickly and accurately is critical and promising, since (1) it can be applied to new storage systems or environments, and (2) the parameters it identifies can then be used by storage administrators or auto-tuning algorithms to further optimize the system.

The three key contributions of this paper are:

1. We provide a thorough quantitative analysis of the effects of storage parameters on system performance, for 7 different file systems across 4 representative workloads.

2. We propose Carver, which uses a variance-based metric of storage-parameter importance and Latin Hypercube Sampling to drive a greedy algorithm that can identify

the most important parameters using only a small number of experimental runs.

3. We thoroughly evaluated Carver's ability to identify important parameters in terms of I/O throughput and latency. We demonstrated that Carver successfully chose a near-optimal set of important parameters for all datasets used.

2   Motivation

In this paper, we define a **storage system** as the entire storage stack from file systems to physical devices, including all intermediate layers. Storage systems have many configurable options that affect their performance [10, 66], energy consumption [59], reliability [63], etc. We define a *parameter* as one configurable option, and a *configuration* as a combination of parameter values. For example, Ext4's *Journal Option* parameter can take three values: *data=writeback*, *data=ordered*, and *data=journal*. Based on this, *[journal="data=writeback", block_size=4K, inode_size=4K]* is one *configuration* with three specific parameter values (*Journal Option*, *Block Size*, and *Inode Size*). The list of all possible (legal) configurations forms a *parameter space*.

Storage systems usually come with many configurable parameters that control and affect their overall behavior. An earlier study [59] showed that tuning even a tiny set of parameters could improve performance and energy efficiency by as much as $9\times$. However, tuning storage systems is not an easy task; we believe its challenges arise from at least the following four aspects:

1. **Large parameter spaces**. Storage systems are complex, incorporating numerous file system types [21, 37, 56, 65], devices [1, 2, 26, 33, 46, 73], and intermediate layers [52, 54]. They often span large networks and distributed environments [6, 23, 30, 57]. Modern storage systems have hundreds or even thousands of tunable parameters—and networks are also parameterized. Worse, evaluating a single configuration can take many minutes or even hours, making experimental tuning unusually time-consuming.

2. **Nontransferable tuning results**. Evaluation results depend on the specific environment, including the hardware, software, and workload [10, 11, 59]. A good configuration for one setup might perform poorly when the environment changes even slightly [60].

3. **Nonlinear parameters**. A system is *nonlinear* when the output is not directly proportional to the input. Many computer systems are nonlinear [16], including storage systems [66]. This makes traditional regression-based analysis more challenging [50, 58].

4. **Discrete and non-numeric (categorical) parameters**. Some storage parameters are continuous, but many are

discrete and take only a limited set of values. Worse, some are *categorical* (e.g., the I/O scheduler name or file system type). Many optimization techniques perform poorly on discrete values, and often cannot address categorical values efficiently or at all [24, 49].

Given these challenges, manually tuning storage systems becomes nearly impossible, and automatic tuning can be computationally infeasible. Recent efforts have used black-box optimization techniques to auto-tune storage configurations [11, 40], addressing several of the above challenges and achieving useful performance improvements. However, we believe that the challenge of tuning storage systems is far from being solved. It has been shown that several of these black-box optimization techniques have scalability problems in high-dimensional spaces [61]. Therefore, directly applying them to tuning systems with hundreds or thousands of parameters would be difficult.

In machine learning and information theory, *dimensionality reduction* is a common technique for coping with large-sized datasets [5, 48]. If it can be applied in storage systems, it will significantly reduce the search space [28], making it easier for humans or algorithms to tune storage systems.

Previous work has reported that not all storage parameters have an equally important performance impact: a few have much greater effect than others [11]. We observed similar trends from our collected datasets. Figure 1 demonstrates the impact of the parameters *Block Size* and *I/O Scheduler* on the throughput of an Ext4 file systems under a typical file server workload. Each boxplot in the figure represents a median and range of throughput that any Ext4 configuration can produce after fixing the value of one parameter (shown on the X axis). We see that setting the *I/O Scheduler* to different values (blue bars) makes little difference, resulting in nearly equal medians and ranges of throughput. However, setting the value of *Block Size* has a greater impact on both the median and the throughput range; specifically, to reach the maximum throughput, *Block Size* must be set to 4K. Although choosing a large *Block Size* is a decision that may be obvious to an expert, we have made similar observations in other storage systems and with different workloads. This naturally led us to investigate how we can quantify the impact or importance of each storage parameter, and how we can select important parameters efficiently.

3 Dimensionality Reduction in a Nutshell

In this section we briefly discuss some commonly applied approaches to *dimensionality reduction*, and argue that some metrics are not suitable for quantifying storage parameters' importance. Note that different disciplines might use somewhat different terminology than storage systems. For example, parameters are analogous to *features* in machine learning, *independent variables* in regression analysis, and *dimensions* in mathematics; optimization objectives can be called *dependent variables* or *target variables*. When discussing
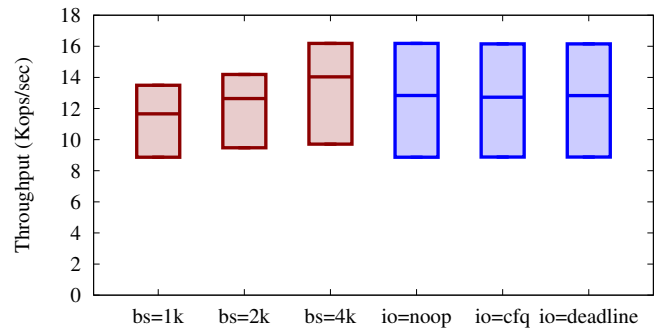
Figure 1: Range of throughput after fixing the value of one parameter. Red bars represent setting the block size to 1K, 2K, or 4K, respectively, while blue bars represent setting the I/O scheduler to noop, cfq, or deadline.

different techniques (Section 3), we use the field-appropriate terms.

Many approaches have been proposed to address the *curse of dimensionality*, which refers to the fact that data become sparse in high-dimensional spaces and thus make algorithms designed for low-dimensional spaces less effective. Dimensionality-reduction approaches can be generally summarized into two categories: *feature extraction* and *feature selection* [25, 39].

Feature extraction refers to projecting high-dimensional data into low-dimensional spaces; the newly constructed features are usually linear or nonlinear combinations of the originals. Common feature-extraction methods include Principal Component Analysis (PCA) [62], Independent Component Analysis [29], and Linear Discriminant Analysis [47]. One major drawback of feature extraction is that the physical meaning of each feature is lost by the projection and the nonlinear combination of many dimensions into fewer ones [39]. Common feature-extraction techniques thus conflict with our goal in this paper, which is to *select a few original storage parameters that can be understood and interpreted*.

Conversely, *feature selection* directly selects a subset of features from the original ones, with the intention of finding only those that are important. Feature-selection methods can be classified as *supervised* or *unsupervised* [39]. Unsupervised feature selection, such as Principle Feature Analysis [43], chooses a subset that contains most of the essential information based on relationships among features. It does not consider the impact of features on optimization objectives during the selection phase. In contrast, supervised feature selection chooses a subset that can discriminate between or approximate the target variables. Examples include Lasso [68] and decision-tree based algorithms [31]. Since we are interested in finding parameters that have significant impact on our optimization objectives, such as I/O throughput, supervised feature selection best fits our needs.

Several intrinsic properties of our project also limit our choice of feature-selection methods. Many storage parameters are discrete or categorical (see Sections 2 and 5.1). The

performance of storage systems is usually presented as I/O throughput or latency, which are continuous. Therefore, an ideal feature-selection method should work with categorical features and continuous targets. Although there are discretization techniques that can break continuous target variables into discrete sections, feature-selection results depend heavily on the quality of discretization [39]. One common approach for dealing with categorical features is to transform each of them into dummy binary parameters that take values of 0 or 1. For instance, *io_scheduler* with three possible values (*noop*, *deadline*, and *cfq*) can be converted into three binary features: *"io_scheduler = noop"*, *"io_scheduler = deadline"*, and *"io_scheduler = cfq"*. All the binary features can take on values 0 or 1. This approach is unsatisfactory because it selects the individual binary features instead of the original categorical ones. Moreover, converting a categorical parameter with N values into N separate binary parameters would *expand* the parameter space exponentially. For this reason, we feel that Lasso [68] is not suitable for our problem, even though it has been successfully applied to selecting important knobs in databases [70]. Although Group Lasso has been proposed to partially address this deficiency [14,34,74], the computational cost of the Lasso-based methods is still high [39].

Another popular category of feature-selection methods has been built upon information theory [8,20,31,39]. These approaches usually define a metric for the *homogeneity* of the target variable within certain subsets. Commonly used metrics include Gini impurity [39] and Entropy [5] for discrete target variables, and Variance [7] for continuous variables. In this paper we propose Carver, which applies a *variance-based* metric for parameter importance, as described in Section 4.1.

4 Design of Carver

In this section we detail the design of Carver. Carver consists of three components: 1) a variance-based metric for measuring storage parameters' importance (Section 4.1), 2) a sampling method to select a small number of configurations from huge parameter spaces—in this paper using Latin Hypercube Sampling (Section 4.2), and 3) a greedy algorithm for finding important parameters (Section 4.3). A good sampling method allows Carver to select a near-optimal subset of important parameters while having to evaluate relatively few configurations. In this section we use throughput as an example of the target (objective) variable, but Carver is also applicable to many other metrics.

4.1 Measuring Parameter Importance

Carver uses a variance-based metric to quantify storage-parameter importance. The variance of a set S of storage configurations is defined as usual:

$$\mathrm{Var}(S) = \frac{1}{|S|} \sum_{i=1}^{|S|} (y_i - \mu)^2, \qquad (1)$$

where y_i is the throughput of the i-*th* configuration; $|S|$ is number of configurations in S; and μ is the average throughput within S. Inspired by CART (Classification and Regression Trees) [7], we use the *reduction in variance* to measure parameter importance. We extend CART's original definition to support categorical parameters taking an arbitrary but finite number of values, as compared with only two in CART.

We define the parameter importance PI of a parameter P that can take a finite number of categorical values, $\{p_1, ..., p_n\}, n > 1$, as:

$$PI(P) = \mathrm{Var}(S) - \sum_{i=1}^{n} \frac{|S_{P=p_i}|}{|S|} \mathrm{Var}(S_{P=p_i}) \qquad (2)$$

Here S is the original set of configurations, and $S_{P=p_i}$ is the subset of configurations with the parameter P taking the value p_i. Intuitively, an important parameter P divides a set S of configurations into multiple subsets, and the weighted sum of variances within each subset should be much smaller than the variance of S. Thus, a high PI indicates a parameter that has a significant effect on performance.

The variance-based metric defined in Carver uses a greedy approach, where the next important parameter will be picked by calculating its importance when fixing the values of previously selected parameters. Therefore, for parameter Q with a total of m possible categorical values $\{q_1, ..., q_m\}, m > 1$, we define the *conditional parameter importance* for Q, given $P = p$ as:

$$CPI(Q|P = p) =$$
$$\mathrm{Var}(S_{P=p}) - \sum_{j=1}^{m} \frac{|S_{Q=q_j,P=p}|}{|S_{P=p}|} \mathrm{Var}(S_{Q=q_j|P=p}) \quad (3)$$

where $S_{Q=q_j,P=p}$ denotes the set of configurations with parameters P and Q taking values p and q_j, respectively. Similar to Equation 2, given $P = p$, the next most important parameter Q divides $S_{P=p}$ into multiple subsets, and if Q is important then the weighted sum of variances within each subset will be much smaller than variance of $S_{P=p}$. To remove the restriction to a given value p, we define $CPI(Q|P)$ as the maximum of $CPI(Q|P = p_i)$ over all possible values $p_i \in \{p_1, ..., p_n\}$ that parameter P can take:

$$CPI(Q|P) = \max_{i=1}^{n} CPI(Q|p = p_i) \qquad (4)$$

Note that in this paper we use only variance-based metrics to measure parameter importance and select the most critical subset. We leave storage-performance prediction, which requires a large amount of training data [71], for future work.

4.2 Sampling

Given the large parameter space and the time needed to evaluate a single storage configuration, we must limit the number of experimental runs required to select important parameters. Therefore, Carver needs an exploratory method that can cover the space uniformly and comprehensively, yet sparsely. In this work, we chose Latin Hypercube Sampling (LHS) [45].

LHS is a stratified sampling method [13]. In two dimensions, a square grid containing samples is a *Latin Square* iff there is only one sample in each row and each column. A *Latin Hypercube* is the generalization of a Latin Square to higher dimensions, where each sample is the only one in each axis-aligned hyperplane containing it [36]. LHS has been shown to be more effective in exploring parameter spaces than random sampling [45] and Monte Carlo sampling [15]. It has been successfully applied in sampling configurations of storage [27] and cloud systems [42].

Previous work has also applied Plackett-Burman (P&B) Design [53] to evaluate the impact of parameters in storage benchmarks [51] and databases [18]. However, P&B design requires each parameter to have only two possible values, and the target variable must be a monotonic function of the input parameters. Neither requirement holds in our problem.

We demonstrated that LHS enables Carver to pick important storage parameters with only a small number of evaluations; see Section 5.4.

4.3 Parameter-Selection Algorithm

Based on our proposed measurements of parameter importance and on Latin Hypercube Sampling (LHS), the pseudo-code for Carver's parameter-selection algorithm is as follows:

Algorithm 1 Parameter Selection

Input: P: set of parameters, S: initial set of configurations; stop(S, selected): user-defined stopping function.

  $selected \leftarrow \{\}$
  $S^* \leftarrow LHS(S)$
  repeat
    $p^* \leftarrow \text{argmax } CPI(p|selected), p \in P$
    $selected.insert(p^*)$
    $P.remove(p^*)$
  until $stop(S, selected)$ **is** true **or** P **is** empty

Output: $selected$

In this algorithm, Carver takes a set of initial parameters and configurations. It first uses LHS to pick a small number of configurations and evaluates them. Carver then greedily selects the current most important parameters based on the evaluation results for the selected configurations. The most-important parameter is selected based on the highest *parameter importance* value. Carver fixes the value of the most important parameter and calculates the *conditional parameter importance* (CPI) values for the remaining parameters;

the parameter with the highest CPI is selected as the second-most important. Carver continues evaluating important parameters by fixing the values of previously selected parameters, until the *stop function* returns true. A naïve stop function could be $sizeof(selected) \geq N$, which would select the N most important parameters. An alternative variance-based stopping function might stop when the variances of subsets of configurations (given the current *selected* parameters) are below a certain threshold ϑ. This stopping condition indicates that by setting the values of the *selected* parameters, the system throughput already falls into a small enough range that there is little potential gain from additional tuning. In our experiments, we applied this idea and used the Relative Standard Deviation (RSD) [13], or Coefficient of Variation, to define our stopping condition. The RSD of a set S of configurations is defined as:

$$\text{RSD}(S) = \frac{1}{\mu}\sqrt{\frac{\text{Var}(S)}{N-1}} \tag{5}$$

where N is the number of configurations and μ is the mean throughput of configurations within S. We chose RSD because it is normalized to the mean throughput and is represented as a percentage; that way the same threshold can be used across different datasets. We used a threshold of 2% in our experiments; as seen in Section 5, parameters selected by this criterion gave us near-optimal and stable throughput.

5 Evaluation

In this section we detail our evaluation of Carver. We first cover the experimental settings we used for collecting datasets in Section 5.1. Section 5.2 provides an overview of storage-parameter importance using our variance-based metric. Section 5.3 demonstrates that the subset of important parameters selected by Carver's importance metric is near-optimal. We show the efficiency of Carver's parameter-selection algorithm in Section 5.4, from multiple perspectives.

5.1 Experiment Settings

To thoroughly study the problem of storage parameter selection and evaluate Carver, we used datasets originally collected for our previous work [11]. The whole dataset consists of more than half a million benchmark results on typical storage systems. We describe the experimental settings and collected datasets in this section.

Hardware. We performed experiments using several Dell PE R710 servers, each with two Intel Xeon quad-core 2.4GHz CPUs, 24GB RAM, and four storage devices: two SAS HDDs, one SATA HDD, and one SSD. Ubuntu 14.04 was installed on all machines with Linux kernel 3.13. We denote this configuration as *S1*. We also collected several datasets on a slightly different configuration, *S2*, where we used the GRUB boot loader to limit the available memory to

4GB. We explain the reasons for this change below. We also upgraded the system to Ubuntu 16.04 with kernel 4.15. Experiments on *S2* were only conducted on the SSD, given the increasing use of SSDs in production systems.

Workload. We benchmarked storage configurations with four common macro-workloads generated by Filebench [3, 67]:

1. **Mailserver** mimics the I/O workload of a multi-threaded email server;

2. **Fileserver** emulates a server hosting users' home directories;

3. **Webserver** emulates a typical static Web server with a high percentage of reads; and

4. **Dbserver** mimics the behavior of an Online Transaction Processing (OLTP) database.

Before each experimental run, we formatted and mounted the storage devices with the selected configuration. In setting *S1* we chose Filebench's default workload profiles, limiting the working-set size so we could evaluate more configurations within a practical time period. We call those profiles *Mailserver-default*, *Fileserver-default*, etc. Our previous study's goal, which applies to this work as well, was to allow us to explore a large set of parameters and values quickly. By evaluating each configuration once, saving the results, and later looking them up in our database, we could test Carver in seconds instead of waiting for several hours to run the benchmarks selected by Algorithm 1. Clearly, a real-world deployment would not have such a database available and a search for the most important parameters would require running actual benchmark tests, each of which would take significant time. However, as shown in Section 5.4, Carver tests few enough configurations that even these experiments can be completed in a short time, ranging from a few hours to a few days. An additional benefit of the full database is that we were able to compare configurations found by Carver with the true best configuration found by our complete datasets.

Because we wanted our database to record results of as many experiments as possible, we decided to trade off a smaller working set size in favor of increasing the number of configurations we could explore in a practical time period. Our experiments demonstrated a wide range of performance numbers and are suitable for the purpose of studying storage-parameter importance. As shown in Table 2, storage parameters do have a wide range of importance under these workloads. We first ran each workload for up to 2 hours to observe its behavior, and then chose a running time long enough for the cumulative throughput to stabilize; we found 100 seconds sufficient for this purpose. In setting *S2*, we increased the working-set size to 10GB and the running time

to 300 seconds, but used relatively fewer total configurations, which we denote *Mailserver-10GB*, *Fileserver-10GB*, etc. The RAM size was set to 4GB in *S2* so that the benchmark working set could *not* fit into memory completely, thus forcing more I/Os.

Parameter space. To evaluate our parameter-selection algorithm, we ideally want our parameter spaces to be large and complex. Considering that evaluating storage systems takes a long time, we decided to experiment with a reasonably sized set of frequently studied and tuned storage parameters. We selected them in close collaboration with several storage experts who have either contributed to storage-stack designs or have spent years tuning storage systems in the field. We chose seven Linux file systems that span a wide range of designs and features: *Ext2* [12], *Ext3* [69], *Ext4* [21], *XFS* [65], *Btrfs* [56], *Nilfs2* [35], and *Reiserfs* [55]. We experimented with various types of parameters, including file-system formatting and mounting options and some Linux kernel parameters. Table 1 lists all our file systems, their (abbreviated) parameters, and the number of possible values that each parameter can take. Note that under *S1* we conducted benchmarks on four storage devices, and we treat the device as one of the parameters. Under *S2* we focused on Ext4 and XFS experiments with an SSD, but evaluated a wider variety of parameters. Cells with "–" mean that the parameters are inapplicable for the given file system. Cells with "dflt" mean we used the default value for that parameter, and so that parameter was not considered during the parameter-selection phase. Note that the total number of configurations for each file system does not necessarily equal the product of the number of parameter values, because some parameter combinations are invalid (e.g., in Ext4 the inode size cannot exceed the block size). The total number of configurations across all datasets is 29,544. We ran all configurations in each parameter space under four workloads. We repeated each experiment at least three times to get a stable and representative measurement of performance. Over a time span of more than two years, we collected data from more than 500,000 experimental runs.

Although we have been collecting benchmarking data over a time span of 4 years, we focused on one dataset at a time, where we benchmarked one file system on the same hardware under the same workload. Each dataset's collection took 1–2 months. Therefore, there may be minor hardware wear-out effects. We repeated each experiment for at least 3 runs, and made sure the variation among the results of these repeated runs were acceptable [10]. We used the average throughput and latency numbers among repeated runs when evaluating Carver.

5.2 Parameter Importance: an Overview

We have collected experimental data from 9 different parameter spaces (Table 1) under 4 representative workload types. Having the complete datasets allowed us to accurately cal-

Set-ting	File System	Blk Size	Inode Size	Block Grp	Jour-nal	Flex Grp	Read-ahead	XFS Sctr Size	Allc Grp Cnt	Log Buf Cnt	Log Buf Size	Allc Size	Node Size	Spec Opt	Atime Opt	I/O Schd	Drty Bg Ratio	Drty Ratio	Dev	Total
S1	Ext2	3	7	6	–	–	–	–	–	–	–	–	–	–	2	3	dflt	dflt	4	2,208
S1	Ext3	3	7	6	3	–	–	–	–	–	–	–	–	–	2	3	dflt	dflt	4	6,624
S1	Ext4	3	7	6	3	dflt	dflt	–	–	–	–	–	–	–	2	3	dflt	dflt	4	6,624
S1	XFS	3	5	–	–	–	–	dflt	9	dflt	dflt	dflt	–	–	2	3	dflt	dflt	4	2,592
S1	Btrfs	–	5	–	–	–	–	–	–	–	–	–	3	4	2	3	dflt	dflt	4	288
S1	Nilfs2	3	–	9	2	–	–	–	–	–	–	–	–	–	2	3	dflt	dflt	4	1,944
S1	Reiserfs	dflt	–	–	3	–	–	–	–	–	–	–	–	2	2	3	dflt	dflt	4	192
S2	Ext4	3	3	dflt	3	3	3	–	–	–	–	–	–	–	dflt	3	2	3	SSD	3,888
S2	XFS	3	2	–	–	–	–	3	4	2	2	2	–	–	dflt	3	2	3	SSD	5,184

Table 1: Details of parameter spaces. Each cell gives the number of settings we tested for the given parameter and file system; empty cells represent parameters that are inapplicable to the given file system and "dflt" represents those that were left at their default setting. We evaluated 29,544 configurations in total under four workloads, and each experiment was repeated $3+$ times.

culate and evaluate the importance of different storage parameters, which serves as the ground truth when evaluating Carver's parameter-selection algorithm, whose goal is to explore only a small fraction of the parameter space yet find the same subset of important parameters as if we had explored it all. In this section, we first provide an overview of the importance of storage parameters.
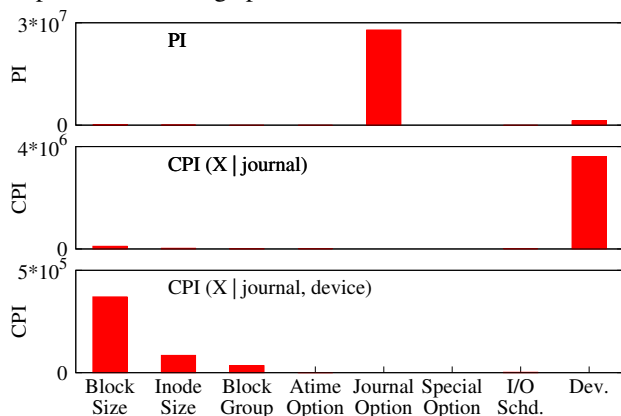
Figure 2: Top 3 most important Ext4 parameters under S1, Fileserver-default. The most important parameter is measured by its PI; the second and third parameters are evaluated by their CPI given higher-ranked parameters. The Y-axis scales in the three sub-figures are different.

Figure 2 shows the three most important parameters for Ext4 under *S1, Fileserver-default*. The parameter with the highest importance was evaluated and selected by its Parameter Importance (PI), as defined in Section 4.1. The second most important parameter was chosen by its Conditional Parameter Importance (CPI) given the most important one, in this case $CPI(X|journal)$. Similarly, the 3rd most important parameter was evaluated by comparing its $CPI(X|journal, device)$. Note that the Y-axis scales in the three sub-figures are different (but higher is always better). The X axis shows the Ext4 parameters that we experimented with. As shown in the top subfigure in Figure 2, *Journal Option* turns out to be the most important parameter for Ext4 under *S1, Fileserver-default*. It has the highest variance reduction, 2.7×10^7. In comparison, the *PI* of *Device* is around 10^6, while all other parameters are under 5×10^4. Similarly, the second and third most important parameters are *Device* and *Block Size*, respectively, both with a much higher *CPI* value than other parameters.

We discovered that parameter importance depends heavily on file system types and on the running workload. Table 2 lists the top 4 important parameters for Ext4, XFS, and Btrfs under various workload types; the column header *#N* identifies the Nth most important parameter. We also applied the stopping criterion described in Section 4.3. Cells marked as "–" here indicate that no parameter gave a large reduction in variance, and thus no parameter was considered important. To avoid cluttering the paper, we only list 3 file systems under 4 workloads here, and we show only the top 4 ranked parameters under each case.

As we can see in Table 2, the important parameters are quite diverse and depend significantly on the file system types and workloads. For Ext4 under *S2* and *Dbserver-10GB*, the top 4 ranked parameters are *Block Size*, *Inode Size*, *I/O Scheduler*, and *Journal Option*. When the workload changes to *Webserver-10GB*, the top 4 parameters become *Inode Size*, *Flex BG*, *Block Size*, and *Journal Option*. For *Fileserver-10GB* under Ext4, we found only three important parameters, indicating that fixing the values of these three parameters already resulted in quite stable throughputs; we discuss this observation in more detail in Section 5.3. We found similar results on XFS: the values and number of important parameters depended heavily on the workloads. Interestingly, for Btrfs under *S1, Webserver-default*, we did not find any important parameters. That is because the *Webserver-default* workload consists primarily of read operations, and the default working-set size used by Filebench is small. All Btrfs configurations actually produce quite similar throughput under *Webserver-default*. For this reason, we also collected datasets from workloads with a much larger working-set size (10GB), denoted as *S2*.

Setting	Workload	File System	Parameter #1	Parameter #2	Parameter #3	Parameter #4
S2	Fileserver-10GB	Ext4	Journal Option	I/O Scheduler	Inode Size	–
S2	Dbserver-10GB	Ext4	Block Size	Inode Size	I/O Scheduler	Journal Option
S2	Mailserver-10GB	Ext4	I/O Scheduler	Inode Size	Journal Option	Block Size
S2	Webserver-10GB	Ext4	Inode Size	Flex Block Group	Block Size	Journal Option
S2	Fileserver–10GB	XFS	I/O Scheduler	Inode Size	Allocation Group Count	–
S2	Dbserver-10GB	XFS	Block Size	Log Buffer Size	Dirty Ratio	Alloc Group Count
S2	Mailserver-10GB	XFS	Inode Size	I/O Scheduler	Log Buffer Size	Allocation Size
S1	Fileserver-default	Btrfs	Special Option	Inode Size	Device	–
S1	Mailserver-default	Btrfs	Inode Size	Device	–	–
S1	Webserver-default	Btrfs	–	–	–	–

Table 2: Top-ranked important parameters for various file systems. The column header #N identifies the Nth most important parameter.

5.3 Evaluating The Greedy Algorithm

In Section 5.2 we used Carver's variance-based metric to pick a set of important parameters for our datasets. However, we must also establish that the selection results are good, i.e., whether there exists another set of parameters, with equal or smaller size, that can lead to an even narrower range of throughput. We demonstrate the effectiveness of Carver's variance-based metric in this section.
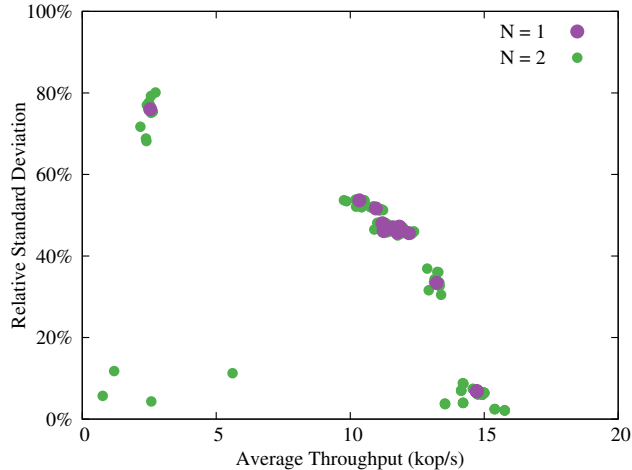
Figure 3: Impact of parameters on performance and stability (Ext4, S1, Fileserver-default). Each dot represents a set of configurations created by fixing N parameters, where different dot sizes and colors are used for different values of N. Performance is measured by the average throughput (X axis) of all possible configurations within each set; stability is measured by the relative standard deviation (Y axis; lower is better) of the throughput within each set.

Figure 3 shows the results for Ext4 under *S1, Fileserver-default*, where each point represents a set of configurations that fixes the values of N parameters. For $N = 1$, we have 28 points, which equals the sum of possible value counts for each parameter, as shown in Table 1. There are 374 points for $N = 2$. We use different point colors and sizes for different numbers of parameters. We only plot up to $N = 2$ here; we extend to $N = 4$ in Figure 4. Larger points are used for smaller N values, since fixing fewer parameter values would result in a larger number of usable configurations. For example, fixing *journal_option = ordered* in our datasets leads to a
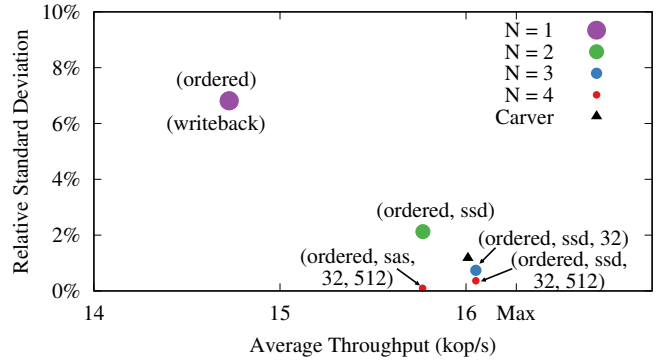
Figure 4: A zoom into the bottom-right part of Figure 3 (the "best" quadrant), with points for $N = 3, 4$ added. Plotted points show either the highest average throughput or the lowest relative standard deviation among all configurations gotten by fixing the values of N parameters. The labels around the dots show the corresponding fixed parameter values. The parameter values are ordered by (Journal Option, Device, Block Group, and Inode Size). The triangle marks the point achieved by fixing the values of parameters selected by Carver.

set of 2,208 configurations; fixing *journal_option = ordered, device=ssd* reduces that number to 552.

In Figure 3, performance is measured by the average throughput within each set of configurations, as presented on the X axis. The Y axis shows the stability of each set, measured by the Relative Standard Deviation (RSD) of throughput within the set. We chose to use the RSD rather than variance because the figure shows sets of varying numbers of configurations; RSD is normalized by the configuration count and the average throughput, and thus is easier to compare. If a set of parameters is important, it should ideally lead to a larger average throughput and lower RSD; therefore the best points should cluster in the bottom-right quadrant of Figure 3. As we can see from that figure, fixing just one parameter value (purple dots) causes the mean throughput to range from 2.5Kops/s to around 15Kops/s, and the RSD ranges from around 7% to 76%. The upper-left purple point (2,500, 76%) represents the configurations achieved by setting *Journal Option* to *journal*. The other two points, representing *Journal Options* of *ordered* and *writeback*, turn out to be the best among all purple points. Both are seen near the bottom right with mean throughput of around 15K and

(a) Ext4, Fileserver-default

(b) Btrfs, Fileserver-default
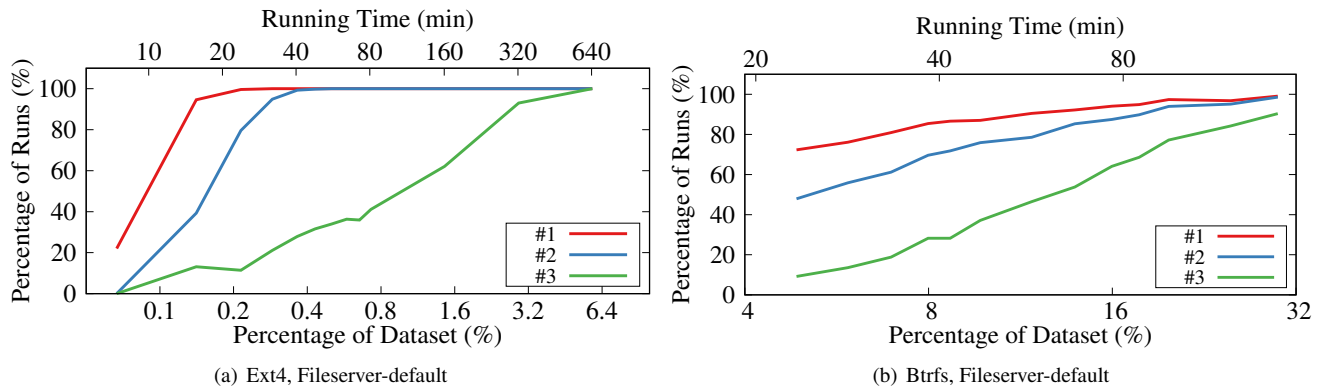
Figure 5: Carver's ability to correctly find the top 3 important parameters within small portions of the dataset. The X1 (bottom) axis (log_2 scale) shows the percentage of the dataset that was used; for each percentage we ran Carver 1,000 times on different, random LHS-compatible subsets of that size. The X2 (top) axis (log_2) shows the running time that would be needed to benchmark the selected configurations. We used the PI calculated from the whole dataset as ground truth. The Y axis shows the percentage of runs that were able to correctly find the important parameters. The solid, dashed, and dotted lines show the results for finding the parameters ranked 1st, 2nd, and 3rd, respectively. Note that although Btrfs required a larger percentage of the dataset, the absolute numbers are similar in both figures, and the running times for Btrfs are shorter (see text).

an RSD value of 7%. Clearly, the *Journal Option* parameter has the highest impact on performance; setting it to an improper value could lead to low throughput and high RSD, while setting it correctly provides significant benefits. The points with $N = 2$ form several clusters. All points with mean throughput less than 9K result from setting *Journal Option* to *journal* (and with another parameter set to various valid values). Conversely, all points with mean throughput larger than 14K result from a *Journal Option* of *ordered* or *writeback*. *Journal Option* is actually the most important parameter selected by Carver (as seen in Table 2).

To probe this question further, we zoomed into the bottom-right part of Figure 3 and added points for $N = 3$ and $N = 4$, as shown in Figure 4. The X and Y axes are similar but with narrower ranges (and the X axis starts at 14K). The label "Max" on the X axis, with a small tick mark, shows the global maximum throughput of all Ext4 configurations within the parameter space. For each N, we plotted only the point(s) with the highest average throughput or lowest RSD. The labels around each point show the associated parameter values, ordered by (*Journal Option*, *Device*, *Block Group*, and *Inode Size*). The black triangle marks the point with highest mean throughput, gotten by fixing the values of the three most important parameters selected by Carver. For $N = 1$, the best two points resulted from setting *Journal Option* to either *ordered* or *writeback*. These two points overlap with each other in this figure, as they share nearly identical mean throughput and RSD values. Only one point is plotted for $N = 2$, since the point (*journal_option=ordered, device=ssd*) shows both the highest throughput and the lowest RSD among all $N = 2$ points; the same is true for $N = 3$. For $N = 4$, the left red point shows the lowest RSD value while the right red point shows the highest average throughput. In Figure 4, the top three parameters selected by Carver are *Journal Option*, *Device*, and *Block Size*. By setting the

values of these three parameters, the best average throughput (denoted as a triangle in Figure 4) is quite close to the global best average throughput achieved by fixing 3 parameter values (blue point). By comparing the two sets of parameters, we can see that Carver successfully identified the top 2 important parameters; the final average throughput and relative standard deviation achieved by the selected top 3 parameters are quite close to the global optimum. We believe the difference in the 3rd selection is due to two reasons:

1. In Carver, the definition of parameter importance focuses on measuring the impact of the parameter on performance, which can be either positive or negative. When discussing "optimality" in Figure 4, we only considered positive impacts.

2. Carver stops after selecting 3 parameters, as the RSD has already dropped below our 2% threshold at that point. If we removed the stopping criterion, the 4th parameter that Carver would select would be *Block Group*, which aligns with the globally optimal set of top 4 parameters, denoted as red dots in Figure 4.

5.4 Carver: Evaluation

All evaluations and analysis in Section 5.2 and 5.3 were conducted on the complete dataset of all possible parameter configurations. However, collecting such datasets for storage parameters is usually impractical, given the challenges discussed in Section 2. One design goal of Carver is to select important parameters while evaluating only a small fraction of configurations. Carver does so by utilizing Latin Hypercube Sampling (LHS), which has been effective in exploring system parameter spaces [27, 42]. We demonstrate the effectiveness of Carver's parameter-selection algorithm from the following two perspectives: selecting important parameters for I/O throughput (see Section 5.4.1) and latency (see

Section 5.4.2).

5.4.1 Selecting Important Parameters for Throughput

A critical question is whether Carver can reliably find the important parameters of a system, and how many experimental runs are necessary to do so. To answer this question, we used our entire dataset of experimental runs on *Ext4, Fileserver-default* and *Btrfs, Fileserver-default* to represent the "ground truth" of which parameters matter. For *Ext4, Fileserver-default*, the top 3 important parameters are *Journal Option*, *Device*, and *Block Size*. For *Btrfs, Fileserver-default*, they are *Special Option*, *Node Size*, and *Device*.

We then tested Carver by repeatedly choosing a random subset of the full dataset, simulating a real-world environment in which an experimenter would use LHS to choose configurations to test, and then using the results of those tests to identify important parameters. In all cases we constrained the random subset to be compatible with Latin Hypercube Sampling (LHS), as our hypothetical investigator would do, and tested whether Carver correctly located the first, second, and third most important parameters. We varied the size of the subsets as a percentage of the entire dataset and ran 1,000 iterations of each trial (with different random subsets).

Figure 5 presents the results of running these experiments. The X1 (bottom) axis shows the percentage of the whole dataset that was used by Carver, and is in $\log_2$ scale. The X2 (top) axis shows the actual running time for benchmarking the selected configurations, and is also in $\log_2$ scale. The Y axis shows the fraction of runs that successfully found the same important parameters as the ground truth. The solid, dashed, and dotted lines show the results of finding the 1st, 2nd, and 3rd most important parameters, respectively.

Figure 5(a) shows that even with only 0.1% of the dataset (7 configurations), Carver has a 60% probability of correctly identifying the most important parameter. When using 0.4% (26), Carver was able to find the 1st and 2nd ranked parameter in 100% and 99.8% of the 1,000 runs, respectively. Setting the values of the most important two parameters would already produce high average throughput (97% of the global optimum) with high stability (2% of RSD), as shown in Figure 4. The chance of correctly selecting the third most important parameter increases with the percentage of the dataset used by Carver. With 1% (67) of the dataset, the probability of correctly finding the 3rd parameter is around 50%, while sampling 5% (331) successfully identifies the 3rd parameter in all 1,000 runs.

For Btrfs, shown in Figure 5(b), Carver needed a larger fraction of the dataset to make correct selections. This is because Btrfs has only 288 configurations, compared with 6,624 for Ext4. Yet by evaluating only 16% (45) of all configurations, Carver found the 1st and 2nd parameters with greater than 80% probability. Carver identified the 3rd parameter in more than 80% of runs with 31% (90) sampled.
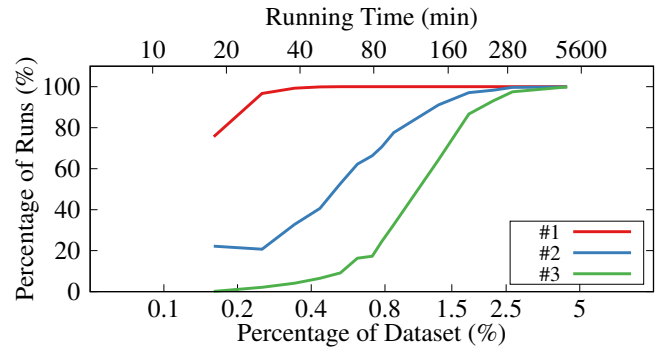
Figure 6: *Carver's ability to correctly find the top 3 important parameters for the latency metric within small portions of the dataset. Experimental settings, graph axes, and legends are the same as in Figure 5.*

5.4.2 Selecting Important Parameters for Latency

To further evaluate Carver's effectiveness in selecting important parameters, we collected datasets with latency metrics. The experimental settings were the same as described in Section 5.1. We ran the *Fileserver* workload on the Ext4 configuration with S2 settings (see Table 1). Instead of using the average I/O throughput reported by Filebench, we now used the average latency. Due to a limitation in Filebench's current implementation, it is difficult to collect and calculate accurate tail latency numbers, such as the 99th percentile, so we leave parameter selection for tail latency as future work.

Figure 6 shows the evaluation results of selecting important parameters using the latency metric. The X axis, Y axis, and legends remain the same as in Figure 5. As shown by the red line, with barely 0.2% of all configurations evaluated, Carver was still able to identify the most important parameters in more than 800 out of 1,000 runs. With 1.5% (58 configurations) evaluated, Carver was able to correctly pick the top 2 parameters in almost all the 1,000 runs. Selecting the third most important parameter required a few more evaluation; using 2.5% of the dataset (97 configurations), Carver successfully identified it in 998 runs.

In sum, Carver is effective in selecting parameters using only a few evaluations. In our experiments, Carver found the top 2 important parameters with higher than 80% probability by evaluating fewer than 50 configurations. Fixing the values of the most important two parameters can already result in high and stable system throughput, as shown in Section 5.3. Carver can find the 3rd parameter with about 50% probability using only about 50 evaluations. Furthermore, the total running time for these evaluations is tractable: the worst case, in Figure 6, is under 4 days. Moreover, auto-tuning a storage system with an optimization algorithms often requires an initialization phase to explore the whole space [11, 42]. Carver can use the data collected during the initialization phase to select parameters; in this case, no *extra* evaluation needs to be conducted. Integrating Carver with auto-tuning

algorithms is part of our future work.

6 Related Work

Parameter selection for computer systems.   There have been several attempts to select important parameters for various types of software systems. Aken *et al.* [70] applied Lasso to choose important knobs for databases. They converted categorical parameters into binary dummy features and included polynomial features to deal with parameter interactions. As discussed in Section 3, Lasso does not scale well when the system has many categorical parameters. Plackett-Burman (P&B) design of experiments [53] has been applied to evaluating the impact of parameters in storage benchmarks [51] and databases [18]. However, P&B assumes that each parameter has only two possible values and that the target variable is a monotonic function of the input parameters; neither assumption holds for storage parameter spaces. Adaptive Sampling [19] and Probabilistic Reasoning [64] have been applied to evaluating the impact of database knobs. They either only work for continuous parameters, or have scalability issues in high-dimensional spaces. In comparison, Carver applies variance-based metrics for storage-parameter importance. To the best of our knowledge, we have conducted the first thorough quantitative study of storage-parameter importance by evaluating Carver on datasets collected from a variety of file systems and workloads. Carver also provides insights into the interactions between parameters.

Auto-tuning storage systems.   Several researchers have built systems to automate storage-system tuning. Strunk *et al.* [63] applied Genetic Algorithms (GAs) to automate storage-system provisioning. Babak *et al.* [4] used GAs to optimize the I/O performance of HDF5 applications. GAs have also been applied to storage-recovery problems [32]. Deep Q-Networks have been successfully applied in optimizing performance for Lustre [40]. More recently, Madireddy *et al.* applied a Gaussian process-based machine learning algorithm to model Lustre's I/O performance and its variability [44]. Our own previous work [11] provided a comparative study of applying multiple optimization algorithms to auto-tune storage systems. However, many auto-tuning algorithms have scalability issues in high-dimensional spaces [61], which is one of the motivations for Carver. Selecting the important subset of parameters could reduce the search space dramatically, which would then benefit either auto-tuning algorithms or manual tuning by experts.

General feature selection.   Many feature-selection techniques have been proposed in various disciplines. Li *et al.* [39] provide a thorough summary and comparison for most state-of-the-art feature-selection algorithms. Based on our arguments in Section 3, we chose to use variance-based metrics for storage-parameter selection.

7 Conclusions

Modern storage systems come with many parameters that affect their behavior. Tuning parameter settings can bring significant performance gains, but both manual tuning by experts and automated tuning have difficulty dealing with large numbers of parameters and configurations. In this paper, we propose Carver, which addresses this problem with the following three contributions:

1. Carver uses a variance-based metric for quantifying storage parameter importance, and proposes a greedy yet efficient parameter-selection algorithm.

2. To the best of our knowledge, we provide the first thorough study of storage-parameter importance. We evaluated Carver across multiple datasets (chosen from more than 500,000 experimental runs) and showed that there is always a small subset of parameters that have the most impact on performance—but that the set of important parameters changes with different workloads, and that there are interactions between parameters.

3. We demonstrated Carver's efficiency by testing it on small fractions of the configuration space. This efficiency gives Carver the potential to be easily applied to new systems and environments and to identify important parameters in a short time, with a small number of configuration evaluations.

In the future, we plan to extend Carver to support other parameter-selection techniques, such as Group Lasso [14,34, 74] and ANOVA [9,13,38,72]. We will evaluate and improve Carver with more optimization objectives (e.g., reliability), and even larger storage-parameter spaces. Currently Carver can only measure storage importance for one objective at a time (e.g., throughput, latency). We plan to investigate how to extend Carver's parameter selection algorithm into the problem of multi-objective optimization [17]. We also plan to integrate Carver with auto-tuning algorithms [11].

References

[1] Abutalib Aghayev, Mansour Shafaei, and Peter Desnoyers. Skylight—a window on shingled disk operation. *Trans. Storage*, 11(4):16:1–16:28, October 2015.

[2] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving ext4 for shingled disks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 105–120,

Santa Clara, CA, February-March 2017. USENIX Association.

[3] George Amvrosiadis and Vasily Tarasov. Filebench github repository, 2016. *https://github.com/filebench/filebench/wiki*.

[4] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. Taming parallel I/O complexity with auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 68:1–68:12, New York, NY, USA, 2013. ACM.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning*, volume 1. Springer New York, 2006.

[6] Dhruba Borthakur et al. HDFS architecture guide. *Hadoop Apache Project*, 53, 2008.

[7] Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. *Classification and regression trees*. CRC press, 1984.

[8] Gavin Brown, Adam Pocock, Ming-Jie Zhao, and Mikel Luján. Conditional likelihood maximisation: A unifying framework for information theoretic feature selection. *Journal of Machine Learning Research*, 13(Jan):27–66, 2012.

[9] Morton B. Brown and Alan B. Forsythe. Robust tests for the equality of variances. *Journal of the American Statistical Association*, 69(346):364–367, 1974.

[10] Zhen Cao, Vasily Tarasov, Hari Raman, Dean Hildebrand, and Erez Zadok. On the performance variation in modern storage stacks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 329–343, Santa Clara, CA, February-March 2017. USENIX Association.

[11] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *Proceedings of the Annual USENIX Technical Conference*, Boston, MA, July 2018. USENIX Association. Data set at *http://download.filesystems.org/auto-tune/ATC-2018-auto-tune-data.sql.gz*.

[12] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings to the First Dutch International Symposium on Linux*, Amsterdam, Netherlands, December 1994.

[13] George Casella and Roger L. Berger. *Statistical Inference*, volume 2. Duxbury Pacific Grove, CA, 2002.

[14] Christophe Chesneau and Mohamed Hebiri. Some theoretical results on the grouped variables Lasso. *Mathematical Methods of Statistics*, 17(4):317–326, 2008.

[15] Liu Chu, Eduardo Souza De Cursi, Abdelkhalak El Hami, and Mohamed Eid. Reliability based optimization with metaheuristic algorithms and Latin hypercube sampling based surrogate models. *Applied and Computational Mathematics*, 4(6):462–468, 2015.

[16] Yvonne Coady, Russ Cox, John DeTreville, Peter Druschel, Joseph Hellerstein, Andrew Hume, Kimberly Keeton, Thu Nguyen, Christopher Small, Lex Stein, and Andrew Warfield. Falling off the cliff: When systems go nonlinear. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems (HOTOS '05)*, 2005.

[17] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[18] Biplob K. Debnath, David J. Lilja, and Mohamed F. Mokbel. SARD: A statistical approach for ranking database tuning parameters. In *IEEE 24th International Conference on Data Engineering Workshop (IDEW)*, pages 11–18, 2008.

[19] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with iTuned. *Proc. VLDB Endow.*, 2(1):1246–1257, August 2009.

[20] Pablo A. Estévez, Michel Tesmer, Claudio A. Perez, and Jacek M. Zurada. Normalized mutual information feature selection. *IEEE Transactions on Neural Networks*, 20(2):189–201, 2009.

[21] Ext4. *http://ext4.wiki.kernel.org/*.

[22] Ext4 documentation. *https://www.kernel.org/doc/Documentation/filesystems/ext4.txt*.

[23] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, NY, October 2003. ACM SIGOPS.

[24] Gradient descent. *https://en.wikipedia.org/wiki/Gradient_descent*.

[25] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3(Mar):1157–1182, 2003.

[26] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 263–276, 2016.

[27] Jun He, Duy Nguyen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Reducing file system tail latencies with Chopper. In *Proceedings of the 13th*

USENIX Conference on File and Storage Technologies, FAST'15, pages 119–133, Berkeley, CA, USA, 2015. USENIX Association.

[28] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U. Michigan Press, 1975.

[29] Aapo Hyvärinen and Erkki Oja. Independent component analysis: Algorithms and applications. *Neural Networks*, 13(4-5):411–430, 2000.

[30] M. Kaminsky, G. Savvides, D. Mazieres, and M. F. Kaashoek. Decentralized user authentication in a global file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, October 2003. ACM SIGOPS.

[31] Jalil Kazemitabar, Arash Amini, Adam Bloniarz, and Ameet S Talwalkar. Variable importance using decision trees. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 426–435. Curran Associates, Inc., 2017.

[32] Kimberly Keeton, Dirk Beyer, Ernesto Brau, Arif Merchant, Cipriano Santos, and Alex Zhang. On the road to recovery: Restoring data after disasters. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 235–248, New York, NY, USA, 2006. ACM.

[33] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 33–45, Berkeley, CA, 2014. USENIX.

[34] Seyoung Kim and Eric P. Xing. Tree-guided group Lasso for multi-task regression with structured sparsity. In *ICML*, pages 543–550, 2010.

[35] 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.

[36] Latin hypercube sampling. *https://en.wikipedia.org/wiki/Latin_hypercube_sampling*.

[37] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[38] Howard Levene. Robust tests for equality of variances. *Contributions to Probability and Statistics. Essays in Honor of Harold Hotelling*, pages 279–292, 1961.

[39] Jundong Li, Kewei Cheng, Suhang Wang, Fred Morstatter, Robert P Trevino, Jiliang Tang, and Huan Liu. Feature selection: A data perspective. *ACM Computing Surveys (CSUR)*, 50(6):94, 2017.

[40] Yan Li, Kenneth Chang, Oceane Bel, Ethan L. Miller, and Darrell D. E. Long. Capes: Unsupervised system performance tuning using neural network-based deep reinforcement learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, 2017.

[41] Z. Li, A. Mukker, and E. Zadok. On the importance of evaluating storage systems' $costs. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, 2014.

[42] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: robustly optimizing tail latencies of cloud systems. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, pages 981–992. USENIX Association, 2018.

[43] Yijuan Lu, Ira Cohen, Xiang Sean Zhou, and Qi Tian. Feature selection using principal feature analysis. In *Proceedings of the 15th ACM international conference on Multimedia*, pages 301–304. ACM, 2007.

[44] Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan M Wild. Machine learning based parallel i/o predictive modeling: A case study on lustre file systems. In *International Conference on High Performance Computing*, pages 184–204. Springer, 2018.

[45] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[46] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2015)*, pages 177–190, Portland, OR, June 2015. ACM.

[47] Sebastian Mika, Gunnar Ratsch, Jason Weston, Bernhard Scholkopf, and Klaus-Robert Mullers. Fisher discriminant analysis with kernels. In *Proceedings of the IEEE Signal Processing Society Workshop on Neural Networks for Signal Processing*, pages 41–48. IEEE, 1999.

[48] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT press, 2012.

[49] John A. Nelder and Roger Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[50] John Neter, Michael H. Kutner, Christopher J. Nachtsheim, and William Wasserman. *Applied Linear Statistical Models*, volume 4. Irwin Chicago, 1996.

[51] Nohhyun Park, Weijun Xiao, Kyubaik Choi, and David J. Lilja. A statistical evaluation of the impact of parameter selection on storage system benchmarks. In *Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, volume 6, 2011.

[52] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD*, pages 109–116, Chicago, IL, June 1988. ACM Press.

[53] Robin L. Plackett and J. Peter Burman. The design of optimum multifactorial experiments. *Biometrika*, pages 305–325, 1946.

[54] LVM2 resource page. *http://sources.redhat.com/lvm2/*.

[55] H. Reiser. ReiserFS v.3 whitepaper. *http://web.archive.org/web/20031015041320/http://namesys.com/*.

[56] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[57] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 231–244, Monterey, CA, January 2002. USENIX Association.

[58] George A.F. Seber and Alan J. Lee. *Linear Regression Analysis*, volume 329. John Wiley & Sons, 2012.

[59] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Evaluating performance and energy in file system server workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

[60] Priya Sehgal, Vasily Tarasov, and Erez Zadok. Optimizing energy and performance for server-class file system workloads. *ACM Transactions on Storage (TOS)*, 6(3), September 2010.

[61] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[62] Jonathon Shlens. A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*, 2014.

[63] John D. Strunk, Eno Thereska, Christos Faloutsos, and Gregory R. Ganger. Using utility to provision storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 313–328, Berkeley, CA, USA, 2008. USENIX Association.

[64] David G. Sullivan, Margo I. Seltzer, and Avi Pfeffer. *Using probabilistic reasoning to automate software tuning*, volume 32. ACM, 2004.

[65] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, San Diego, CA, January 1996.

[66] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking file system benchmarking: It *IS* rocket science. In *Proceedings of HotOS XIII:The 13th USENIX Workshop on Hot Topics in Operating Systems*, Napa, CA, May 2011.

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

[68] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[69] Stephen Tweedie. Ext3, journaling filesystem. In *Ottawa Linux Symposium*, July 2000. *http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html*.

[70] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, 2017.

[71] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage device performance prediction with CART models. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems. (MASCOTS)*, pages 588–595, 2004.

[72] Bernard Lewis Welch. On the comparison of several mean values: An alternative approach. *Biometrika*, 38(3/4):330–336, 1951.

[73] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, Dec 2010.

[74] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the*

Royal Statistical Society: Series B (Statistical Methodology), 68(1):49–67, 2006.

Read as Needed: Building WiSER, a Flash-Optimized Search Engine

Jun He, Kan Wu, Sudarsun Kannan[†], Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

Department of Computer Sciences, University of Wisconsin–Madison
[†]*Department of Computer Science, Rutgers University*

Abstract

We describe WiSER, a clean-slate search engine designed to exploit high-performance SSDs with the philosophy "read as needed". WiSER utilizes many techniques to deliver high throughput and low latency with a relatively small amount of main memory; the techniques include an optimized data layout, a novel two-way cost-aware Bloom filter, adaptive prefetching, and space-time trade-offs. In a system with memory that is significantly smaller than the working set, these techniques increase storage space usage (up to 50%), but reduce read amplification by up to 3x, increase query throughput by up to 2.7x, and reduce latency by 16x when compared to the state-of-the-art Elasticsearch. We believe that the philosophy of "read as needed" can be applied to more applications as the read performance of storage devices keeps improving.

1 Introduction

Modern solid-state storage devices (SSDs) [19, 20] provide much higher throughput and lower latency than traditional persistent storage such as hard disk drives (HDDs). Currently, flash-based SSDs [19, 20] are readily available; in the near future, even higher performance NVRAM-based systems may supplant flash [4], boosting performance even further.

SSDs exhibit vastly different characteristics from HDDs [24, 36]; as we shift from HDDs to SSDs, the software on top of the storage stack must evolve to harvest the high performance of the SSDs. Thus far, optimization for SSDs has taken place within many important pieces of the storage stack. For example, RocksDB [16], Wisckey [38] and other work [23, 34, 42] have made key-value stores more SSD-friendly; FlashGraph [59], Mosaic [43] and other work [48, 49, 60] optimize graphs for SSDs; SFS [45], F2FS [39] and other work [30, 37] have made systems software more SSD-friendly.

In this evolution, an important category of application has been overlooked: full-text search engines. Search engines are widely used in many industrial and scientific settings today, including popular open-source offerings such as Elasticsearch [7] and Solr [3]. As of the time of this writing, Elasticsearch is ranked 7th among all database engines, higher than well-known systems such as Redis, Cassandra, and SQLite [6]. Elasticsearch is used in important applications, such as at Wikipedia and Github to power text (edited contents or source code) search [7, 22]. They are also widely used for data analytics [7]; for example, Uber uses Elasticsearch to generate dynamic prices in real time based on users' locations.

Furthermore, the challenges in search engines are unique, interesting, and different from the ones in key-value stores, graphs, and system software. The key data structure in a search engine is an inverted index, which maps individual terms (i.e., words) to lists of documents that contain the terms. On top of the inverted index, multiple auxiliary data structures (e.g., posting lists) and technologies (e.g., compression) also play important roles to implement an efficient search engine. In addition to compelling data structures, the unique workloads of search engines also provoke new thoughts on SSD-based optimizations. For example, phrase queries (e.g., "United States") stress multiple data structures in the engine and require careful design.

Search engines pose great challenges to storage systems. First, search engines demand low latency as users often interface with them interactively. Second, search engines demand high data throughput because they retrieve information from a large volume of data. Third, search engines demand high scalability because data grows quickly. Due to these challenges, many search engines eschew using secondary storage, putting most/all data directly into memory instead [13, 15].

However, we believe the advent of faster storage suggests a reexamination of modern search engine design. Given that using RAM for large datasets can be cost prohibitive, can one instead rebuild a search engine to better utilize SSDs to achieve the necessary performance goals with main memory that is significantly smaller than the dataset?

We believe the answer is *yes*, if we re-design the system with the principle *read as needed*. Emerging fast storage devices [14, 18, 29, 36, 57] offer high bandwidth; for example, inexpensive (i.e., $0.17/GB) SSDs currently offer 3.5 GB/s read bandwidth [17], and even higher performance can be provided with multiple devices (e.g., RAID). These high-performance storage devices allow applications to read data as needed, which means that main memory can be used as a staging buffer, instead of a cache; thus, large working sets can be kept within secondary storage and less main memory is required. To read as needed, applications must optimize the efficiency of the data stream flowing from the storage device, through the small buffer (memory), to CPU.

In this paper, we present the design, implementation, and evaluation of *WiSER*, a flash-optimized high-I/O search engine that reads data as needed. WiSER reorganizes traditional search data structures to create and improve the read stream, thus exploiting the bandwidth that modern SSDs provide. First, we propose a technique called cross-stage grouping, which produces locality-oriented data layout and significantly reduces read amplification for queries. Second, we propose a novel two-way cost-aware Bloom filter to reduce I/O for phrase queries. Third, we use adaptive prefetch to reduce latency and improve I/O efficiency. Fourth, we enable a space-time trade-off, increasing space utilization on flash for fewer I/Os; for example, we compress documents individually, which consumes more space than compression in groups, but allows us to retrieve documents with less I/O.

We built WiSER[1] from ground up with 11,000 lines of C++ code, for the following reasons. First, an implementation in C++ allows us to interact with the operating system more closely than the state of the art engine, Elasticsearch, which is written in Java; for example, it allows us to readily prefetch data using OS hints. Second, a fresh implementation allows us to reexamine the limitations of current search engines. For example, by comparing with WiSER, we found that the network implementation in Elasticsearch significantly limits its performance and could be improved. Overall, our clean slate implementation produces highly efficient code, which allows us to apply various flash-optimized techniques to achieve high performance. For some (but not all) workloads, WiSER with limited memory performs better than Elasticsearch with memory that holds the entire working set.

We believe that this paper makes the following contributions. First, we propose a design philosophy, read as needed, and follow the philosophy to build a flash-optimized search engine with several novel techniques. For example, we find that plain Bloom filters employed elsewhere [11, 16, 42] surprisingly increase I/O traffic; consequently, we propose two-way cost-aware Bloom filters, which exploit unique properties of search engine data structures. Second, WiSER significantly reduces the need for vast amounts of memory by exploiting high-bandwidth SSDs to achieve high performance at low cost. Third, we have built a highly efficient search engine: thanks to our proposed techniques and efficient implementation, WiSER delivers higher query throughput (up to 2.7x) and lower latencies (up to 16x) than a state-of-the-art search engine (Elasticsearch) in a low-memory system that we use to stress the engine.

The paper is organized as follows. We introduce the background of search engines in Section 2. We propose techniques for building a flash-optimized search engine in Section 3. We evaluate our flash-optimized engine in Section 4, discuss related work in Section 5, and conclude in Section 6.

ID	Text
1	I thought about naming the engine CHEESE, but I could not explain CHEE.
2	Fried cheese curds, cheddar cheese sale.
3	Tofu, also known as bean curd, may not pair well with cheese.

Table 1: **An Example of Documents** *An indexer parses the documents to build an inverted index; a document store will keep the original text.*
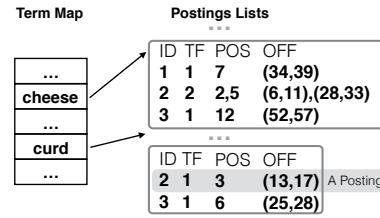
Figure 1: **An Example of Inverted Index** *This figure shows the general contents of inverted index without specific layout information. Term Map allows one to look up the location of the postings list of a term.*

2 Search Engine Background

Search engines play a crucial role in retrieving information from large data collections. Although search engines are designed for text search, they are increasingly being used for data analytics because search engines do not need fixed data schemes and allow flexible data exploration. Popular modern search engines, which share similar features, include Elasticsearch, Solr, and Splunk [6]. Elasticsearch and Solr are open-source projects based on Lucene [1], an information retrieval library. Elasticsearch and Solr wrap Lucene by implementing practical features such as sharding, replication, and network capability. We use Elasticsearch as our baseline as it is the most popular [6] and well-maintained project. Although we only study Elasticsearch, our results also apply to other engines, which share the same underlying library (i.e., Lucene) or key data structures.

2.1 Elasticsearch Data Structures

Search engines allow users to quickly find documents (e.g., text files, web pages) that contain desired information. Documents must be indexed to allow fast searches; the core index structure in popular engines is the *inverted index*, which stores a mapping from a term (e.g., a word) to all the documents that contain the term.

An indexer builds the inverted index. Table 1 shows an example of documents to be indexed. First, the indexer splits a document into tokens by separators such as space and punctuation marks. Second, the indexer transforms the tokens. A common transformation is stemming, which unifies tokens (e.g., curds) to their roots (e.g., curd). The transformed tokens are usually referred to as terms. Finally, the location informa-

---
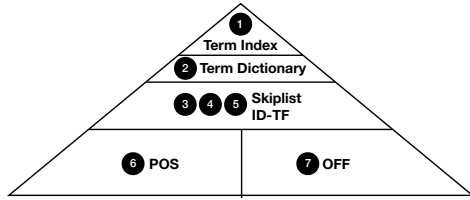
Figure 2: **Inverted Index in Elasticsearch** *Term Index maps a term to an entry in Term Dictionary. A Term Dictionary entry contains metadata about a term (e.g., doc frequency) and multiple pointers pointing to files that contain document IDs and Term Frequencies (ID-TF), positions (POS), and byte offsets (OFF). The number in the figure indicates a typical data access sequence to serve a query. No.3, 4, and 5 indicate the access of skip lists, document ID and Term Frequencies. For Wikipedia, the sizes of each component are Term Index: 4 MB, Term Dictionary: 200 MB, Skiplist.ID.TF: 2.7 GB, POS: 4.8 GB, OFF: 2.8 GB.*

tion of the term is inserted to a list, called a postings list. The resulting inverted index is shown in Figure 1.

A posting contains the location information of a term in a particular document (Figure 1). Such information often includes a document ID, positions, and byte offsets of the term in the document. For example, a position records that term "cheese" is the 2-th and 5-th token in document 2. Positions enable the processing of phrase queries: given a phrase such as "cheese curd", we know that a document contains the phrase if the first term, "cheese", is the x-th token and the second term "curd" is the $x+1$-th one. An offset pair records the start and end byte address of a term in the original document. Offsets enable quick highlighting; the engine presents the most relevant parts (with the queried terms highlighted) of a document to the user. A posting also contains information such as term frequency for ranking the corresponding document.

Query processing includes multiple stages: document matching, ranking, phrase matching, highlighting; different types of queries go through different stages. For queries with a single term, an engine executes the following stages: iterating document IDs in a term's postings list (document matching); calculating the relevance score of each document, which usually uses term frequencies, and finding the top documents (ranking); and highlighting queried terms in the top documents (highlighting). For *AND* queries such as "cheese AND curd", which look for documents containing multiple terms, document matching includes intersecting the document IDs in multiple postings lists. For the example in Figure 1, intersecting $\{1,2,3\}$ and $\{2,3\}$ produces $\{2,3\}$, which are the IDs of documents that contain both "cheese" and "curd". For phrase queries, a search engine needs to use positions to perform phrase matching after document matching.

Figure 2 shows the data structures of Elasticsearch. To serve a query with a single term, Elasticsearch follows these steps. First, Elasticsearch locates a postings list by Term In-
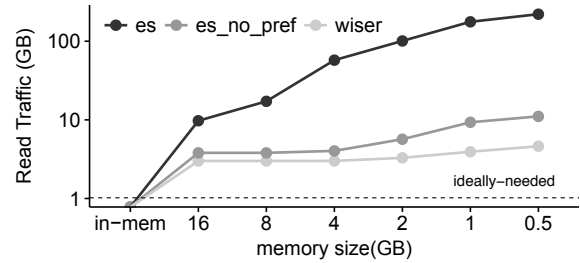
Figure 3: **Read Traffic of Search Engines** *This figure shows read I/O traffic of various search engines as the size of memory decreases.* es*: Elasticsearch,* es_no_pref*: Elasticsearch without prefetch. Note that serving queries leads to only read traffic. The ideally-needed traffic assumes a byte-addressable storage device.*

dex (1) and a Term Dictionary (2). The Term Index and Term Dictionary contain location information of the skip lists, document IDs, positions, and offsets (details below). Second, the engine will load the skip list, which contains more information for navigating document IDs, term frequencies, positions, and offsets. Third, it will iterate through the document IDs and use the corresponding term frequencies to rank documents. Fourth, after finding the documents with top scores, it will read offsets and document text to generate snippets.

2.2 Elasticsearch Performance Problems

Elasticsearch cannot achieve the highest possible performance from the storage system due in part to read amplification. Elasticsearch groups data of different stages into multiple locations and arranges data such that data items in the early stages are smaller. The intention is that data in early stages, which is accessed more frequently than data in later stages, is cached. However, grouping data by stage also could lead to large read amplification.

Figure 3 shows the I/O traffic of a real query workload over Wikipedia; as the amount of memory is decreased, the I/O traffic incurred by Elasticsearch increases significantly. In contrast, the amount of read traffic in WiSER remains relatively low regardless of the amount of memory available.

3 WiSER: A Flash-Optimized Search Engine

Given that SSDs offer high bandwidth, applications that "read data as needed" may be able to run effectively on systems that do not contain enough main memory to cache the entire working set. However, since device bandwidth is not unlimited, applications must carefully control how data is organized and accessed to match the performance characteristics of modern SSD storage [36].

At the highest level, the less I/O an application must perform, the faster that application will complete; since search engine queries involve only read operations, we should *reduce read amplification* as much as possible. Second, retrieving data from SSDs instead of RAM can incur relatively long latency; therefore, applications should *hide I/O latency* as

Term Map

term	zone size	location
term	zone size	location
term	zone size	location

...

| ... | md | skiplist | IDs | TFs | BFs | POSs | OFFs | ... |

Prefetch Zone

Figure 4: **Structure of WiSER's Inverted Index** *Contents of each postings list are placed together. Within each postings list, IDs, TFs and so on are also placed together to maximize compression ratio, which is the same as Elasticsearch.*

much as possible with prefetching or by overlapping computation with I/O. Third, SSDs deliver higher data bandwidth for larger requests; therefore, an application should organize and structure its data to *enable large requests*.

We introduce techniques that allow WiSER to reduce read amplification, hide I/O latency, and issue large requests. First, *cross-stage data grouping* groups data of different stages and stores it compactly during indexing (reducing read implication and enabling large requests). Second, *two-way cost-aware filtering* employs special Bloom filters to prune paths early and reduce I/O for positions in the inverted index; our Bloom filters are novel in that they are tightly integrated with the query processing pipeline and exploit unique properties of search engines (again reducing read amplification). Third, we *adaptively prefetch* data to reduce query latency; unlike the prefetch employed by Elasticsearch (i.e., OS readahead), our prefetch dynamically adjusts the prefetch size to avoid reading unnecessary data (hiding I/O latency and enabling large re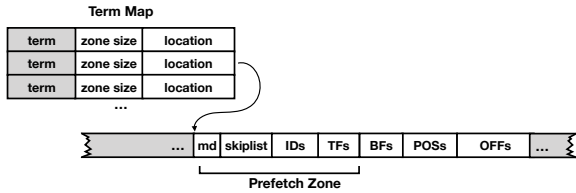quests without increasing read amplification). Fourth, we take advantage of the inexpensive and ample space of SSDs by *trading disk space for I/O speed*; for example, WiSER compresses documents independently and aligns compressed documents to file system blocks to prevent data from crossing multiple blocks (reducing read amplification). We now describe these techniques in more detail.

In discussing our design, we will draw on examples from the English Wikipedia, which is a representative text data set [33, 35, 44, 50, 53, 54, 56]. Its word frequency follows the same distribution (zipf's law) as many other corpuses [41, 50], such as Twitter [47].

3.1 Technique 1: Cross-Stage Data Grouping

One key to building a flash-optimized high-I/O search engine is to reduce read amplification. We propose cross-stage grouping to reduce read amplification for posting lists of small or medium sizes. The processing of such postings lists is critical because most of the postings lists fall into this category. For example, 99% of the postings lists in Wikipedia are small or medium (less than 10,000 documents are in the postings list). Also, search engines often shard large postings lists into such smaller ones to reduce processing latency.

Cross-stage data grouping puts data needed for different stages of a query into continuous and compact blocks on the

storage device, which increases block utilization when transferring data for a query. Figure 4 shows the resulting data structures after group; data needed for a query is located in one place and in the order that it will be accessed. Essentially, the grouped data becomes a stream of data, in which each piece of data is expected to be used at most once. Such expectation matches the query processing in a search engine, in which multiple iterators iterate over lists of data. Such streams can flow through a small buffer efficiently with high block utilization and low read amplification.

Grouped data introduces significantly less read amplification than Elasticsearch for small and medium postings lists. Due to highly efficient compression, the space consumed by each postings list is often small; however, due to Elasticsearch's layout, the data required to serve a query is spread across multiple distant locations (Term Dictionary, ID-TF, POS, and OFF) as shown in Figure 2. Elasticsearch's layout increases the I/O traffic and also the number of I/O operations. On the other hand, as shown in Figure 4, the grouped data can often be stored in the one block (e.g., 99% of the terms in Wikipedia can be stored in a block), incurring only one I/O.

3.2 Technique 2: Two-way Cost-aware Filters

Phrase queries are pervasive and are often used to improve search precision [31]. Unfortunately, phrase queries put great pressure on a search engine's storage system, as they require retrieving a large amount of positions data (as described in Section 2). To build a flash-optimized search engine, we must reduce the I/O traffic of phrase queries.

Bloom filters, which can test if an element is a member of a set, are often used to reduce I/O; however, we have found that plain Bloom filters, which are often directly used in data stores [11, 42, 46], increase I/O traffic for phrase queries because *individual* positions data is relatively small due to compression and, therefore, the corresponding Bloom filter can be larger than the positions data.

As a result, we propose special *two-way cost-aware* Bloom filters by exploiting unique properties of search engines to reduce I/O. The design is based on the observation that the postings list sizes of two (or more) terms in a phrase query are often different. Therefore, we construct Bloom filters during indexing to allow us to pick the smaller Bloom filter to filter out a larger amount of positions data during querying. In addition, we design a special bitmap-based structure to store Bloom filters to further reduce I/O. This section gives more details on our Bloom filters.

3.2.1 Problems of plain Bloom filters

A *plain* Bloom filter set contains terms that are after a particular term; for example, the *set.after* of term "cheese" in document 2 of Table 1 contains "curd" and "sale". To test the existence of a phrase "cheese curd", an engine can simply test if "curd" is in *set.after* of "cheese", without reading any positions. If the test result is negative, we stop and consequently avoid reading the corresponding positions. If the test
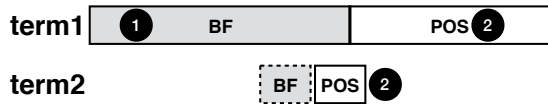
Figure 5: **Phrase Processing With Bloom Filters** *WiSER uses one of the Bloom filters to test the existence of a phrase (1) and then read positions for positive tests to confirm (2).*

Figure 6: **Bloom Filter Store** *The sizes of the arrays may vary because some Bloom filters contain no entries and thus are not stored in the array.*

result is positive, we must confirm the existence of the phrase by checking positions because false positives are possible in Bloom filters; also, we may have to use positions to locate the phrase within a document for highlighting.

However, we must satisfy the following two conditions to reduce I/O. First, the percentage of negative tests must be high because this is the case where we only read the Bloom filters and avoid other I/O. If the test is positive (a phrase may exist), we have to read both Bloom filters and positions, which increases I/O. Empirically, the percentage of positive results is low for real phrase queries to Wikipedia [21]: only 12% of the tests are positive. Intuitively, the probability for two random terms to form a phrase in a document is also low due to a large number of terms in a regular document. The second condition is that the I/O traffic to the Bloom filters must be smaller than the traffic to positions needed to identify a phrase; otherwise, we would just use the positions.

Meeting the second condition is challenging because the sizes of plain Bloom filters are too large in our case, although they are considered space-efficient in other uses [11, 42]. Bloom filters can be larger than their corresponding positions because positions are already space efficient after compression (delta encoding and bit packing [2]). In addition, Bloom filters are configured to be relatively large because their false positive ratios must be low. The first reason to reduce false positive is to increase negative test results, as mentioned above. The second reason is to avoid reading unnecessary file system blocks. Note that a 4-KB file system block contains positions of hundreds of postings. If any of the positions are requested due to false positive tests, the whole 4-KB block must be read; however, none of the data in the block is useful.

3.2.2 Two-way and cost-aware filtering

We now show how we can reduce I/O traffic to both Bloom filters and positions with cost-aware pruning and a bitmap-based store. To realize it, first we estimate I/O cost and use Bloom filters conditionally (i.e., cost-aware): we only use Bloom filters when the I/O cost of reading Bloom filters is much smaller than the cost of reading positions if Bloom filters are not used. For example, in Figure 5, we will not use Bloom filters for query "term1 term2" as the I/O cost of reading the Bloom filters is too large. We estimate the relative I/O costs of Bloom filter and positions among different terms by term frequencies (available before positions are needed), which is proportional to the sizes of Bloom filters and positions.

Second, we build two Bloom filters for each term to al-

low filtering in either direction (i.e., two-way): a set for all following terms and another set for all preceding terms of each term. This design is based on the observation that the positions (and other parts) sizes of the terms in a query are often vastly different. With these two Bloom filters, we can apply filters forward or backward, whichever can reduce I/O. For example, in Figure 5, instead of using Bloom filters of term1 to test if term2 is *after* term1, we can now use Bloom filters of term2 to test if term1 is *before* term2. Because the Bloom filters of term2 are much smaller, we can apply it to significantly reduce I/O.
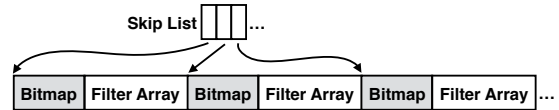
To further reduce the size of Bloom filters, we employ a *bitmap-based data layout* to store Bloom filters. Figure 6 shows the data layout. Bloom filters are separated into groups, each of which contains a fixed number of filters (e.g., 128); the groups are indexed by a skip list to allow skipping reading large chunks of filters. In each group, we use a bitmap to mark the empty filters and only store non-empty filters in the array; thus, empty Bloom filters only take one bit of space (in the bitmap). Reducing the space usage of empty filters can significantly reduce overall space usage of Bloom filters because empty filters are common. For instance, about one-third of the filters for Wikipedia are empty. Empty filters of a term come from surrounding punctuation marks and stop words (e.g., "a", "is", "the"), which are not added to filters.

Empirically, we find that expecting five insertions and a false positive probability of 0.0009 in the Bloom filter [12] (each filter is 9-byte) offers a balanced trade-off between space and test accuracy for English Wikipedia; these parameters should be tuned for other data sets. We use the same parameters for all Bloom filters in WiSER because storing parameters would require extra space and steps before testing each filter; one could improve the space overhead and accuracy by limiting the number of parameter sets for the engine and selecting the optimal ones for specific filters from the available sets.

3.3 Technique 3: Adaptive Prefetching

Although the latency of SSDs is low, it is still much higher than that of memory. The relatively high latency of SSDs adds to query processing time, especially the processing of long postings lists which demands a large amount of I/O. If we load one page at a time as needed, query processing will frequently stop and wait for data, which also increases system overhead. In addition, the I/O efficiency will be low due to small request sizes [36].

To mitigate the impact of high I/O latency and improve the I/O efficiency, we propose adaptive prefetching. Prefetching, a commonly used technique, can reduce I/O stall time, increase the size of I/O requests, and reduce the number of requests, which boosts the efficiency of flash devices and reduces system overhead. However, naive prefetching, such as the Linux readahead [10], used by Elasticsearch, suffers from significant read amplification. Linux unconditionally prefetches data of a fixed size (default: 128 KB), which causes high read amplification due to the diverse data sizes needed.

For the best performance, prefetching should adapt to the queries and the structures of persistent data. Among all data in the inverted index, the most commonly accessed data includes metadata, skip lists, document IDs, and term frequencies, which are often accessed together and sequentially; thus we place them together in an area called the *prefetch zone*. Our adaptive approach prefetches data when doing so can bring significant benefits. We prefetch when all prefetch zones involved in a query are larger than a threshold (e.g., 128 KB); we divide the prefetch zone into small prefetch segments to avoid accessing too much data at a time.

To enable adaptive prefetch, WiSER employs prefetch-friendly data structures, as shown in Figure 4. A search engine should know the size of the prefetch zone before reading the posting list (so the prefetch size can be adapted). To enable such prefetching, we hide the size of the prefetch zone in the highest 16 bits of the offset in WiSER's Term Map (the 48 bits left is more than enough to address large files). In addition, the structure in the prefetch zone is also prefetch-friendly. Data in the prefetch zone is placed in the order it is used, which avoid jumping ahead and waiting for data that has not been prefetched. Finally, compressed data is naturally prefetch-friendly. Even if there are data "holes" in the prefetch zone that are unnecessary for some queries, prefetching data with such holes is still beneficial because these holes are usually small due to compression and the improved I/O efficiency can well offset the cost of such small read amplification.

WiSER prefetches by dynamically calling `madvise()` with the `MADV_SEQUENTIAL` hint to readahead in the prefetch zone. We could further improve prefetching with more precise memory management; for example, we could isolate the buffers used for different queries and avoid interference between queries. In addition, Linux prefetches in fixed sizes; we could allow variable sizes to avoid wasting I/O.

3.4 Technique 4: Trade Disk Space for I/O

With a small increase in disk space, WiSER is able to perform less I/O to its document store. We compress each document individually in WiSER, which often increases space usage but avoids reading and decompressing unnecessary documents. Compression algorithms, such as LZ4, achieve better compression when more data is compressed together. As a result, when compressing documents, engines like Elasticsearch put documents into a buffer (default size: 16 KB) and compresses

all data in the buffer together. Unfortunately, decompressing a document requires reading and decompressing all documents compressed before the document, leading to more I/O and computation. In WiSER, we trade space for less I/O by using more space but reducing the I/O while processing queries.

In addition, WiSER aligns compressed documents to the boundaries of file system blocks if the unaligned data would incur more I/O. A document may suffer from the block-crossing problem, where a document is unnecessarily placed across two (or more) file system blocks and requires reading two blocks during decompression. For example, a 3-KB data chunk has a 75% chance of spanning across two 4-KB file system blocks. To avoid this problem, WiSER aligns compressed document if doing so could reduce the block span.

3.5 Impact on Indexing

Our techniques focus on optimizing query processing instead of index creation since query processing is performed far more frequently. Overall, we believe the overhead introduced to indexing is more than justified by the significant performance improvements on query processing. Cross-stage data grouping does not add overhead to indexing since the same data is simply placed in different locations. Adaptive prefetching employs existing information and does not add any overhead during indexing. Trading space for less I/O adds moderate I/O overhead for the indexing phase (25% for Wikipedia) because the document store takes more space.

Building two-way cost-aware Bloom filters requires additional computation: the indexer in WiSER builds two Bloom filters, *set.before* and *set.after*, for each term in each document. Although a fixed number of hashing calls are required to add an entry to a filter and some filters are empty, the accumulative cost can be high. Currently, we have not optimized the building of Bloom filters. One way to speed up the building is to parallelize it, which also speeds up writing the filters to SSDs. Another way is to cache the hash values of popular terms to avoid hashing the same term frequently; popular terms appear hundreds of thousands times but would only need to be hashed once.

3.6 Implementation

We have implemented WiSER with 11,000 lines of C++ code, which allows us to interact with the OS more directly than higher-level languages. Data files are mapped by `mmap()` to avoid complex buffer management. We rigorously conducted hundreds of unit tests to ensure the correctness of our code.

The major implementation differences between WiSER and Elasticsearch are the programming languages and network libraries. From our experimentation, we found that C++ does not bring significant advantage to WiSER over Elasticsearch. In fact, to make the starting performance of WiSER similar to that of Elasticsearch we had to implement a number of optimizations: we switched from class virtualization to templates; we manually inlined frequently-called functions; we used case-specific functions to allow special optimizations

for the case; and, we avoided frequent memory allocations (e.g., by reusing preallocated std::vector).

4   Evaluation

In this section, we evaluate WiSER with WSBench, a benchmark suite we built, which includes synthetic and realistic search workloads. The impact of a particular technique can be demonstrated by comparisons between two versions of WiSER (i.e., with and without the technique). For example, we demonstrate the effect of two-way cost-aware Bloom filters by comparing WiSER with and without them.

At the beginning of this section, we analyze in detail how each of the proposed techniques in WiSER is able to improve performance by significantly reducing read amplification. We show that: cross-stage data grouping reduces I/O traffic by 2.9 times; two-way cost-aware Bloom filters reduce I/O traffic by 3.2 times; adaptive prefetching prefetches only necessary data; and, trading disk space for less I/O reduces I/O traffic by 1.7 times.

Later in this section, we show that our techniques improve end-to-end performance. For example, we compare WiSER (with Bloom filters) and WiSER (without Bloom filters) to show that our Bloom filters increase query throughput up to 2.6x. We also show that WiSER delivers higher end-to-end performance than Elasticsearch, which indicates that WiSER is well implemented and its techniques can be applied to modern search engines.

We strive to conduct a fair comparison between Elasticsearch and WiSER. We pre-process the dataset using Elasticsearch and input the same pre-processed data to both WiSER and Elasticsearch. The pre-processor produces tokens, positions, and offsets. We implement the exact same relevance calculation (BM25 [7]) in WiSER as is used in Elasticsearch. The pre-processing and the implementation ensure that both WiSER and Elasticsearch will produce query results with the same quality. Despite our efforts, WiSER and Elasticsearch still have many differences (e.g., network implementation, where Elasticsearch performs poorly, and program languages). However, by comparing time-independent metrics such as read traffic size, we can see how WiSER reduces amplification, which in turn improves end-to-end performance.

We conduct experiments on machines with 16 CPU cores, 64-GB RAM and a 256-GB NVMe SSD (peak read bandwidth is 2.0 GB/s; peak IOPS is 200,000) [5]. We use Ubuntu with Linux 4.4.0. We optimize the configuration of Elasticsearch by following the best practices and tune parameters such as the number of threads, heap size and stack size.

To evaluate how well each search engine can scale up to large data sets that do not fit in main memory, our experiments focus on environments with a small ratio of main memory to working set size. The total size of English Wikipedia dataset is 18 GB, and from our experiments, we infer that the working sets are generally a few GBs. Therefore, we configure the search engine processes to use only 512 MB of memory (using

a Linux container); this limits the engine's page cache to a small size (i.e., tens of MBs). Such a configuration allows us to demonstrate that our proposed techniques are effective at reducing read amplification, hiding I/O latency and increasing I/O efficiency, which are essential challenges at larger scale.

4.1   WSBench

We had to create our own benchmark to evaluate WiSER and Elasticsearch because existing benchmarks are not sufficient. To evaluate its engine, the Elasticsearch team uses Wikipedia [33,44,54] and scientific papers from PubMed Central (PMC); unfortunately, the Wikipedia benchmarks do not include real queries and the PMC dataset is very small (only 574,199 documents and 5.5 GB when compressed) with only a few hand-picked queries [8, 9].

We create WSBench, a benchmark based on the Wikipedia corpus, to evaluate WiSER and Elasticsearch. The corpus is from English Wikipedia in May 2018, which includes 6 million documents and 6 million unique terms (excluding stop words). WSBench contains 24 synthetic workloads varying the number of terms, the type of queries, and the popularity level of the queried terms (also known as document frequency: the number of documents in which a term appears). A high popularity level indicates a long postings list and large data size per query. These variables allow us to cover a wide range of query types and stress the system. WSBench also includes a realistic query workload extracted from production Wikipedia servers [21], and three workloads with different query types derived from the original realistic workload.

4.2   Impact of Proposed Techniques

We evaluate the proposed techniques in WiSER for three types of synthetic workloads: single-term queries, two-term queries, and phrase queries. Such evaluations allow us to investigate how the proposed techniques impact various aspects of the system as different techniques have different impacts on workloads. We investigate low-level metrics such as traffic size to precisely show why the proposed techniques improve end-to-end performance.

4.2.1   Cross-stage Data Grouping

Cross-stage grouping can reduce the read amplification for all types of queries. Here we show its impact on single-term and two-term queries where grouping plays the most important role; phrase queries are dominated by positions data where two-way cost-aware Bloom filters play a more important role (as we will soon show).

Figure 7 shows the decomposed read traffic for single-term queries. The figure shows that WiSER can significantly reduce read amplification (indicated by lower `waste` than Elasticsearch); the reduction is up to 2.9x. The reduction is more when the popularity level is lower because the block utilization is lower. To process queries with low-popularity terms, a search engine only needs a small amount of data; for example, an engine only needs approximately 30 bytes of data to
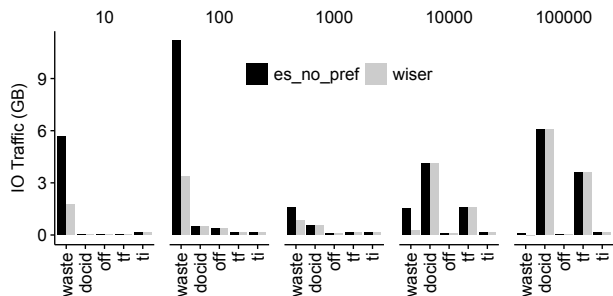
Figure 7: **Decomposed Traffic of Single-Term Queries** `waste` *represents the data that is unnecessarily read;* `docid`, `off`, `skiplist`, `tf`, *and* `ti` *represents the ideally needed data of document ID, offset, skip list, term frequency, term index/dictionary. Positions is not needed in match queries and thus not shown. This figure only shows the traffic from inverted index, which relates to cross-stage data grouping; we investigate the rest of the traffic (document store) later.*
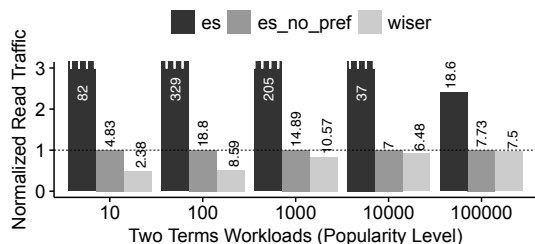
Figure 8: **I/O Traffic of Two-term Match Queries** *The size (GB) is normalized to the traffic size of Elasticsearch without prefetching.*

process the term "tugman" (popularity=8). To retrieve such small data, read amplification is inevitable as the minimal I/O request size is 4 KB. However, we can (and should) minimize the read amplification. Elasticsearch, which groups data by stages, often needs three separate I/O requests for such queries: one to term index, one to document IDs/term frequency, and one to offsets. In contrast, WiSER only needs one I/O request because the data is grouped to one block.

For high popularity levels (popularity=100,000), the traffic reduction is inconspicuous because queries with popular terms require a large amount of data for each stage (KBs or even MBs). In that case, the waste from grouping data by stages in Elasticsearch is negligible.

Figure 8 shows the aggregated I/O traffic for two-term queries, which read two postings lists. Similar to Figure 7, we can see that WiSER (`wiser`) incurs significantly less traffic than Elasticsearch. In this figure, we show two configurations of Elasticsearch: one with prefetch (`es`) and one without prefetch (`es_no_pref`). Prefetch is a common technique to boost performance in systems with ample memory; however, as shown in Figure 8, naive prefetch in Elasticsearch (`es`) can increase read amplification significantly. Such a dilemma motivates our adaptive prefetch.
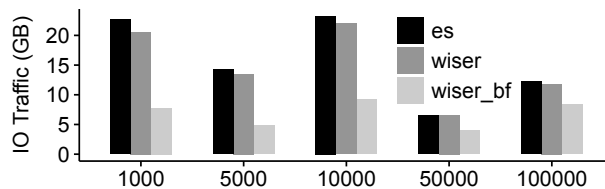
Figure 9: **I/O Traffic of Phrase Queries** *Results of Elasticsearch with prefetch is not shown as it is always much worse than Elasticsearch without prefetch.*
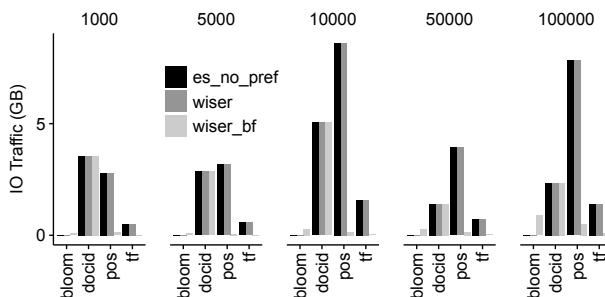
Figure 10: **Decomposed Traffic Analysis of Phrase Queries** *The bars show the ideal traffic sizes for each engine, assuming the storage device is byte-addressable. The sizes were obtained by adding counters to engine code.*

4.2.2 Two-Way Cost-Aware Bloom Filters

Two-way cost-aware Bloom filters only affect phrase queries as filters are only used to avoid positions data, which is used for phrase queries. In Figure 9, we show that WiSER without our Bloom filters demands a similar amount of data as Elasticsearch; WiSER with our Bloom filters incurs much less I/O traffic than WiSER without them and Elasticsearch.

Figure 10 shows the read amplification by the decomposed traffic in Elasticsearch, WiSER without Bloom filters, and WiSER with Bloom filters. The bars labeled with data type names show the data needed ideally, assuming the storage device is byte-addressable. First, we can see that applying filters significantly reduce the data needed ideally which is shown by the reduced aggregated size of all the bars. As shown in the figure, both Elasticsearch (`es`) and WiSER without filters (`wiser`) demand a large amount of position data; in contrast, WiSER with our two-way cost-aware filters (`wiser_bf`) significantly reduces positions needed in all workloads. Surprisingly, we find that our filters also significantly reduce the traffic from term frequencies (`tf`), which is used to iterate positions (an engine needs to know the number of positions in each document in order to iterate to the positions of the destination document). The traffic to term frequencies is reduced because the engine no longer need to iterate many positions.

Note that the introduction of our Bloom filters only adds a small amount of traffic to the Bloom filters (Figure 10), thanks to our two-way cost-aware design and the bitmap-based data layout of Bloom filters. The two-way cost-aware design allows us to prune by the smaller Bloom filter between the two Bloom filters of the two terms in the query. The bitmap-based

Figure 11: **Bloom Filter Footprint** *Our bitmap-based layout reduces footprint by 29%.*

Figure 12: **Document Store Traffic** doc *indicates ideal traffic size. The relative quantity between Elasticsearch and WiSER is the same across different workloads; therefore, we show the result of one workload here for brevity (single-term queries with the popularity level = 10).*

layout, which uses only one bit to store an empty Bloom filter, significantly compresses Bloom filters, reducing traffic. We observe that 32% of the Bloom filters for Wikipedia are empty, which motivates the bitmap-based layout. Figure 11 shows that using bitmap-based layout reduces the Bloom filter footprint by 29%.

4.2.3 Adaptive Prefetching

Adaptive prefetching aims to avoid frequent wait for I/O and reduce read amplification by prefetching only the data needed for the current queries. As shown in Figure 7 and Figure 8, WiSER incurs less traffic than Elasticsearch with and without prefetching. As expected, by taking advantage of the information embedded in the in-memory data structure (Section 3.3), WiSER only prefetches the necessary data. Later in this section, we show that adaptive prefetching is able to avoid waiting for I/O and improve end-to-end performance.

4.2.4 Trade Disk Space for Less I/O

The process of highlighting, which is the last step of all common queries, reads documents from the document store and produces snippets. Figure 12 show that WiSER's highlighting incurs significantly less I/O traffic (42%) to the document store than Elasticsearch because in WiSER documents are decompressed individually and are aligned, whereas Elasticsearch may have to decompress irrelevant documents and read more I/O blocks due to misalignment. The size of WiSER's document store (9.5 GB) is 25% larger than that of Elasticsearch (7.6 GB); however, we argue that this space amplification is well justified by the 42% I/O traffic reduction. WiSER still wastes some traffic because the compressed documents in Wikipedia are small (average: 1.44 KB) but WiSER must read at least 4 KB (the file system block size).

4.3 End-to-end Performance

We examine various types of workloads in this section, including match queries (single-term and multi-term), phrase queries, and real workloads. For match queries, WiSER achieves 2.5 times higher throughput than Elasticsearch. For
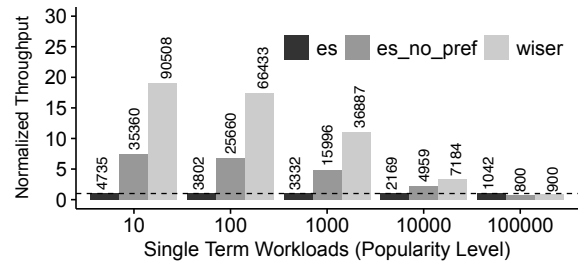
Figure 13: **Single Term Matching Throughput** *The throughput (QPS) is normalized to the performance of Elasticsearch with prefetch (the default 128 KB).*
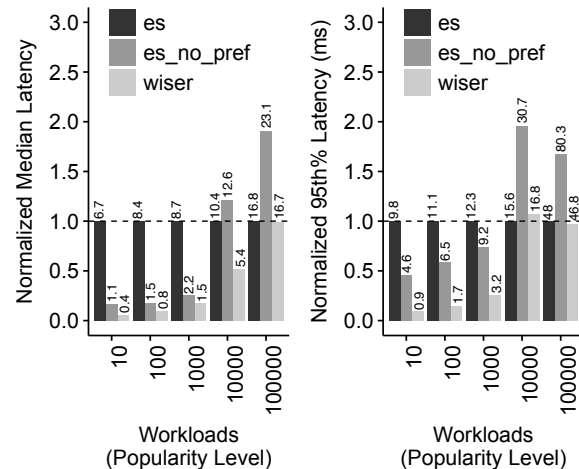
Figure 14: **Single Term Matching Latency** *The latency (ms) is normalized to the median latency of Elasticsearch with default prefetching.*

phrase queries, WiSER achieves 2.7 times higher throughput than Elasticsearch. WiSER achieves consistently higher performance than Elasticsearch for real-world queries. The end-to-end evaluation shows that WiSER is overall more efficient than Elasticsearch, thanks to our proposed techniques and efficient implementation.

4.3.1 Match Queries

We now describe the results for the single-term and multi-term queries. Because queries that match more than two terms share similar characteristics with two-terms queries, we only present the results of two-term queries here.

Figure 13 presents the single-term match QPS (Queries Per Second) of WiSER. The default Elasticsearch is much worse than other engines when the popular levels are low because Elasticsearch incurs significant read amplification: Elasticsearch reads 128 KB of data when only a much smaller amount is needed (e.g., dozens of bytes). Elasticsearch without prefetch (es_no_pref) performs much better than es_default, thanks to much less read amplification.

WiSER achieves higher throughput than Elasticsearch without prefetch (es_no_pref) for low/medium popularity levels, which accounts for a large portion of the postings lists; the speedup is up to 2.5 times. When popularity level is 100,000,
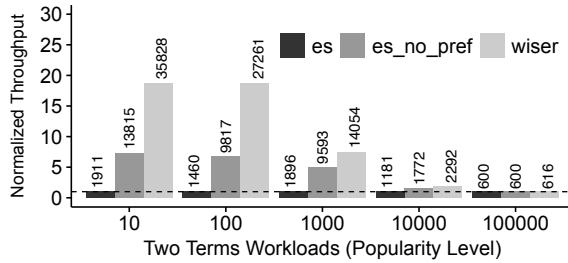
Figure 15: **Two Terms Intersecting Performance** *Throughput (QPS) is normalized to the performance of Elasticsearch.*

Figure 16: **Phrase Query QPS** *The throughput (QPS) is normalized to the performance of Elasticsearch.*

the query throughput of WiSER is 14% worse than Elasticsearch with default prefetching. We found that the difference is related to WiSER's less efficient score calculation, which is not directly linked to I/O.

The query throughput improvement largely comes from the reduced I/O traffic as queries with low popularity levels are I/O intensive and I/O is the system bottleneck. Indeed, we see that the query throughput improvement is highly correlated with the I/O reduction. For example, WiSER's query throughput for popularity level 10 is 2.6 times higher than Elasticsearch's; WiSER's I/O traffic for the same workload is 2.9 times lower than Elasticsearch's.

WiSER achieves better median latency and tail latency than Elasticsearch, thanks to adaptive prefetch and cross-stage data grouping. Figure 14 shows that WiSER achieves up to 16x and 11x lower median latency than Elasticsearch, for median and tail latency respectively. The latency of Elasticsearch is longer due to similar reasons for its low query throughput. Elasticsearch's data layout incurs more I/O requests than WiSER; the time of waiting for page faults adds to the query latency. In contrast, WiSER's more compact data layout and adaptive prefetch incur minimal I/O requests, eliminating unnecessary I/O wait.

Grouped data layout also benefits two-term match queries. Figure 15 presents results for two-term match queries, which are similar to single-term ones. As we have shown in Figure 8 that WiSER reduces by 17% to 51% of I/O traffic for workloads with popularity levels no more than 1,000. As a result, WiSER achieves 1.5x to 2.6x higher query throughput compared with Elasticsearch. When a workload includes popular terms, WiSER's traffic reduction becomes negligible since data grouping has little effects.

4.3.2 Phrase Queries

In this section, we show that our two-way cost-aware Bloom filters make fast phrase query processing possible. Specifically, WiSER can achieve up to 2.7 times higher query throughput and up to 8 times lower latency, relative to Elasticsearch. To support early pruning, WiSER needs to store 9 GB of Bloom filters (the overall index size increases from 18 GB to 27 GB, a 50% increase). We believe such space amplification is reasonable because flash is an order of magnitude cheaper than RAM.
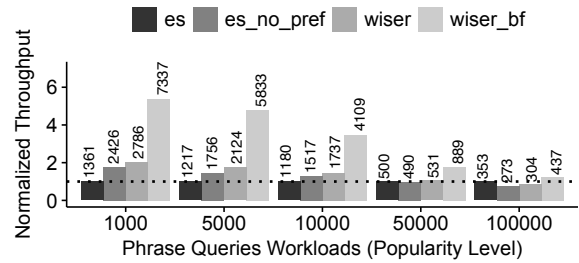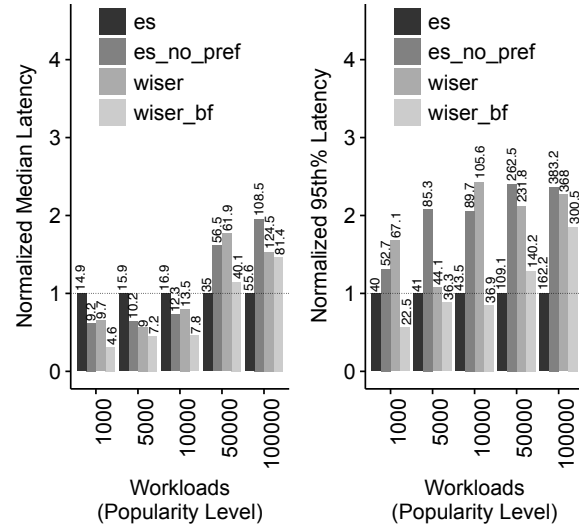
Figure 17: **Phrase Queries Latency** *The latency (ms) is normalized to the corresponding latency of Elasticsearch with default prefetching.*

WSBench produces the phrase query workloads by varying the probability that the two terms in a phrase query become a phrase. WSBench chooses one term from popular terms (popularity level is larger than 10,000); it then varies the popularity level of another term from low to high. As the popularity increases, the two terms are more likely to co-exist in the same document and also more likely to appear as a phrase in the document.

Figure 16 presents the phrase queries results among workloads in Elasticsearch (`es` and `es_no_pref`), WiSER (`wiser`), and WiSER with two-way cost-aware pruning (`wiser_bf`). The QPS of the basic WiSER (`wiser` and Elasticsearch with no prefetch (`es_no_pref`) are similar because our techniques in the basic WiSER (cross-stage data grouping, adaptive prefetch, trading space for less I/O) have little effect for phrase query. WiSER with two-way cost-aware Bloom filters achieves from 1.3x to 2.7x higher query throughput than that of basic WiSER, thanks to significantly lower read amplification brought by the filters.

Figure 17 shows that Bloom filters can significantly reduce latency (`wiser` vs. `wiser_bf`); also WiSER reduces median and tail latency by up to 3.2x and 8.7x respectively, compared to Elasticsearch. The reduction is more evident when
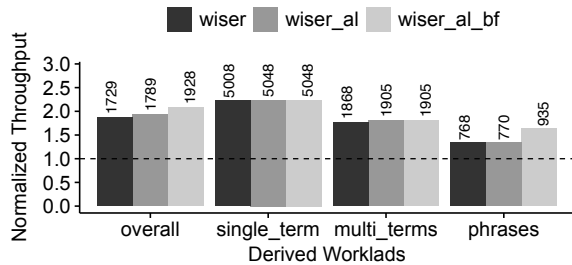
Figure 18: **Throughput of Derived Workloads** *The throughput (QPS) is normalized to the throughput of Elasticsearch without Prefetching*

the probability of forming a phrase is lower (low popularity level) because the Bloom filters are smaller in that case. Interestingly, Elasticsearch with OS prefetch (`es`) achieves the lowest latency when the probability of forming phrases is higher. The latency is lower because the OS prefetches 128 KB of positions data and avoids waiting for many page faults, although the large prefetch increases read amplification. In contrast, Elasticsearch without prefetch (`es_no_pref`) and WiSER do not prefetch; thus they may have to frequently stop query processing to wait for data (WiSER's adaptive prefetch does not prefetch position data due to fragments of unnecessary data.). However, the reduction of latency comes at a cost: although the latency of individual queries is lower, the query throughput is also lower due to the read amplification caused by prefetch (Figure 16).

Interestingly, we find our Bloom filters also speed up the computation of phrase queries. WiSER checks if a phrase exists by Bloom filters, which is essentially O(1) hashing. In the common case that the Bloom filter is empty, WiSER can even check faster because an empty Bloom filter is marked as 0 in the bitmap and we only need to check this bit. As a result, in addition to avoiding reading positions, Bloom filters also allow us to avoid intersecting the positions.

4.3.3 Real Workload

WSBench also includes realistic workloads. We compare Elasticsearch and WiSER's query throughput on the real query log; we also split the query log into different types of query workloads to examine the performance closely.

WiSER performs significantly better than Elasticsearch as shown in Figure 18. For example, for single-term queries, WiSER achieves as high as 2.2x throughput compared to Elasticsearch. We observe that around 60% queries in the real workload are of popularity less than 10,000, which benefits from our cross-stage data grouping. For multi-term match queries, grouped data layout also helps to increase throughput by more than 60%. For phrase queries extracted from the realistic workload, WiSER with Bloom filters increases throughput by more than 60% compared to Elasticsearch. Note that WiSER cannot achieve 2.7x higher throughout as shown in our synthetic phrase queries because, in this real workload, many phrases are the names of people, brand, or events and

so on. Among these names, many terms are unpopular terms that are not I/O intensive, where pruning has limited effect. Finally, the overall performance of WiSER is similar to that of real single-term query log because single-term queries occupy half of the overall query log.

4.4 Scaling with Memory

Our previous experiments showed that WiSER performs significantly better than Elasticsearch for a single small memory size of 512 MB. This small memory size was chosen to stress the I/O performance of each search engine. For our final experiments, we show that we have rebuilt a search engine that can rely less on expensive memory and more on cheaper flash. Over a broader range of memory sizes, WiSER's techniques continue to improve I/O and end-to-end performance. In addition, it shows that WiSER works well with a low `memory size / working set size` ratio, which may allow WiSER to scale to large dataset without increasing much memory.

Figure 19 compares the query throughput, 50-th% query latency, bandwidth, and amount of traffic for Elasticsearch (`es`), WiSER with only cross-stage grouping (`wiser_base`), and fully-optimized WiSER (`wiser_final`). For our two end-to-end metrics, both versions of WiSER have much higher query throughput and much lower query latency than Elasticsearch across all workloads and memory sizes. As expected, query throughput is higher, and latency is lower, when more memory is available. For workload `twoterm`, `single.high`, and `single.low`, our highly efficient implementation allows WiSER with limited memory (e.g., 128 MB) to perform better than Elasticsearch with memory that can hold the entire working set (e.g., 8 GB).

The significant difference in end-to-end performance between WiSER and Elasticsearch for workload `twoterm`, `single.high`, and `single.low` at large memory sizes (Figure 19a and Figure 19b) is attributed to the network issue of Elasticsearch (we have carefully setup tests in Elasticsearch and compared it with similar applications to confirm the issue); with this memory size, I/O is not a bottleneck and our techniques should not make big differences. For the same workloads, we can identify the effects of our techniques in Figure 19d as we reduce the memory sizes, where I/O operations increase due to reduced cache. We can see that when the memory size is big, the traffic sizes between Elasticsearch and WiSER are similar, but WiSER's traffic sizes increase much slower than Elasticsearch's as we reduce memory sizes, thanks to data grouping, adaptive prefetching and trading disk space for I/O. Note that `wiser_final` has more read traffic than `wiser_base` because adaptive prefetch is turned on, which increases the read bandwidth (Figure 19c) and helps with end-to-end performance.

The effect of two-way cost-aware Bloom filters is evident in Figure 19a by comparing `wiser_base` (no Bloom filters) and `wiser_final`. The improvement is up to 1.4x (memory size = 256 MB, note that the improvement here is less than
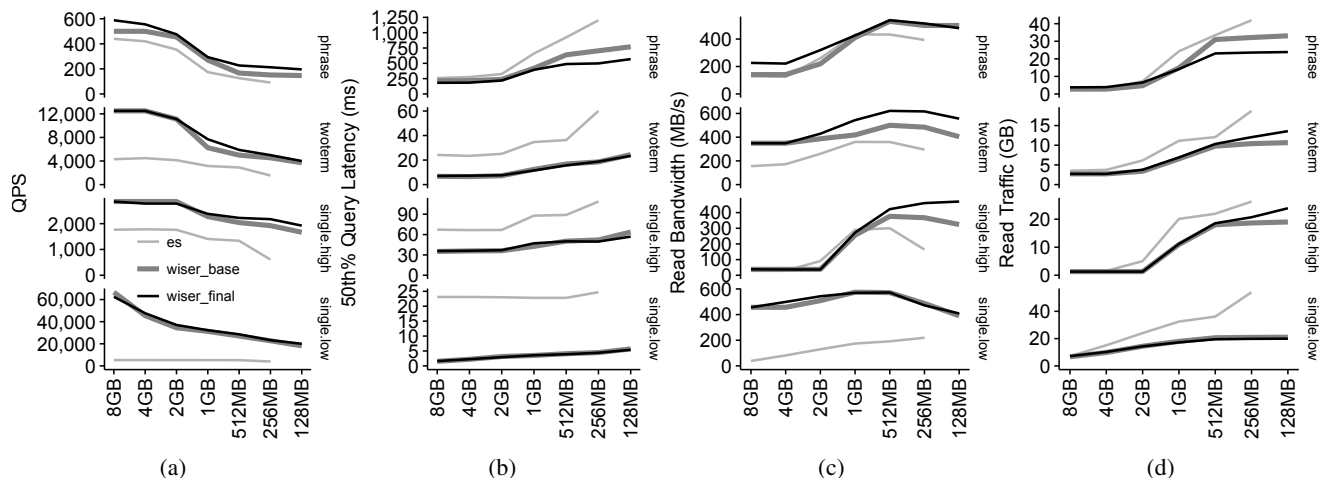
Figure 19: **Performance over a range of memory sizes.** *Elasticsearch fails to run at some low memory sizes and thus some of its data points are missing. Elasticsearch's default prefetch is turned off here because we have found that it hurts performance. Note that we place smaller memory sizes on the right side of the X axes to emphasize the effect of reducing memory sizes.*

the max in Figure 16 because we have queries with mixed popularity levels here). Figure 19d shows the `wiser_final`, the engine with two-way cost-aware Bloom filters, incurs much less I/O than `wiser_base` and `es` when memory is reduced; the reduction is also up to 1.4x (`wiser_base` vs `wiser_final`, memory size = 256 MB). As we can see, when I/O is the bottleneck, the reduction of traffic correlates well with improvement of end-to-end performance.

5   Related Work

Much work has gone into building flash-optimized key-value stores that utilize high-performance SSDs [23, 34, 38, 42]. For example, Wisckey [38] separates keys and values to reduce I/O amplification on SSDs. FAWN-KV [23] is a power-efficient key-value store with wimpy CPUs, small RAM and some flash. Facebook [34] proposes yet another SSD-based key-value store to reduce the consumption of DRAM by small block sizes, aligning blocks and adaptive polling.

Graph applications are also often optimized for SSDs. FlashGraph [59] speeds up graph processing by storing vertices in memory and edges in SSDs. MOSAIC [43] uses locality-optimizing, space-efficient graph representation on a single machine with NVMe SSDs. Many other work also facilitate high performance SSDs [48, 49, 60].

Search engines have different data manipulation and data structures from regular key-value stores and graphs. Among the limited literature, Wang et al. [55] and Tong et al. [52] studied studied search engine cache policies for SSDs; Rui et al. proposes to only cache metadata of snippets in memory and leave the data on SSDs because the I/O cost is reduced [58]. In this paper, we systematically redesign and implement many key data structures and processing algorithms to optimize search engines for SSDs. Such study exposes new opportunities and insights; for example, although using Bloom filters is straightforward in a key-value store, using them in search

engines requires understanding the search engine pipeline, which leads us to the novel two-way cost-aware Bloom filter.

Many proposed techniques for search engines seek to reduce the overhead/cost of query processing [25–28,32,40,51]. These techniques may be adopted in WiSER to further improve its performance.

6   Conclusions

We have built a new search engine, WiSER, that efficiently utilizes high-performance SSDs with smaller amounts of system main memory. WiSER employs multiple techniques, including optimized data layout, a novel Bloom filter, adaptive prefetching, and space-time trade-offs. While some of the techniques could increase space usage, these techniques collectively reduce read amplification by up to 3x, increase query throughput by up to 2.7x, and reduce latency by 16x when compared to the state-of-the-art Elasticsearch. We believe that the design principle behind WiSER, "read as needed", can be applied to optimize a broad range of data-intensive applications on high performance storage devices.

Acknowledgments

References

[1] Apache Lucene. `https://lucene.apache.org/`.

[2] Apache Lucene Index File Formats. `https://lucene.apache.org/core/6_0_0/index.html/`.

[3] Apache Solr. `lucene.apache.org/solr/`.

[4] Breakthrough Nonvolatile Memory Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[5] CloudLab. http://www.cloudlab.us.

[6] DB-Engines Ranking. https://db-engines.com/en/ranking/.

[7] Elasticsearch. https://www.elastic.co/.

[8] Elasticsearch Adhoc Benchmark. https://elasticsearch-benchmarks.elastic.co/no-omit/pmc/index.html.

[9] Full Text Benchmark With Academic Papers from PMC. https://github.com/elastic/rally-tracks/blob/master/pmc/.

[10] Improving Readahead. https://lwn.net/Articles/372384/.

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

[12] Libbloom. https://github.com/jvirkki/libbloom.

[13] Lucene Memory Index. https://lucene.apache.org/core/4_0_0/memory/org/apache/lucene/index/memory/MemoryIndex.html.

[14] Micron NAND Flash Datasheets. https://www.micron.com/products/nand-flash.

[15] RediSearch. redisearch.io/.

[16] RocksDB. https://rocksdb.org.

[17] Samsung 970 EVO SSD. https://www.amazon.com/Samsung-970-EVO-1TB-MZ-V7E1T0BW/dp/B07BN217QG/.

[18] Samsung K9XXG08UXA Flash Datasheet. http://www.samsung.com/semiconductor/.

[19] Samsung Semiconductor. http://www.samsung.com/semiconductor/.

[20] Toshiba Semiconductor. https://toshiba.semicon-storage.com/ap-en/top.html.

[21] WikiBench. http://www.wikibench.eu/.

[22] Wikimedia Moving to Elasticsearch. https://blog.wikimedia.org/2014/01/06/wikimedia-moving-to-elasticsearch/.

[23] David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 1–14, Big Sky, Montana, October 2009.

[24] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[25] Nima Asadi and Jimmy Lin. Fast Candidate Generation for Two-phase Document Ranking: Postings List Intersection with Bloom Filters. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2419–2422, Maui, HI, 2012. ACM.

[26] Nima Asadi and Jimmy Lin. Effectiveness/efficiency Tradeoffs for Candidate Generation in Multi-stage Retrieval Architectures. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 997–1000, Dublin, Ireland, 2013. ACM.

[27] Nima Asadi and Jimmy Lin. Fast Candidate Generation for Real-time Tweet Search with Bloom Filter Chains. *ACM Transactions on Information Systems (TOIS)*, 31(3):13, 2013.

[28] Aruna Balasubramanian, Niranjan Balasubramanian, Samuel J Huston, Donald Metzler, and David J Wetherall. FindAll: a Local Search engine for Mobile Phones. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 277–288. ACM, 2012.

[29] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, pages 359–374, Santa Clara, California, February 2017.

[30] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, pages 181–192, Seattle, Washington, June 2009.

[31] G. G. Chowdhury. *Introduction to Modern Information Retrieval*. Neal-Schuman, 2003.

[32] Austin T Clements, Dan RK Ports, and David R Karger. Arpeggio: Metadata Searching and Content Sharing with Chord. In *International Workshop on Peer-To-Peer Systems*, pages 58–68. Springer, 2005.

[33] Ludovic Denoyer and Patrick Gallinari. The Wikipedia XML Corpus. In *International Workshop of the Initiative for the Evaluation of XML Retrieval*, pages 12–19. Springer, 2006.

[34] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the EuroSys Conference (EuroSys '18)*, page 42, Porto, Portugal, April 2018. ACM.

[35] Evgeniy Gabrilovich and Shaul Markovitch. Computing Semantic Relatedness Using Wikipedia-based Explicit Semantic Analysis. In *IJcAI*, volume 7, pages 1606–1611, 2007.

[36] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the EuroSys Conference (EuroSys '17)*, pages 127–144, Belgrade Serbia, April 2017. ACM.

[37] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. *ACM Transactions on Storage*, 8(4):14, 2012.

[38] Lanyue Lu and Thanumalayan Sankaranarayana Pillai and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 133–148, Santa Clara, California, February 2016.

[39] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 273–286, Santa Clara, California, February 2015.

[40] Jinyang Li, Boon Thau Loo, Joseph M Hellerstein, M Frans Kaashoek, David R Karger, and Robert Morris. On the Feasibility of Peer-to-peer Web Indexing and Search. In *International Workshop on Peer-to-Peer Systems*, pages 207–215. Springer, 2003.

[41] Wentian Li. Random Texts Exhibit Zipf's-law-like Word Frequency Distribution. *IEEE Transactions on information theory*, 38(6):1842–1845, 1992.

[42] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[43] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a Trillion-edge Graph on a Single Machine. In *Proceedings of the EuroSys Conference (EuroSys '17)*, pages 527–543, Belgrade Serbia, April 2017. ACM.

[44] David Milne and Ian H Witten. Learning to Link with Wikipedia. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 509–518. ACM, 2008.

[45] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[46] James K. Mullin. Optimal Semijoins for Distributed Database Systems. *IEEE Transactions on Software Engineering*, (5):558–560, 1990.

[47] Alexander Pak and Patrick Paroubek. Twitter as a Corpus for Sentiment Analysis and Opinion Mining. In *LREc*, volume 10, pages 1320–1326, 2010.

[48] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[49] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 472–488, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[50] Bin Tan and Fuchun Peng. Unsupervised Query Segmentation Using Generative Language Models and Wikipedia. In *Proceedings of the 17th international conference on World Wide Web*, pages 347–356. ACM, 2008.

[51] Nicola Tonellotto, Craig Macdonald, and Iadh Ounis. Efficient and Effective Retrieval Using Selective Pruning. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 63–72. ACM, 2013.

[52] Jiancong Tong, Gang Wang, and Xiaoguang Liu. Latency-aware Strategy for Static List Caching in Flash-based Web Search Engines. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1209–1212. ACM, 2013.

[53] Max Völkel, Markus Krötzsch, Denny Vrandecic, Heiko Haller, and Rudi Studer. Semantic Wikipedia. In *Proceedings of the 15th international conference on World Wide Web*, pages 585–594, 2006.

[54] Jakob Voß. Measuring Wikipedia. *Proceedings of ISSI 2005: 10th International Conference of the International Society for Scientometrics and Informetrics*, 1, 01 2005.

[55] Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, and Xiaoguang Liu. The Impact of Solid State Drive on Search Engine Cache Management. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 693–702. ACM, 2013.

[56] Fei Wu and Daniel S Weld. Autonomously Semantifying Wikipedia. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 41–50, 2007.

[57] 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)*, Renton, WA, July 2019.

[58] Rui Zhang, Pengyu Sun, Jiancong Tong, Rebecca Jane Stones, Gang Wang, and Xiaoguang Liu. Compact Snippet Caching for Flash-based Search Engines. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1015–1018. ACM, 2015.

[59] 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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 45–58, Santa Clara, California, February 2015.

[60] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale Graph Processing on a Single machine Using 2-Level Hierarchical Partitioning. In *Proceedings of the USENIX Annual Technical Conference (USENIX '15)*, pages 375–386, Santa Clara, California, July 2015.

How to Copy Files

Yang Zhan
UNC Chapel Hill and Huawei

Alex Conway
Rutgers Univ.

Yizheng Jiao
UNC Chapel Hill

Nirjhar Mukherjee
UNC Chapel Hill

Ian Groombridge
Pace Univ.

Michael A. Bender
Stony Brook Univ.

Martín Farach-Colton
Rutgers Univ.

William Jannen
Williams College

Rob Johnson
VMware Research

Donald E. Porter
UNC Chapel Hill

Jun Yuan
Pace Univ.

Abstract

Making logical copies, or clones, of files and directories is critical to many real-world applications and workflows, including backups, virtual machines, and containers. An ideal clone implementation meets the following performance goals: (1) creating the clone has low latency; (2) reads are fast in all versions (i.e., spatial locality is always maintained, even after modifications); (3) writes are fast in all versions; (4) the overall system is space efficient. Implementing a clone operation that realizes all four properties, which we call a *nimble clone*, is a long-standing open problem.

This paper describes nimble clones in BetrFS, an open-source, full-path-indexed, and write-optimized file system. The key observation behind our work is that standard copy-on-write heuristics can be too coarse to be space efficient, or too fine-grained to preserve locality. On the other hand, a write-optimized key-value store, as used in BetrFS or an LSM-tree, can decouple the logical application of updates from the granularity at which data is physically copied. In our write-optimized clone implementation, data sharing among clones is only broken when a clone has changed enough to warrant making a copy, a policy we call *copy-on-abundant-write*.

We demonstrate that the algorithmic work needed to batch and amortize the cost of BetrFS clone operations does not erode the performance advantages of baseline BetrFS; BetrFS performance even improves in a few cases. BetrFS cloning is efficient; for example, when using the clone operation for container creation, BetrFS outperforms a simple recursive copy by up to two orders-of-magnitude and outperforms file systems that have specialized LXC backends by 3–4×.

1   Introduction

Many real-world workflows rely on logically copying files and directories. Backup and snapshot utilities logically copy the entire file system on a regular schedule [36]. Virtual-machine servers create new virtual machine images by copying a pristine disk image. More recently, container infrastructures like Docker make heavy use of logical copies to package and deploy applications [34, 35, 37, 44], and new container creation typically begins by making a logical copy of a reference directory tree.

Duplicating large objects is so prevalent that many file systems support *logical* copies of directory trees without making full *physical* copies. Physically copying a large file or directory is expensive—both in time and space. A classic optimization, frequently used for volume snapshots, is to implement copy-on-write (CoW). Many logical volume managers support block-level CoW snapshots [24], and some file systems support CoW file or directory copies [29] via `cp --reflink` or other implementation-specific interfaces. Marking a directory as CoW is quick, especially when the file system can mark the top-level directory as CoW and lazily propagate the changes down the directory tree. Initially, this approach is also space efficient because blocks or files need not be rewritten until they are modified. However, standard CoW presents a subtle tradeoff between write amplification and locality.

The main CoW knob to tune is the copy granularity. If the copy granularity is large, such as in file-level CoW, the cost of small changes is amplified; the first write to any CoW unit is high, drastically increasing update latency, and space is wasted because sharing is broken for *all* data. If the copy granularity is small, updates are fast but fragmented; sequentially reading the copy becomes expensive. Locality is crucial: poor locality can impose a persistent tax on the performance of all file accesses and directory traversals until the file is completely rewritten or the system defragmented [8–10].

Nimble clones. An ideal logical copy—or ***clone***—implementation will have strong performance along several dimensions. In particular, clones should:

- be fast to create;
- have excellent read locality, so that logically related files can be read at near disk bandwidth, even after modification;
- have fast writes, both to the original and the clone; and
- conserve space, in that the write amplification and disk footprint are as small as possible, even after updates to the original or to the clone.
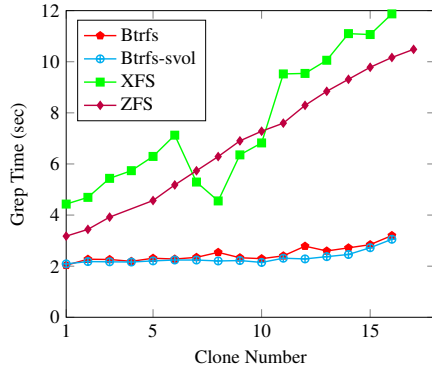
Figure 1: Grep Time for a logically copied 256MiB directory, as a function of the number of prior copies with small edits. (Lower is better.) Btrfs-svol is a volume snapshot, Btrfs and XFS use `cp --reflink`. Full experiment details are in §5.1.

We call a clone with this constellation of features **nimble**. Existing CoW clone implementations are not nimble.

Figure 1 illustrates how performance can degrade using standard CoW techniques in two file systems with copy optimizations. We start by creating a two-level directory hierarchy with 64 4-MiB files (256MiB total), and the experiment proceeds for several rounds. Each round does a volume snapshot or a reflink copy (depending on what the file system supports) and then performs a small, 16-byte edit to each file. We report the time to do a recursive, cold-cache grep over the entire directory at the end of each round. The experiment is detailed further in §5.1.

After each copy and modification, read performance degrades. In the case of XFS and ZFS, we see a factor of 3–4× after only 16 rounds. Btrfs degrades more gradually, about 50% over the same period. In both cases, however, the degradation appears monotonic.

The critical issue here is the need to decouple the granularity of *writes* to a clone from the granularity of *copies* of the shared data. It makes perfect sense to copy a large file that is effectively overwritten. But, for very small changes, it is more IO efficient to keep a "delta" in scratch space until enough changes accrue to justify the cost of a substantial rewrite. In other words, the CoW *copy size* should be tuned to preserve locality (e.g., set to an efficient transfer size for the device), not to whatever granularity a single workload happens to use.

Contributions. In this paper, we present a logical copy specification, which we call a clone, and a set of performance criteria that a *nimble* clone must satisfy. We present the design for a file system and nimble clone implementation that meets all of these criteria.

One key insight into our solution is that the write-optimized message-batching model used in systems such as BetrFS is well suited to decouple writes from copies. There is already a mechanism in place to buffer and apply small changes, although implement the semantics of cloning requires sub-

stantial, additional data-structural work.

We extend BetrFS 0.4, an open-source, full-path-indexed, write-optimized file system. BetrFS performance matches or exceeds other local Linux file systems on a range of applications [8, 17, 39, 42], but BetrFS 0.4 does not support cloning. BetrFS 0.5's clone implements a policy we call **Copy-on-Abundant-Write**, or **CAW**, by buffering small changes to a cloned file or directory in messages until enough changes accrue to warrant the cost of unsharing the cloned data.

This paper also contributes several data-structural techniques to write-optimized dictionaries, in order to implement nimble clones in BetrFS. First, we enable different traversal paths to re-use the same physical data by transforming BetrFS's B^ε-tree [3, 6] data structure into a B^ε-DAG (directed acyclic graph). Second, in order to realize very fast logical copies, we develop new techniques that apply write-optimization, which has previously been used to accelerate changes to *data* stored in the key-value store, towards batching changes to the *topology* of the data structure itself, i.e., its *pivots* and *internal pointers*. An essential limitation of the state of the art, including BetrFS, is that renames, which modify the tree structure, cannot be batched; rather, renames must be completed immediately, including applying all pending changes to the relevant portions of the file system namespace. We introduce a **GOTO message**, which can rapidly persist a logical copy into the message buffer, and is as fast as any small write. With GOTOs, B^ε-DAG-internal housekeeping work is piggy-backed onto any writes to the logically copied region. Third, we introduce a *translation prefix* abstraction that can—at rest—logically correct stale keys in shared data, facilitating both deferred copies and correct queries of partially shared data. As a result of these techniques, BetrFS can rapidly persist large logical copies much faster than the current state of the art (33%–6.8×), without eroding read, write, or space efficiency.

The contributions of this paper are as follows:

- A design and implementation of a B^ε-DAG data structure, which supports nimble CAW clones. The B^ε-DAG extends the B^ε-tree buffered-message substrate to store and logically apply small changes to a clone, until enough changes accrue to warrant the cost of rewriting a clone.
- A write-optimized clone design, wherein one can persist a clone by simply writing a message into the root of the DAG. The work of the clone is batched with other operations and amortized across other modifications.
- An asymptotic analysis, indicating that adding cloning does not harm other operations, and that cloning itself has a cost that is logarithmic in the size of the B^ε-DAG.
- A thorough evaluation of BetrFS, which demonstrates that it meets the nimble performance goals, does not erode the advantages of baseline BetrFS on unrelated workloads, and can improve performance of real-world applications. For instance, we wrote an LXC (Linux Container) backend that uses cloning to create containers, and BetrFS is 3–4×

faster than other file systems with cloning support, and up to 2 orders of magnitude faster than those without.

2 BetrFS Background

This section presents B^ε-tree and BetrFS background that is necessary to understand the cloning implementation presented in the rest of the paper.

BetrFS [17, 18, 39, 40, 42, 43] is an in-kernel, local file system built on a key-value store (KV-store) substrate. A BetrFS instance keeps two KV-stores. The metadata KV-store maps full paths (relative to the mountpoint, e.g., /foo/bar/baz) to struct stat structures, and the data KV-store maps {full path + block number} keys to 4KiB blocks.

The B^ε-tree. BetrFS is named for its KV-store data structure, the B^ε-tree [3, 6]. A B^ε-tree is a write-optimized KV-store in the same family of data structures as an LSM-tree [25] or COLA [2]. Like B-tree variants, B^ε-trees store key-value pairs in leaves. A key feature of the B^ε-tree is that interior nodes buffer pending mutations to the leaf contents, encoded as *messages*. Messages are inserted into the root of the tree, and, when an interior node's buffer fills with messages, messages are *flushed* in large batches to one or more children's buffers. Eventually, messages reach the leaves and the updates are applied. As a consequence, random updates are inexpensive—the B^ε-tree effectively logs updates at each node. And since updates move down the tree in batches, the IO savings grow with the batch size.

A key B^ε-tree invariant is that all pending messages for a given key-value pair are located on the root-to-leaf traversal path that is defined by its key. So a point query needs to read and apply all applicable buffered messages on the traversal path to construct a correct response. Messages have a logical timestamp, and one can think of the contents of these buffered messages as a history of mutations since the last time the leaf was written.

Range operations. BetrFS includes optimizations for contiguous ranges of keys. These are designed to optimize operations on subtrees of the file system namespace (e.g., mv).

Importantly, because BetrFS uses full-path keys, the contents of a directory are encoded using keys that have a common prefix and thus are stored nearly contiguously in the B^ε-tree, in roughly depth-first order. One can read a file or recursively search a directory with a *range query* over all keys that start with the common directory or file prefix. As a result, BetrFS can use a *range delete* message to delete an entire file or recursively (and logically) delete a directory tree with a single message. The range delete is lazily applied to physically delete and recover the space.

Full-path indexing and renaming. Efficient rename operations pose a significant challenge for full-path-indexed file systems. BetrFS has a range rename operation, which can *synchronously* change the prefix of a contiguous range of keys

in the B^ε-tree [42]. In a nutshell, this approach *slices* out the source and destination subtrees, such that there is a single pointer at the same B^ε-tree level to the source and destination subtrees. The range rename then does a "pointer swing", and the tree is "healed" in the background to ensure balance and that nodes are within the expected branching factor. Some important performance intuition about this approach is that the slicing work is logarithmic in the size of the renamed data (i.e., the slicing work is only needed on the right and left edge of each subtree).

BetrFS ensures that range rename leaves most of the *on-disk* subtree untouched by *lifting* out common key prefixes. Consider a subtree T whose range is defined at T's parent by pivots p_1 and p_2. Then the longest common prefix of p_1 and p_2, denoted $\text{lcp}(p_1, p_2)$, must be a prefix of all the keys in T. A lifted B^ε-tree omits $\text{lcp}(p_1, p_2)$ from all keys in T. We say that $\text{lcp}(p_1, p_2)$ has been *lifted out of T*, and that lcp-T is lifted. The lifted B^ε-tree maintains the lifting invariant, i.e. that every subtree is lifted at all times. Maintaining the lifting invariant does not increase the IO cost of insertions, queries, flushes, node splits or merges, or any other B^ε-tree operations.

With the combination of tree surgery and lifting, BetrFS renames are competitive with inode-based file systems [42].

Crash consistency. BetrFS's B^ε-tree nodes are copy-on-write. Nodes are identified using a logical node number, and a *node translation table* maps logical node numbers to on-disk locations. The node translation table also maintains a bitmap of free and allocated disk space. Node writeback involves allocating a new physical location on disk and updating the node translation table. This approach removes the need to modify a parent when a child is rewritten.

All B^ε-tree modifications are logged in a logical redo log. The B^ε-tree is checkpointed to disk every 60 seconds; a checkpoint writes all dirty nodes and the node translation table to disk and then truncates the redo log. After a crash, one need only replay the redo log since the last checkpoint.

Physical space is reclaimed as part of the checkpointing process with the invariant that one can only reuse space that is not reachable from the last stable checkpoint (otherwise, one might not recover from a crash that happens before the next checkpoint). As a result, node reclamation is relatively straightforward: when a node is overwritten, the node translation table tracks the pending free, and then applies that free at the next checkpoint. We note that range delete of a subtree must identify all of the nodes in the subtree and mark them free as part of flushing the range delete message; the node translation table does not store the tree structure.

3 Cloning in BetrFS 0.5

This section describes how we augment BetrFS to support cloning. We begin by defining clone semantics, then describe how to extend the lifted B^ε-tree data structure to a lifted B^ε-

DAG (directed acyclic graph), and finally describe how to perform mutations on this new data structure. The section concludes with a brief asymptotic analysis of the B^ε-DAG.

When considering the design, it helps to differentiate the three layers of the system: the file system directory hierarchy, the KV-store keyspace, and the internal B^ε-tree structure. We first define the clone operation semantics in terms of their effect on file system directory tree. However, because all file system directories and their descendants are mapped onto contiguous KV-store keys based on their full paths, we then focus the BetrFS clone discussion on the KV-store keyspace and the internal B^ε-tree structure implementation.

CLONE operation semantics. A CLONE operation takes as input two paths: (1) a source path—either a file or directory tree root—and (2) a destination path. The file system directory tree is changed so that a logically identical copy of the source object exists at the location specified by the destination path. If a file or directory was present at the destination before the clone, that file or directory is unlinked from the directory tree. The clone operation is atomic.

In the KV-store keyspace, $clone(s,d)$ copies all keys with prefix s to new keys with prefix s replaced with prefix d. It also removes any prior key-value pairs with prefix d.

3.1 Lifted B^ε-DAGs

Our goal in making a lifted B^ε-DAG is to share, along multiple graph traversal paths, a large amount of cloned data, and to do so without immediately rewriting any child nodes. Intuitively, we should be able to immediately add one edge to the graph, and then tolerate and lazily repair any inconsistencies that appear in traversals across that newly added edge. As illustrated in Figure 2, we construct the lifted B^ε-DAG by extending the lifted B^ε-tree in three ways.

First, we maintain reference counts for every node so that nodes can be shared among multiple B^ε-DAG search paths. Reference counts are decoupled from the node itself and stored in the node translation table. Thus, updating a node's reference does not require modifying any node. Whenever a node's reference count reaches zero, we decrement all of its children's reference counts, and then we reclaim the node. Section 4 describes node reclamation.

A significant challenge for sharing nodes in a B^ε-tree or B^ε-DAG is that nodes are large (target node sizes are large enough to yield efficient transfers with respect to the underlying device, typically 2–4MiB) and packed with many key-value pairs, so a given node may contain key-value pairs that belong to unrelated logical objects. Thus, sharing a B^ε-DAG node may share more than just the target data.

For example, in Figure 2, the lower node is the common ancestor of all keys beginning with s, but the subtree rooted at the node also contains keys from q to v. We would like to be able to clone, say, s to p by simply inserting a new edge, with pivots p and pz, pointing to the common ancestor of all
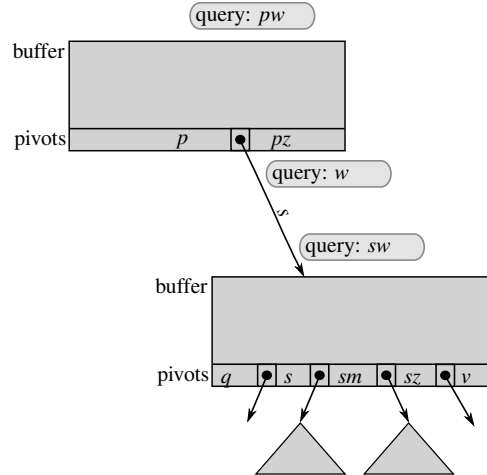
Figure 2: Query processing example in a lifted B^ε-DAG. Initially, the query pw arrives at the parent node. Since the target child's pointer is bracketed by pivots that share the common prefix p (pivots p and pw bracket the pointer to the child), the lifted B^ε-DAG lifts (i.e., removes) the common prefix p from the query term used for searching in the child, transforming the query from pw to w. Next, the query w reaches an edge with translation prefix s. The lifted B^ε-DAG prepends the translation prefix s to the query before continuing to the child. Thus, the query that finally arrives at the child is sw: the common prefix p was lifted out, and the translation prefix s was prepended. The query process proceeds recursively until a terminal node is reached.

s keys but, as the example illustrates, this could have the side effect of cloning some additional keys as well.

Thus, our second major change is to alter the behavior of pivot keys so that they can exclude undesirable keys from traversals. This filtering lets us tolerate unrelated data in a subgraph. A baseline B^ε-tree has an invariant that two pivot keys in a parent node must bound all key-value pairs in their child node (and sub-tree). In the B^ε-DAG, we must relax this invariant to permit node sharing, and we change the graph traversal behavior to simply ignore any key-value pair, message, or pivot that lies outside of the parent pivot keys' range. This partially addresses the issue of sharing a subgraph with extraneous data at its fringe.

The third, related change is to augment each edge with an optional ***translation prefix*** that alters the behavior of traversals that cross the edge. When cloning a source range of keys to a destination, part of the source key may not be lifted. A translation prefix on an edge specifies any remaining part of the source prefix that was not lifted at the time of cloning. As Figure 2 shows, whenever a query crosses an edge with translation prefix s, we prepend s to the query term before continuing to the child, so that the appropriate key-value pairs are found. Once completed, a query removes the translation prefix from any results, before the lifted destination key
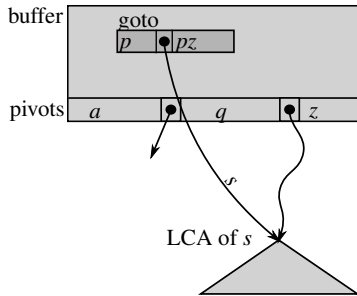
Figure 3: Creating a clone by inserting a GOTO message. Note that the GOTO message's bracketing pivots are (p, pz), and its child pointer contains translation prefix s. The GOTO message supersedes the node's other pivots during a traversal.

along the search path is added back. In the common case, the translation prefix will be NULL.

With these changes—reference counting, filtering pivots, and translation prefixes—a B^ε-DAG can efficiently represent clones and share cloned data among different search paths.

3.2 Creating clones with GOTO messages

To clone all keys (and associated data) with prefix s to new keys with prefix p, we first find the lowest-common ancestor (LCA) of all s keys in the B^ε-DAG, as shown in Figure 3. Intuitively, the LCA is the root of the lowest sub-graph that includes all source keys. We will call the LCA of all s keys node L_s. We then flush to L_s any pending messages for s keys, so that all information about s keys can be found within the sub-DAG rooted at node L_s. We also insert into the root's buffer a GOTO message (described below) for all p keys with target node L_s. We finally increment the reference count of L_s. This completes the cloning process.

GOTO messages. A GOTO message behaves like a pair of bracketing pivots and an associated child pointer. Each GOTO message specifies a range of keys, (a,b); a target height; and a node, U. Whenever a query for some key x reaches a node with a GOTO message, if x falls in the range (a,b), then the query continues directly to node U; said differently, a node's GOTO message supersedes the node's other pivots during a traversal. Like regular pivots, if the two pivots in a GOTO message share a common prefix, then that prefix is removed (lifted) from the query before continuing. Furthermore, like regular child pointers, the pointer in a GOTO message can specify a translation prefix that gets prepended to queries before they continue. Figure 3 illustrates a simple GOTO example, where s is cloned to p. There is a normal child pointer associated with node pivots that bracket prefix s, as well as a GOTO message that redirects queries for p to the LCA of s. In this example, we assume s has not been lifted from the LCA, and, thus, s is used as a translation prefix on the GOTO message.

Flushing GOTO messages. Unlike a regular pair of pivots that

bracket a child pointer, a GOTO message can be flushed from one node to another, just like any other message. Encoding DAG structure inside a message is an incredibly powerful feature: we can quickly persist a logical clone and later batch any post-cloning clean-up work with subsequent writes. When subsequent traversals process buffered messages in logical order, a GOTO takes precedence over all older messages pertaining to the destination keys; in other words, a GOTO implicitly deletes all key-value pairs for the destination range, and redirects subsequent queries to the source sub-graph.

For performance, we ensure that all root-to-leaf B^ε-DAG paths have the same length. Maintaining this invariant is important because, together with the B^ε-DAG's fanout bounds, it guarantees that the maximum B^ε-DAG path has logarithmic length, which means that all queries have logarithmic IO complexity. Thus, we must ensure that paths created by GOTO messages are not longer than "normal" root-to-leaf paths.

This length invariant constrains the minimum height of a GOTO message to be one level above the message's target node, U. At the time we flush to the LCA and create the GOTO message, we know the height of U; as long as the GOTO message is not flushed to the same level as U (or deeper), the maximum query path will not be lengthened.

So, for example, if the root node in Figure 3 is at height 7 and the LCA of s is at height 3, then the GOTO message will get lazily flushed down the tree until it resides in the buffer of some node at height 4. At that point the GOTO will be converted to a regular bracketing pair of node pivots and a child pointer, as shown in Figure 4.

In flushing a GOTO above the target height, the only additional work is possibly deleting obviated internal nodes. In the simple case, where a GOTO covers the same key range as one child, flushing simply moves the message down the DAG one level, possibly lifting some of the destination key. One may also delete messages obviated by the GOTO as part of flushing. The more difficult case is when a GOTO message covers more than one child pointer in a node. In this case, we retain only the leftmost and rightmost nodes. We flush the GOTO to the leftmost child and adjust the pivot keys to include both the left "fringe" and the GOTO message's key range. We similarly adjust the rightmost pivot's keys to exclude any keys covered by the GOTO message (logically deleting these keys, but deferring clean-up). Any additional child pointers and pivots between the left and rightmost children covered by the GOTO are removed and the reference counts on those nodes are reduced by one, effectively deleting those paths.

Converting a GOTO message into node pivots and a child pointer is conceptually similar to flushing a GOTO. As with flushing, a GOTO message takes precedence over any older messages or pre-existing node pivots and child pointers that it overlaps. This means that any messages for a child that are obviated by the GOTO may be dropped before the GOTO is applied.

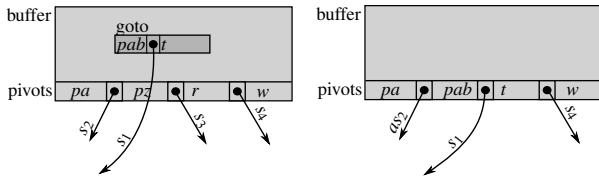The simplest case is where a single child is exactly covered

Figure 4: Converting a GOTO message (left) into a pair of bracketing pivots and a child pointer (right). Note that the GOTO message's pivots pab and t completely cover the range specified by the pre-existing node pivots pz and r, so the GOTO's pivots replace those pivots in the new node (right). Additionally, the translation prefix s_2 is changed to as_2. This is because, in the original node (left), the prefix p is lifted by pivots pa and pz, but in the new node (right), new prefix pa is lifted by pivots pa and pab; a must therefore be prepended to the translation prefix in order to maintain traversal equivalence. (Not shown: the reference counts of covered children are dropped.)

by the GOTO; here, we just replace the pointer and decrement the original child's reference count. For example, in Figure 4, the GOTO message's range (pab, t) completely covers the old pivot range (pz, r). Thus, when converting the GOTO message into regular pivots, we drop the node pointer with translation prefix s_3, and we decrement the reference count of the node to which it pointed.

Partial overlap with a pivot range is handled by a combination of adjusting pivots and adding new pointers. In Figure 4, the GOTO message partially overlaps the old pivot ranges (pa, pz) and (r, w), and there is live data on the "left" fringe of this child (keys between pa and pab are not covered by this GOTO). We modify the original pivot keys so that subsequent traversals through their child pointers only consider live data, but we leave the child nodes untouched and defer physically deleting their data and relifting their keys. Note that in this example, the subtree between updated pivots pa and pb should lift pa instead of just p, so we add a to the translation prefix until the next time this child is actually flushed and re-lifted. We finally replace the covered pivots with new pivot keys and a child pointer for the GOTO's target (the pointer between pab and t in the right portion Figure 4). In the case where a GOTO message overlaps a single child with live data on the left and right fringe (not illustrated), we would create a third pointer back to the original child and increment its reference count accordingly, with appropriate translation prefixes and pivots to only access the live data on the "right" side.

Finally, as with flushing a GOTO, if a GOTO covers multiple children, we remove all of the references to the "interior" children, and replace them with a single child pointer to the GOTO target. We note that this can temporarily violate our target fanout; we allow the splitting and merging process, described next, to restore the target fanout in the background.

3.3 Flushes, splits, and merges

We now explain how node flushes, splits, and merges interact with reference counting, node sharing, translation prefixes, and GOTO messages.

At a high level, we break flushing, splitting, and merging into two steps: (1) convert all involved children into **simple** children (defined below), then (2) apply the standard lifted B^ε-tree flushing, splitting, or merging algorithm.

A child is **simple** if it has reference count 1 and the edge pointing to the child has no translation prefix. When a child is simple, the B^ε-DAG locally looks like a lifted B^ε-tree, so we can use the lifted B^ε-tree flushing, splitting, and merging algorithms, since they all make only local modifications to the tree.

The B^ε-DAG has an invariant that one may only flush into a simple child. Thus, one of two conditions that will cause a node to be made simple is the accumulation of enough messages in the parent of a node—i.e., a copy-on-abundant-write. The second condition that can cause a node to become simple is the need to split or merge the node by the background, healing thread; this can be triggered by healing a node that has temporarily violated the target fanout, or any other condition in the baseline B^ε-tree that would cause a split or merge.

We present the process for converting a child into a simple child as a separate step for clarity only. In our implementation, the work of making a child simple is integrated with the flushing, splitting and merging algorithms. Furthermore, all the transformations described are performed on in-memory copies of the node, and the nodes are written out to disk only once the process is completed. Thus simplifying children does not change the IO costs of flushes, splits, or merges.

The first step in simplifying a child is to make a private copy of the child, as shown in Figure 5. When we make a private copy of the child, we have to increment the reference counts of all of the child's children.

Once we have a private copy of the child, we can discard any data in the child that is not live, as shown in the first two diagrams of Figure 6. For example, if the edge to the child has translation prefix s_1, then all queries that reach the child will have s_1 as a prefix, so we can discard any messages in the child that don't have this prefix, because no query can ever see them. Similarly, we can drop any children of the child that are outside of the range of s_1 keys, and we can update pivots to be entirely in the range of s_1 keys. When we adjust pivots in the child, we may have to adjust some of the child's outgoing translation prefixes, similar to when we converted GOTO messages to regular pivots.

Finally, we can relift the child to "cancel out" the translation prefix on the edge pointing to the child and all the s_1 prefixes inside the child. Concretely, we can delete the s_1 translation prefix on the child's incoming edge and delete the s_1 prefix on all keys in the child.

A consequence of this restriction is that translation prefixes
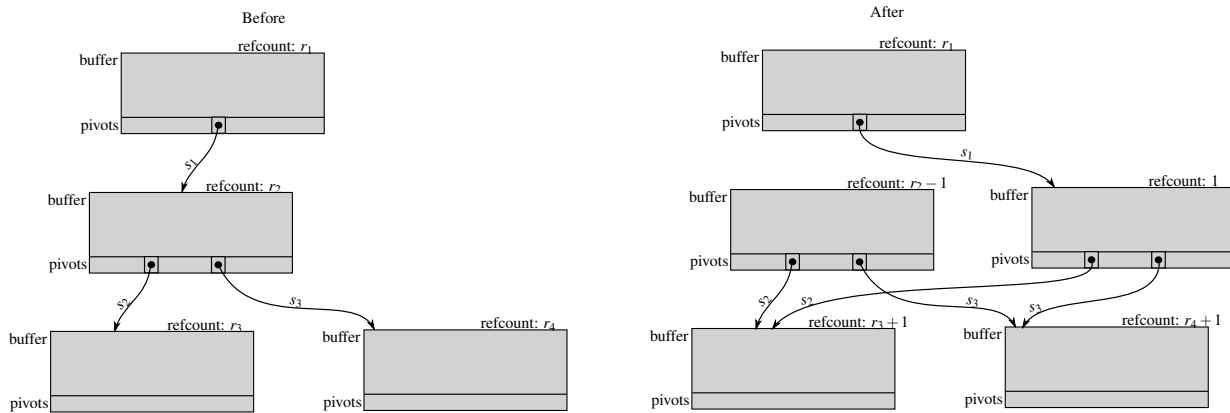
Figure 5: Creating a private copy of a shared child. The original node's contents are copied, and its reference count is decremented. Since the private copy points to all of the original node's children, those children have their reference count increased by one. (Pivot keys are omitted for clarity; they remain unchanged.)
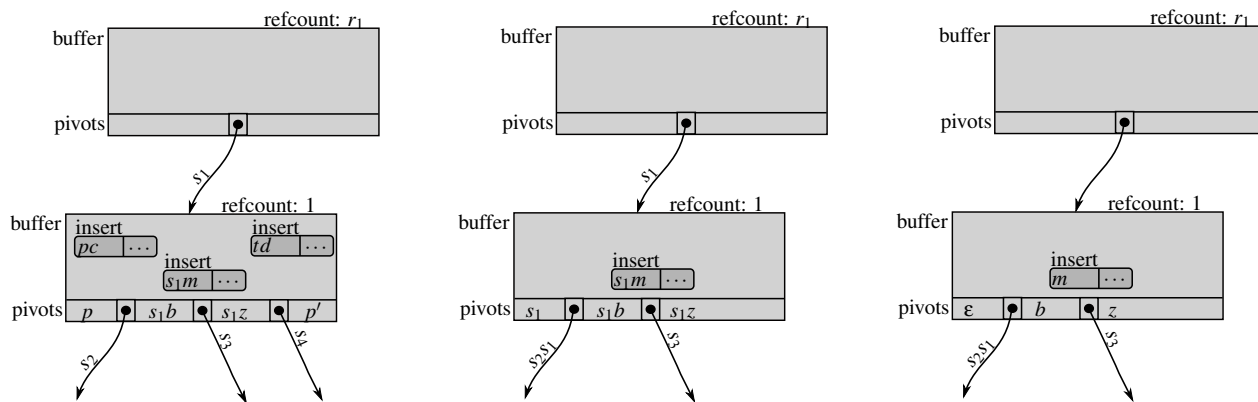
Figure 6: Eliminating a child's translation prefix. The original child node (left) is a private copy with reference count one. First, nodes with unreachable keys are deleted and reclaimed (center). Then the translation prefix s_1 is removed from the incident edge and logically applied to all pivot keys and all keys in buffered messages (right).

should always be NULL after a flush. Intuitively, one only needs a translation prefix to compensate for the effect on lifting of logically deleted data still in a node; after a flush, this data is physically deleted and the node is re-lifted, obviating the need for a translation prefix.

As described, these steps slightly modify the amortized and background work to "heal" irregularities in the B^ε-DAG. This work is primarily driven by subsequent writes to the affected range; a shared node that is not modified on any path can remain shared indefinitely. In our current prototype, we do track shared nodes with very little live data, and mark them for flushing either in the background or under space pressure to reclaim space. The key feature of this design is the flexibility to rewrite nodes only when it is to the advantage of the system—either to reclaim space or recover locality for future queries.

3.4 Putting it all together

The remaining lifted B^ε-tree operations are unchanged in a B^ε-DAG. Inserts, deletes, and clones just add messages to the root node's buffer. When an internal node's buffer becomes full, we flush to one of its children (after making the child simple, if necessary). When a leaf becomes too large or too small, we split or merge it (after making the leaf simple). When an internal node has too many or too few children, we split or merge it (again after making it simple).

3.5 Asymptotic Analysis

This subsection shows that adding cloning does not affect the asymptotics of other operations, and that the cost of a clone is logarithmic in the size of the tree.

Insert, query, and clone complexity all depend on the B^ε-DAG height, which is bounded by the height of a lifted B^ε-tree with the same logical state. To see why, consider the following straightforward transformation of a B^ε-DAG to a B^ε-tree: first

flush all `GOTO` messages until they become regular pivots, then break the CoW sharing of all nodes. Since this conversion can only increase the height of the data structure, a logically equivalent lifted B^ε-tree is at least as tall as a B^ε-DAG.

The height of a B^ε-tree is $O(\log_B N)$, where N is the total number of items that have been inserted into the tree. Hence the height of a B^ε-DAG is $O(\log_B N)$, where N is the number of keys that have been created, either through insertion or cloning, in the B^ε-DAG.

Queries. Since the height of the B^ε-DAG is $O(\log_B N)$, the IO cost of a query is always $O(\log_B N)$.

Insertions. The B^ε-DAG insertion IO cost is the same as in a B^ε-tree, i.e., $O(\frac{\log_B N}{B^{1-\varepsilon}})$. This is because the IO cost of an insertion is $h \times c/b$, where h is the height of the B^ε-DAG, c is the IO cost of performing a flush, and b is the minimum number of items moved to child during a flush. Flushes cost $O(1)$ IOs in a B^ε-DAG, just as in a B^ε-tree, and flushes move at least $\Omega(B/B^{1-\varepsilon})$ items, since the buffer in each node has size $\Omega(B)$, and the fanout of each node is $O(B^\varepsilon)$.

Clones. The cost to create a clone can be broken into the online cost, i.e,. the costs of all tasks that must be completed before the clone is logically in place, and the offline costs, i.e., the additional work that is performed in the background as the `GOTO` message is flushed down the tree and eventually converted to regular pivots.

The online cost of cloning s to d is merely the cost to push all s messages to s's LCA and insert the `GOTO` message. The cost of pushing all the messages to the LCA is $O(\log_B N)$ IOs. Inserting the new `GOTO` message costs less than 1 IO, so the total cost of creating a clone is $O(\log_B N)$ IOs.

The background cost is incurred by the background thread that converts all edges with a translation prefix into simple edges. We bound the IO cost of this work as follows. A clone from s to d can result in edges with translation prefixes only along four root-to-leaf paths in the B^ε-DAG: the left and right fringes of the sub-dag of all s keys, and the left and right fringes of the sub-dag of all d keys. Thus the total IO cost of the background work is $O(\log_B N)$.

4 Implementation and Optimizations

In this section, we describe two optimizations that reduce the total cost of clones. Although they do not alter the asymptotics, we leverage the file system namespace and BetrFS design to save both background and foreground IO.

Preferential splitting. Most background cloning work involves removing unrelated keys and unlifted prefix data from *fringe* nodes, i.e., nodes that contain both cloned and non-cloned data. Thus, we could save work by reducing the number of fringe nodes.

Baseline BetrFS picks the middle key when splits a leaf node. With *preferential splitting*, we select the key that maximizes the common prefix of the leaf, subject to the constraint

that both new leaves should be at least 1/4 full. Since data in the same file share the same prefix (as do files in the same directory), preferential splitting reduces the likelihood of having fringe nodes in a clone.

A naïve approach would compare the central half of all leaf keys and pick the two adjacent keys with the shortest common prefix. However, this scan can be costly. We can implement preferential splitting and only read two keys: because the shortest common prefix among adjacent keys is the same as the common prefix of the smallest and the largest candidate keys (the keys at 1/4 and 3/4 of the leaf), we can construct a good parent pivot from these two keys.

Node reclamation. We run a thread in the background that reclaims any node whose reference count reaches 0. As part of the node reclamation process, we decrement each child node's reference count, including nodes pointed to by `GOTO` messages. Node reclamation proceeds recursively on children whose reference counts reach zero, as well.

This thread also checks any node with a translation prefix. In an extreme case, a node with no reachable data may have a positive reference count due to translation prefixes. For example, if the only incident edge to a sub-DAG has translation prefix s, but no key in the entire sub-DAG has s as a prefix, then all data in the sub-DAG is reclaimable. As part of space reclamation, BetrFS finds and reclaims nodes with no live data, or possibly unshares and merges nodes with relatively little live data.

Concurrency. B-tree concurrency is a classic problem, since queries and inserts proceed down the tree, but splits and merges proceed up the tree, making hand-over-hand locking tricky. B^ε-trees have similar issues, since they also perform node splits and merges, and many of the B-tree-based solutions, such as preemptive splitting and merging [28] or sibling links [20], apply to B^ε-trees, as well.

We note here that our cloning mechanism is entirely top-down. Messages get pushed down to the LCA, GOTO messages get flushed down the tree, and non-simple edges get converted to simple edges in a top-to-bottom manner. Thus cloning imposes no new concurrency issues within the tree.

Background cleaning. BetrFS includes a background process that flushes messages for frequently queried items down the tree. The intention of this optimization is to improve range and point query performance on frequently queried data: once messages are applied to key-value pairs in B^ε-tree leaves, future queries need not reprocess those messages.

We found that, in the presence of clones, this background task increased BetrFS 0.5's space consumption because, by flushing small changes, the cleaner would break B^ε-DAG nodes' copy-on-write sharing.

Thus we modified the cleaner to never flush messages into any node with a reference count greater than 1; such messages instead wait to be flushed in normal write-optimized batches once enough work has accrued to warrant rewriting the node.
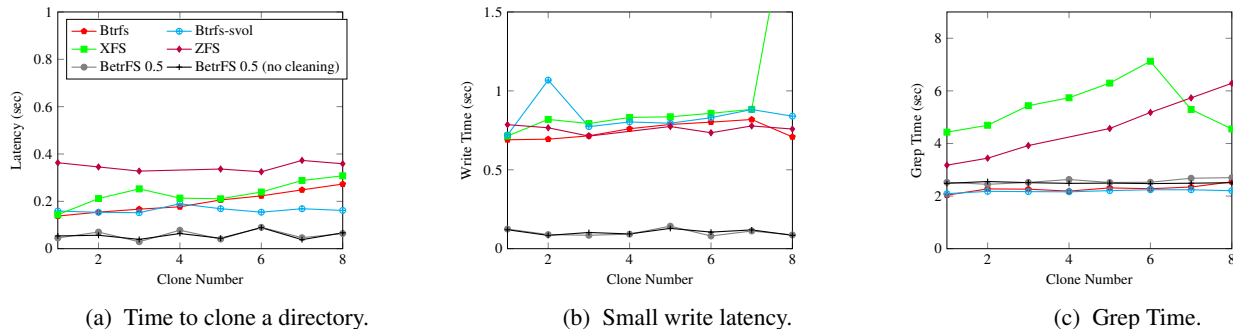
(a) Time to clone a directory. (b) Small write latency. (c) Grep Time.

Figure 7: Latency to clone, write, and read as a function of the number of times a directory tree has been cloned. Lower is better for all measures.

5 Evaluation

This section evaluates BetrFS 0.5 performance. The evaluation centers around the following questions:

- Do BetrFS 0.5 clones meet the performance goals of simultaneously achieving (1) low latency clone creation, (2) reads with good spatial locality, even after modifications, (3) fast writes, and (4) space efficiency? (§5.1)
- Does the introduction of cloning harm the performance of unrelated operations? (§5.2)
- How can cloning improve the performance of a real-world application? (§5.3)

All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4GiB RAM, and a 500GB, 7200 RPM SATA disk, with a 4096-byte block size. We boot from a USB stick with the root file system, isolating the file system under test to only the workload. The system runs 64-bit Ubuntu 14.04.5.

We compare BetrFS 0.5 to baseline BetrFS, ext4, Btrfs, XFS, ZFS, and NILFS2. We used BetrFS version 0.4 from github.com/oscarlab/betrfs, ZFS 0.6.5.11 from zfsonlinux.org and kernel default versions of the other file systems. Unless noted, each experiment was run a minimum of 5 times. We present the mean and display bars that indicate the minimum and maximum times over all runs. Similarly, ± terms bound the minimum and maximum values over all runs. Unless noted, all benchmarks are cold-cache tests.

BetrFS only works on a modified 3.11.10 kernel, so we run BetrFS on that kernel; all other file systems run on 4.9.142. We note that we ran experiments on both kernel versions, and performance was generally better on the newer kernel; we present the numbers for the newer kernel.

5.1 Cloning Performance

To evaluate the performance of cloning (and similar copy-on-write optimizations in other file systems), we wrote a microbenchmark that begins by creating a directory hierarchy with eight directories, each containing eight 4MiB-files. The

microbenchmark then proceeds in rounds. In each round, we create a new clone of the original directory hierarchy and measure the clone operation's latency (Figure 7a). We next write 16 bytes to a 4KiB-aligned offset in each newly cloned file—followed by a sync—in order to measure the impact of copy-on-write (Figure 7b) on writes. We then clear the file system caches and grep the newly copied directory to measure cloning's impact on read time (Figure 7c). Finally, we record the change in space consumption for the whole file system at each step (Table 1). We call this workload *Dookubench*.

We compare directory-level clone in BetrFS 0.5 to 3 Linux file systems that either support volume snapshots (Btrfs and ZFS) or reflink copies of files (Btrfs and XFS). We compare in both modes; the label Btrfs-svol is in volume-snapshot mode. For the file systems that support only file-level clones (XFS and Btrfs without svol), the benchmark makes a copy of the directory structure and clones the files.

For BetrFS 0.5, we present data in two modes. In "no cleaner" mode, we disable the background process in BetrFS 0.5 that flushes data down the tree (Section 4). We found that this background work created a lot of noise in our space experiments, so we disabled it to get more precise measurements. We also run the benchmark in BetrFS 0.5's default mode (with the cleaner enabled). As reported below, the cleaner made essentially no difference on any benchmark, except to increase the noise in the space measurements.

Figure 7a shows that BetrFS 0.5's cloning time is around 60ms, which is 33% faster than the closest data point from another file system (the first clone on XFS), 58% faster than a volume clone on Btrfs, and an order of magnitude faster than the worst case for the competition. Furthermore, BetrFS 0.5's clone performance is essentially flat throughout the experiment. Thus we have achieved our objective of cheap clones. Btrfs and ZFS also have flat volume-cloning performance, but worse than in BetrFS 0.5. Both Btrfs and XFS file-level clone latencies, on the other hand, degrade as a function of the number of prior clones; after 8 iterations, clone latency is roughly doubled.

In terms of write costs, the cost to write to a cloned file or

FS	Δ KiB/round		σ
Btrfs	176	± 112	56.7
Btrfs-svol	32	± 0	0
XFS	32.6	± 95.4	50.9
ZFS	250	± 750	462.9
BetrFS 0.5 (no cleaning)	31.3	± 29.8	19.9
BetrFS 0.5	16.3	± 950.8	460.8

Table 1: Average change in space usage after each Dookubench round (a directory clone followed by small, 4KiB-aligned modifications to each newly cloned file).

volume is flat for all file systems, although BetrFS 0.5 can ingest writes 8–10× faster. Thus we have not sacrificed the excellent small-write performance of BetrFS.

Figure 7c shows that scans in BetrFS 0.5 are competitive with the best grep times from other file systems in our benchmarks. Furthermore, grep times in BetrFS 0.5 do not degrade during the experiment. XFS and ZFS degrade severely—after six clones, the grep time is nearly doubled. For XFS, there appears to be some work that temporarily improves locality, but the degradation trend resumes after more iterations. Btrfs degrades by about 20% for file-level clones and 10% for volume level clones after eight clones. This trend continues: after 17 iterations (not presented for brevity), Btrfs read performance degrades by 50% with no indication of leveling off.

Table 1 shows the change in file system space usage after each microbenchmark round. BetrFS 0.5 uses an average of 16KiB per round, which is half the space of the next best file system, Btrfs in volume mode. BetrFS 0.5's space usage is very noisy due to its cleaner—unsurprisingly, space usage is *less* after some microbenchmark rounds complete, decreasing by up to 693KiB. When the cleaner is completely disabled, space usage is very consistent around 32KiB. Thus enabling the cleaner reduces average space consumption but introduces substantial variation. Overall, these results show that BetrFS 0.5 supports space-efficient clones.

In total, these results indicate that BetrFS 0.5 supports a seemingly paradoxical combination of performance features: clones are fast and space-efficient, and random writes are fast, yet preserve good locality for sequential reads. No other file system in our benchmarks demonstrated this combination of performance strengths, and some also showed significant performance declines with each additional clone.

5.2 General Filesystem Performance

This section evaluates whether adding cloning erodes the performance advantages of write-optimization in BetrFS. Our overarching goal is to build a file system that performs well on *all* operations, not just clones; thus, we measure a wide range of of microbenchmarks and application benchmarks.

Sequential IO. We measure the time to sequentially write a
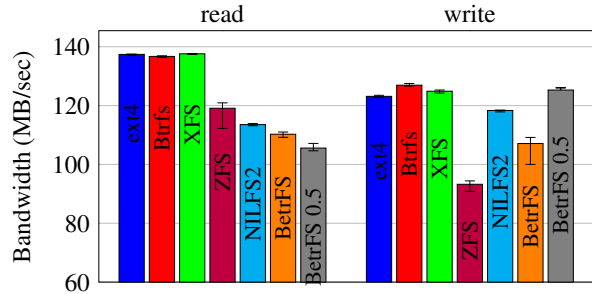
Figure 8: Bandwidth to sequentially read and write a 10 GiB file (higher is better).
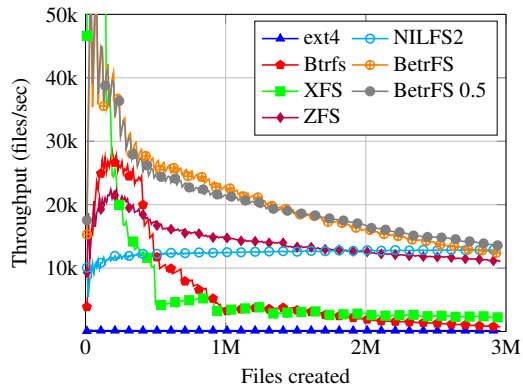
Figure 9: Cumulative file creation throughput during the Tokubench benchmark (higher is better).

10GiB file to disk (the benchmarking machine has only 4GiB of RAM, so this is more than double the available RAM), and then sequentially re-read the data from disk. Figure 8 shows the throughput of both operations. All the filesystems perform sequential IO relatively well. BetrFS 0.5 performs sequential reads at comparable throughput to BetrFS, ZFS, and NILFS2, which is only about 19% less than ext4, Btrfs and XFS. Sequential writes in BetrFS 0.5 are within 6% to the fastest file system (Btrfs). We attribute this improvement to preferential splitting, which creates a pivot matching the maximum file data key at the beginning of the workload, avoiding expensive leaf relifting in subsequent node splits.

Random IO. We measure random write performance with a microbenchmark that issues 256K 4-byte overwrites at random offsets within a 10GiB file, followed by an fsync. This number of overwrites was chosen to run for at least several seconds on the fastest filesystem. Similarly, we measure random read performance by issuing 256K 4-byte reads at random offsets within an existing 10GiB file.

Table 2 shows the execution time of the random write and random read microbenchmarks. BetrFS 0.5 performs these random writes 39–67× faster than conventional filesystems and 8.5% slower than BetrFS. BetrFS 0.5 performs random reads 12% slower than the fastest file system.

Tokubench. We evaluate file creation using the Tokubench benchmark [13]. Tokubench creates three million 200-byte files in a balanced directory tree (no directory is allowed to

FS	random write (s)		random read (s)	
ext4	2770.6	± 21.3	1947.9	± 5.9
Btrfs	2069.1	± 14.6	1907.5	± 6.4
XFS	2863.4	± 14.1	2023.3	± 27.8
ZFS	3410.6	±937.4	2163.9	±112.2
NILFS2	2022.0	± 4.8	1931.1	± 26.6
BetrFS	4.7	± 0.2	2201.1	± 2.9
BetrFS 0.5	5.5	± 0.1	2129.8	± 6.8

Table 2: Time to perform 256K 4-byte random writes/reads (1 MiB total IO, lower is better).

Back-end	FS	lxc-clone (s)		
	ext4	19.514	±	1.214
	Btrfs	14.822	±	0.076
	ZFS	16.194	±	0.538
Dir	XFS	55.104	±	1.033
	NILFS2	26.622	±	0.396
	BetrFS 0.5	8.818	±	1.073
ZFS	ZFS	0.478	±	0.019
Btrfs	Btrfs	0.396	±	0.036
BetrFS 0.5	BetrFS 0.5-clone	0.118	±	0.010

Table 4: Latency of cloning a container.

have more than 128 children). BetrFS 0.5 matches BetrFS throughput, which is strictly higher than any other file system, (except for one point at the end where NILFS2 is 8.7% higher), and as much as 95× higher throughput than ext4.

Directory Operations. Table 3 lists the execution time of three common directory operations—grep, find or delete—on the Linux 3.11.10 kernel source tree.

BetrFS 0.5 is comparable to the baseline BetrFS on all of these operations, with some marginal (4–5%) overhead on grep and delete from adding cloning. We also note that we observed a degradation for BetrFS on larger directory deletions; the degradation is unrelated to cloning and we leave investigation of this for future work. Overall, BetrFS 0.5 maintains the order-of-magnitude improvement over the other file systems on find and grep.

Application Benchmarks. Figure 10 reports performance of the following application benchmarks. We measure two BetrFS 0.5 variants: one with no clones in the file system (labeled BetrFS 0.5), and one executing in a cloned Linux-3.11.10 source directory (labeled BetrFS 0.5-clone).

The `git clone` workload reports the time to clone a local Linux source code repository, which is cloned from github.com/torvalds/linux, and `git diff` reports the time to diff between the v4.14 and v4.7 tags. The tar workload measures the time to `tar` or un-`tar` the Linux-3.11.10 source. The rsync workload copies the Linux-3.11.10 source tree from a source to a destination directory within the same partition and file system. With the `-in-place` option, rsync writes data directly to the destination file rather than creating a temporary file and updating via atomic rename. The IMAP

FS	find (s)		grep (s)		delete (s)	
ext4	2.22	± 0.0	37.71	± 7.1	3.38	± 2.2
Btrfs	1.03	± 0.0	8.88	± 0.3	2.88	± 0.0
XFS	6.81	± 0.2	57.79	±10.4	10.33	± 1.4
ZFS	10.50	± 0.2	38.64	± 0.4	9.18	± 0.1
NILFS2	6.72	± 0.1	8.75	± 0.2	9.41	± 0.4
BetrFS	0.23	± 0.0	3.71	± 0.1	3.22	± 0.4
BetrFS 0.5	0.21	± 0.0	3.87	± 0.0	3.37	± 0.1

Table 3: Time to perform recursive grep, find and delete of the Linux 3.11.10 source tree (lower is better)

server workload initializes a Dovecot 2.2.13 mailserver with 10 folders, each containing 2500 messages, then measures throughput of 4 threads, each performing 1000 operations with 50% reads and 50% updates (marks, moves, or deletes).

In most of these application benchmarks, BetrFS 0.5 is the highest performing file system, and generally matches the other file systems in the worst cases. In a few cases, where the application is write-intensive, such as git clone and rsync, BetrFS 0.5-clone degrades relative to BetrFS 0.5, attributable to the extra work of unsharing nodes, but the performance is still competitive with, or better than, the baseline file systems. These application benchmarks demonstrate that extending write-optimization to include clones does not harm—and can improve—application-level performance.

5.3 Cloning Containers

Linux Containers (LXC) is one of several popular container infrastructures that has adopted a number of storage back-ends in order to optimize container creation. The default backend (dir) does a `rsync` of the component directories into a single, chroot-style working directory. The ZFS and Btrfs back-ends use subvolumes and clones to optimize this process. We wrote a BetrFS 0.5 backend using directory cloning.

Table 4 shows the latency of cloning a default Ubuntu 14.04 container using each backend. Interestingly, BetrFS 0.5 using clones is 3–4× faster than the other cloning backends, and up to two orders of magnitude faster than the others.

6 Related work

File systems with snapshots. Many file systems implement a *snapshot* mechanism to make logical copies at whole-file-system-granularity [27]. Tree-based file systems, like WAFL [15], ZFS [41], and Btrfs [29], implement fast snapshots by copying the root. WAFL FlexVols [12] add a level of indirection between the file system and disks, supporting writable snapshots and multiple active file system instances.

FFS [21] implements read-only file system *views* by creating snapshot inode with a pointer to each disk block; the first time a block is modified, FFS copies the block to a new address and updates the block pointer in the snapshot inode.

(a) Git latency.
Lower is better.

(b) Tar latency.
Lower is better.

(c) Rsync throughput.
Higher is better.

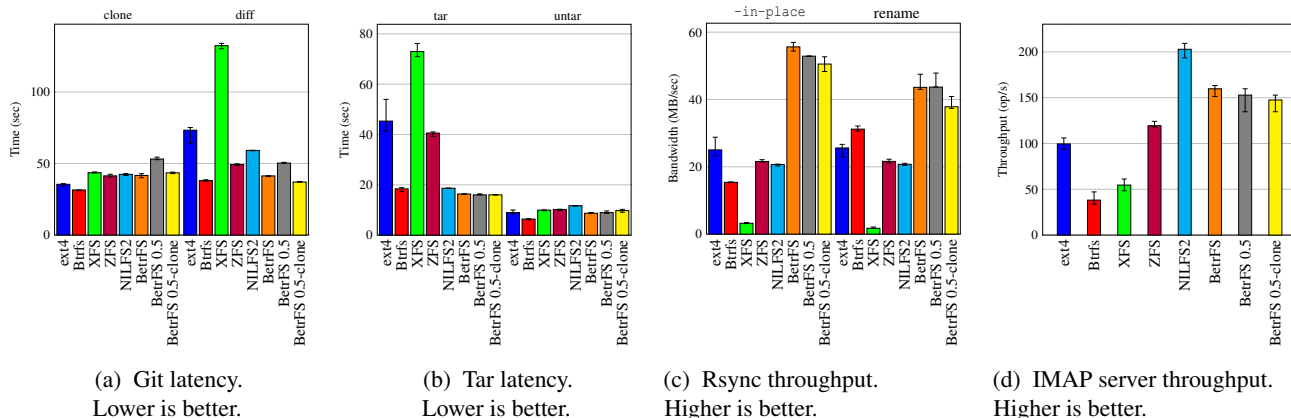(d) IMAP server throughput.
Higher is better.

Figure 10: Application benchmarks.

NILFS [19] is a log-structured file system that writes B-tree checkpoints as part of each logical segment. NILFS can create a snapshot by making a checkpoint block permanent.

GCTree [11] implements snapshots on top of ext4 by creating chains of metadata block versions. Each pointer in the metadata block has a "borrowed bit" to indicate whether the target block was inherited from the previous version. Ext3cow [26] snapshots are defined by an epoch. Ext3cow can render any epoch's file-system view by fetching entries alive at that epoch. NOVA-Fortis [38] supports snapshots by adding private logs to each inode.

File or directory clones. AFS [16] introduced the idea of volumes as a granularity for cloning and backup in a large, distributed file system; volumes isolate performance disruption from cloning one user's data from other users. Episode [7] can create immutable *fileset* clones by copying all the fileset's anodes (inodes) and marking all block pointers copy-on-write. Btrfs [29] can create *file* clones by sharing a file's extents. Windows® 2000 Single Instance Storage (SIS) [5] uses deduplication techniques to implement a new type of link that has copy semantics. Creating the first SIS link requires a complete data copy to a shared store. Writes are implemented copy-on-close: once all open references to an SIS link are closed, sharing is broken at whole-file granularity. Copy-on-close optimizes for the case of complete overwrites.

Versioning file systems. Versioning files is an old idea, dating back to at least TENEX system [4]. Versioning file systems have appeared in a number of OSes [1, 22, 31], but often with limitations such as a fixed number of versions per file and no directory versioning. The Elephant File System [30] automatically versions all files and directories, creating/finalizing a new file version when the file is opened/closed. Each file has an inode log that tracks all versions. CVFS [32] suggests journal-based metadata and multi-version B-trees as two ways to save space in versioning file systems. Versionfs [23] is a stackable versioning file system where all file versions are maintained as different files in the underlying file system.

Exo-clones [33] were recently proposed as a file format for efficiently serializing, deserializing, and transporting volume clones over a network. Exo-clones build upon an underlying file system's mechanism for implementing snapshots or versions. Nimble clones in BetrFS 0.5 have the potential to make exo-clones faster and smaller than on a traditional copy-on-write snapshotting system.

Database indexes for dynamic hierarchical data. The closest work to ours in databases is the BO-tree [14], a B-tree indexing scheme for hierarchical keys that supports moving key subtrees from one place to another in the hierarchy. They even support moving internal nodes of the key hierarchy, which we do not. However, they do not support clones—only moves—and their indexes are not write optimized.

7 Conclusion

This paper demonstrates how to use write-optimization to decouple writes from copies, rendering a cloning implementation with the *nimble* performance properties: efficient clones, efficient reads, efficient writes, and space efficiency. This technique does not harm performance of unrelated operations, and can unlock improvements for real applications. For instance, we demonstrate from 3–4× improvement in LXC container cloning time compared to optimized back-ends. The technique of applying batched updates to the data structure itself likely generalize. Moreover, our cloning implementation in the B$^\varepsilon$-DAG could be applied to any application built on a key-value store, not just a file system.

Acknowledgments

References

[1] Vax/VMS System Software Handbook, 1985.

[2] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *Proc. 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 81–92, 2007.

[3] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. An introduction to B$^\varepsilon$-trees and write-optimization. *:login; Magazine*, 40(5):22–28, Oct 2015.

[4] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson. Tenex, a paged time sharing system for the pdp - 10. *Commun. ACM*, 15(3):135–143, March 1972.

[5] Bill Bolosky, Scott Corbin, David Goebel, and John (JD) Douceur. Single instance storage in windows 2000. In *Proceedings of 4th USENIX Windows Systems Symposium*. USENIX, January 2000.

[6] Gerth Stolting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 546–554, 2003.

[7] Sailesh Chutani, Owen T Anderson, Michael L Kazar, Bruce W Leverett, W Anthony Mason, Robert N Sidebotham, et al. The episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 43–60, 1992.

[8] Alex Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Martin Farach-Colton. File systems fated for senescence? nonsense, says science! In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, pages 45–58, 2017.

[9] Alex Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Martin Farach-Colton. How to fragment your file system. *:login; Magazine*, 42(2):22–28, Summer 2017.

[10] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald Porter, and Martin Farach-Colton. Filesystem aging: It's more usage than fullness. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[11] Chris Dragga and Douglas J. Santry. Gctrees: Garbage collecting snapshots. *ACM Transactions on Storage*, 12(1):4:1–4:32, 2016.

[12] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. Flexvol: Flexible, efficient file volume virtualization in wafl. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 129–142, 2008.

[13] John Esmet, Michael A Bender, Martin Farach-Colton, and Bradley C Kuszmaul. The tokufs streaming file system. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, 2012.

[14] Jan Finis, Robert Brunel, Alfons Kemper, Thomas Neumann, Norman May, and Franz Faerber. Indexing highly dynamic hierarchical data. In *VLDB*, 2015.

[15] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 19–19, 1994.

[16] 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 Transactions on Computer Systems*, 6(1):51–81, 1988.

[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*, pages 301–315, 2015.

[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 Transactions on Storage*, 11(4):18:1–18:29, 2015.

[19] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[20] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems*, 6(4), December 1981.

[21] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 1–17, 1999.

[22] Lisa Moses. TOPS-20 User's manual.

[23] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 115–128, 2004.

[24] Prashanth Nayak and Robert Ricci. Detailed study on linux logical volume manager. *Flux Research Group University of Utah*, 2013.

[25] Patrick O'Neil, Edward Cheng, Dieter Gawlic, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[26] Zachary Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[27] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. In *In Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990.

[28] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage*, 3(4):2:1–2:27, 2008.

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

[30] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 110–123, 1999.

[31] Mike Schroeder, David K. Gifford, and Roger M. Needham. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Opeating Systems Principles*. Association for Computing Machinery, Inc., November 1985.

[32] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, 2003.

[33] Richard P. Spillane, Wenguang Wang, Luke Lu, Maxime Austruy, Rawlinson Rivera, and Christos Karamanolis. Exo-clones: Better container runtime image management across the clouds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.

[34] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Wenji Li, Raju Rangaswami, and Ming Zhao. Evaluating docker storage performance: from workloads to graph drivers. *Cluster Computing*, pages 1–14, 2019.

[35] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. In search of the ideal storage configuration for docker containers. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 199–206. IEEE, 2017.

[36] Veritas. Veritas system recovery. https://www.veritas.com/product/backup-and-recovery/system-recovery, 2019.

[37] Xingbo Wu, Wenguang Wang, and Song Jiang. Totalcow: Unleash the power of copy-on-write for thin-provisioned containers. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 15. ACM, 2015.

[38] 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*, pages 478–496, 2017.

[39] 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*, pages 1–14, 2016.

[40] 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. Writes wrought right, and other adventures in file system optimization. *ACM Transactions on Storage*, 13(1):3:1–3:26, 2017.

[41] ZFS. http://zfsonlinux.org/. Accessed: 2018-07-05.

[42] 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 *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, pages 123–138, 2018.

[43] Yang Zhan, Yizheng Jiao, Donald E. Porter, Alex Conway, Eric Knorr, Martin Farach-Colton, Michael A. Bender, Jun Yuan, William Jannen, and Rob Johnson. Efficient directory mutations in a full-path-indexed file system. *ACM Transactions on Storage*, 14(3):22:1–22:27, 2018.

[44] Frank Zhao, Kevin Xu, and Randy Shain. Improving copy-on-write performance in container storage drivers. Storage Developer's Conference, 2016.

Uncovering Access, Reuse, and Sharing Characteristics of I/O-Intensive Files on Large-Scale Production HPC Systems

Tirthak Patel
Northeastern University

Suren Byna, Glenn K. Lockwood, Nicholas J. Wright
Lawrence Berkeley National Laboratory

Philip Carns, Robert Ross
Argonne National Laboratory

Devesh Tiwari
Northeastern University

Abstract

Large-scale high-performance computing (HPC) applications running on supercomputers produce large amounts of data routinely and store it in files on multi-PB shared parallel storage systems. Unfortunately, storage community has a limited understanding of the access and reuse patterns of these files. This paper investigates the access and reuse patterns of I/O-intensive files on a production-scale supercomputer.

1 Introduction

High-performance computing (HPC) applications running on large-scale facilities routinely perform TBs of I/O. Consequently, significant efforts have been made to study the I/O behavior of HPC systems and workloads in the recent past. Previous studies have attempted to characterize the I/O of workloads based on application-level traces [10,11,17,39], present experimental analysis of factors affecting I/O [35,56–58], and provide guidance for I/O storage systems [29,32–34,54,59]. However, there is limited understanding about how different files produced by HPC systems are re-accessed and re-used, from the same application and across applications. This is primarily because it is fundamentally challenging to measure and collect file-based I/O information across multiple executions as it requires tracing all executions of an application and the affected files which imposes high overhead and hence, is unsuitable for production HPC systems. The benefits of such a study are multi-fold, including understanding the nature of file-specific I/O, uncovering file reuse patterns, studying the effect of I/O variability on I/O performance, and optimizing file placement decisions. However, the costs of conducting such a study are prohibitively high for production systems [4,8,44]. This is one of the major reasons why the community has lacked such an understanding so far.

To the best of our knowledge, this is the first work to perform in-depth characterization and analysis of access, reuse, and sharing characteristics of I/O-intensive files. In particular, this is the first work to characterize (1) whether HPC files are ready-heavy, write-heavy, or both; (2) inter-arrival times for re-access and type of re-access across runs; (3) sharing of a file across multiple applications. Furthermore, our file-based I/O timing analysis also reveals key sources of inefficiencies that cause I/O variability within and across runs.
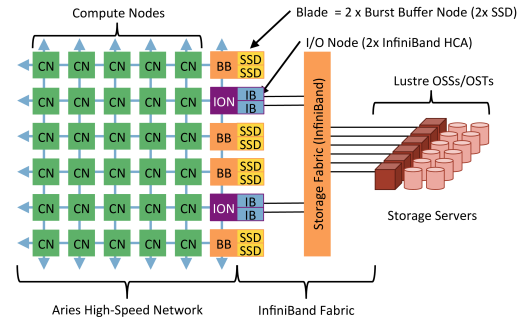
Figure 1: Architecture of the Cori supercomputer [7].

This study was carried out using a lightweight Darshan I/O monitoring tool to trace application I/O on Cori, a leading top 500 supercomputer, for a period spanning four months (Oct'17-Jan'18) during production - covering ≈36 million node-hours of operational system time.

Next, we briefly describe Cori and our methodology.

2 Background and Methodology

Brief Overview of the System. This study is based on a Cray XC40 supercomputer, Cori, ranked at #13 in the Top-500 supercomputers list. Cori achieves the peak computational performance of ≈27 Pflop/s. Cori contains 9,688 Intel Xeon Phi and 2,388 Intel Haswell processors. Fig. 1 shows Cori's network and storage structure. Cori features a disk-based Lustre file system which is composed of ≈10,000 disks organized as 248 Lustre Object Storage Targets (OST). Each OST is configured with GridRAID and has a corresponding Object Storage Server (OSS) for handling I/O requests. The total size of the file system is ≈30 PB with a peak I/O bandwidth of 744 GB/s. During the data collection period of this study, the file system was shared with Edison, an older Cray XC30 system which was near the end of its lifetime (retired in May'19). Edison was comparatively much smaller system (only 2 Pflop/s of peak performance) and generated much lesser I/O traffic compared to Cori as it was also near the end of its lifetime. As Edison was recently decommissioned, we only focus on Darshan logs collected on the Cori system. Cori also has a SSD-based Cray DataWarp burst-buffer storage layer. We note this study does not focus on burst-buffer I/O activities as they are limited (5-15%) and the shared file system observes almost all of the I/O traffic as per Darshan data.
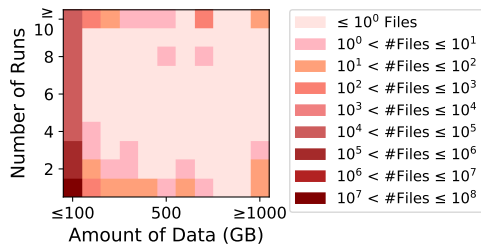
Figure 2: Over 99% of ≈52 million files transferred < 1 GB data and were accessed only once during the study period.
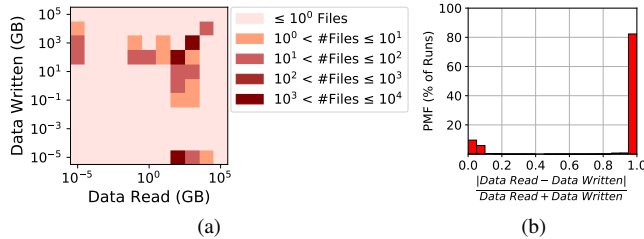
Figure 3: (a) Files can be divided into groups: read-heavy (RH), write-heavy (WH), or both, read- and write- heavy (RW). (b) Difference in data read and written per run shows >82% of runs either perform only read I/O or only write I/O.

Data Collection. We use Darshan, a light-weight I/O monitoring tool which provides application-level I/O tracing capability [11] to collect file I/O access patterns. Darshan V3.10 was enabled by default on Cori for all users during the study period. Darshan reports key information including user id, job id, application (executable) id, start timestamp, end timestamp, and number of processes (ranks). Darshan also traces key statistical metrics for each file at the I/O-software-stack-level for different types of I/O interfaces including POSIX (Portable Operating System Interface) I/O, MPI (Message Passing Interface) I/O, and STD (Standard) I/O. These metrics include amount of read/write data, aggregate time for read/write/meta operations, rank id of I/O performing rank(s), and variance of I/O size and time among different application ranks. Lastly, Darshan also collects Lustre-file-system-level metrics such as stripe width and OST IDs over which a file is striped. However, Darshan does not report actual file sizes, only the size of the data transferred. Over the period of this study, ≈84 million logs (one per execution) were collected with information spanning ≈52 million unique files, 8489 applications, 651 users, and 12.8 PB of data transfer (6.9 PB read data and 5.9 PB write data).

Explanation of Analysis Figures. We now briefly describe the format of the analysis figures used for our study.

Heatmaps. These plots are used to show the significance of a specific relationship between two metrics. The intensity of a heatmap box color indicates the number of files which exhibit the corresponding relationship between the two metrics.

CMF Plots. We use CMF (Cumulative Mass Function) plots to show the cumulative distribution of a metric. A vertical dotted blue line is used to indicate the mean of the distribution.

Some CMF plots show the distribution of the CoV (Coefficient of Variation $(\%) = \frac{\text{standard deviation}}{\text{mean}} \times 100$) of a metric to highlight the normalized variability observed by the metric.

Violin Plots. These plots are used to show the density (in terms of the number of files) for different values of a metric in a vertical format. A horizontal solid blue line is used to indicate the mean of the density distribution.

Next we describe how we select I/O-intensive files, classify these files, and classify the runs which access them.

2.1 Selecting I/O-Intensive Files

As mentioned previously, Cori's Darshan logs contain information about ≈52 million files. However, our analysis shows that a large majority of these files perform very little I/O during the study period. Fig. 2 shows a heatmap of the aggregate amount of data transferred to/from a file vs. the number of runs during which a file is accessed. Most of the files experience less than 100 GB of I/O during the study period and are accessed by only one run. In fact, over 99% of these files transfer less than 1 GB data. Note that this does not mean that the actual file size is less than 1 GB; but the data transfer to/from the file amounts to less than 1 GB.

Therefore, a majority of such files may not be helpful in establishing representative characteristics related to dominant I/O patterns of HPC applications. These files include user notes, scripts, executables, non-I/O-intensive-application outputs, and error logs. Therefore, *our study focuses on a class of "I/O-intensive" files which individually experience data transfer of at least 100 GB during the study period and are accessed by at least 2 runs - to capture the most dominant and representative I/O patterns. From here on, we refer to these I/O-intensive files as "files" simply.* This methodology streamlines our analysis to useful Darshan logs spanning ≈400k runs, 791 applications, 149 users, 8.5k files, and 7.8 PB of data transfer (4.7 PB read data and 3.1 PB write data). We ensured that our analysis is not skewed by only a handful of users performing most of the I/O to these files. In fact, over 70% of selected users perform I/O to more than 2 files, with each user performing I/O to 57 files on average.

2.2 File Classification

Next, we classify I/O-intensive files in terms of the type of I/O they perform. This helps us derive type-specific insights for different types of files in Sec. 3. We study the aggregate amount of read and write data transferred per file. Fig. 3(a) shows a heatmap of the amount of read data transfer vs. the amount of write data transfer. We observe that files can be classified into three distinct clusters. The lower right cluster consists of 22% of the files which transferred mostly read data during the four months. We refer to these as read-heavy or RH files. The upper left cluster consists of 7% of the files which transferred only write data (write-heavy or WH files). Lastly, the cluster in the top right corner with the largest percentage of files (71%), consists of files which are both, read- and write-heavy (referred to as RW files).
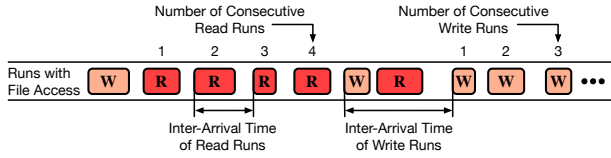
Figure 4: Visual representation of inter-arrival times and number of consecutive runs for both read (R) and write (W) runs.

Finding 1. *HPC files can be classified as read-heavy (RH), write-heavy (WH), or read- and write- heavy (RW). For the first time, we quantify that a significant fraction of the files are read-heavy (22%) and 7% of files are write-heavy - these 7% files are constantly written to but not read, which may indicate unread checkpoint/analysis data. 71% of HPC files are RW files (i.e., both read- and write- heavy). These files may include checkpoint/analysis files which do get read. Such a file classification can be used for file placement decisions in a multi-tier storage system including burst buffers, where each tier is suitable for different kind of I/O operations.*

2.3 Run Classification

While the files can be cleanly classified into three clusters, they can be accessed by multiple "application runs" (simply, referred to as "runs") and can perform both read and write I/O. A run refers to a job running on multiple compute nodes and consisting of multiple MPI processes/ranks and possibly shared-memory threads within a node. We found that a vast majority of runs perform either mostly-read or mostly-write I/O. To demonstrate this, we calculate the difference in the amount of read and write data for each run using the formula: $\frac{|\text{data read}-\text{data written}|}{\text{data read}+\text{data written}}$. The value of this formula ranges from 0 to 1: 1 indicates that all of the data transacted by the run is either exclusively read or exclusively write and 0 indicates equal amount of read and write data transfer. Fig. 3(b) shows that over 82% of all runs have a value very close to 1, i.e., they are either read-intensive or write-intensive. In the context of I/O, we refer to read-intensive runs as simply *"read runs"* and write-intensive runs as *"write runs"*. We found that 69% of all runs are read runs and 31% are write runs. RH files are mostly read by read runs, WH files are mostly written by write runs, and both read and writes runs operate on RW files. This classification helps us establish a producer-consumer relationship among runs in Sec. 3.1.

Finding 2. *Somewhat surprisingly, modern HPC applications largely tend to perform only one type of I/O during a single run: either read or write. This is in contrast to the commonly-held assumption that HPC applications have both read and write I/O phases during the same run [16, 20, 21, 28, 36, 46, 49, 60]. This finding indicates the potential rise of scientific workflows instead of traditional monolithic scientific applications [6, 40, 45]. The presence of non-monolithic applications provides the opportunity to better schedule different components of a large workflow to avoid I/O contention among different workflows.*
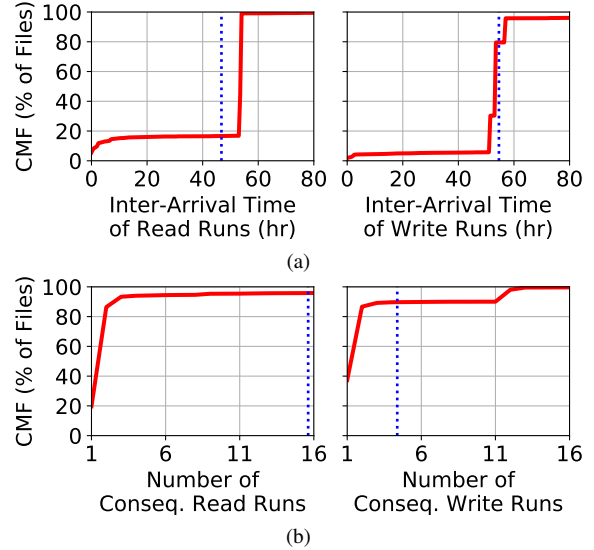
Figure 5: (a) Most of the read and write runs have inter-arrival times of 50-55 hours per file (file re-access interval). (b) The mean number of consecutive read runs in 13 and the mean number of consecutive write runs is 3.

3 Result Discussion and Analysis

In this section, we explore HPC file behavior concerning multi-run reuse and multi-application sharing (Sec. 3.1), and we study I/O data characteristics pertaining to load imbalance and intra- and inter- run I/O variability (Sec. 3.2).

3.1 File Reuse Characteristics

Run Inter-Arrival Times. In Sec. 2.3, we showed that a run can be classified as either read run or write run, and found that the total number of read runs are more than 2x the number of write runs. Now, we study the inter-arrival times of these different runs to understand the avgerage time taken to reuse the same file (inter-arrival time is defined as shown in Fig. 4). Fig. 5(a) shows that the mean inter-arrival time of read runs experienced by a file is 47 hours, while that of write runs is 55 hours. But, on an average, 80% of files are re-accessed only after 50-55 hours for both read and write runs. We note that the average inter-arrival time is much longer than the average runtime of jobs on Cori (e.g., >80% of HPC jobs on these systems finish in less than 2 hours) [3, 43].

Finding 3. *Read and write runs have similar inter-arrival times of over 2 days for 80% of the files. For the first time we find that most files get re-accessed after a relatively long period (>50 hours) - much longer than the runtime of jobs. This enables opportunity for data compression [18] of files which are expected to remain inactive for some time and also leverage transparent burst-buffer prefetching and caching [9, 47] for files expected to be accessed in a short while.*

Consecutive Runs of the Same I/O Type. Read and write runs having similar inter-arrival times motivates us to test
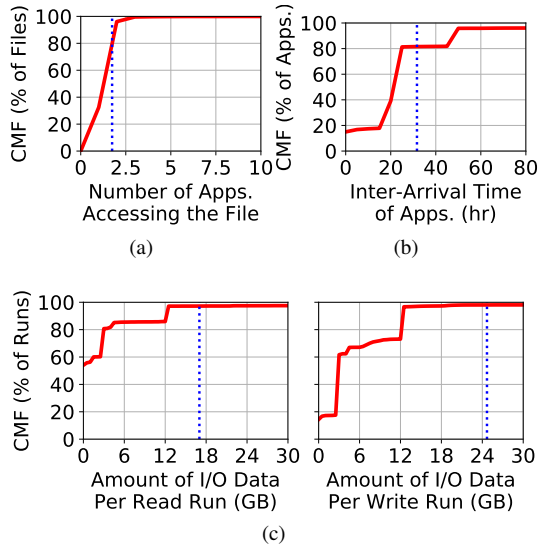
Figure 6: (a) Over 65% of files are accessed by at least 2 applications. (b) The average inter-arrival time of each application to perform I/O to a specific file is 31 hours. (c) On average, read runs transfer 17 GB of data per run, while write runs transfer 25 GB of data per run.

if read and write runs are scheduled back to back, and if so, how long do these sequences last. We calculate the average number of consecutive read runs and write runs for each file (as shown in Fig. 4) and plot the distribution in Fig. 5(b). Over 80% of files experience 2 or more consecutive read runs and over 65% of files experience 2 or more consecutive write runs. A majority of files experience 2 consecutive read runs (65%) and 2 consecutive write runs (50%). This suggests that files get accessed in alternating phases of multiple read runs and multiple write runs - consistent with our observation that RW files dominate the population (71%). However, there are many files which experience a large number of consecutive read runs (due to RH files). In fact, the mean number of consecutive read runs experienced by a file is over 14, while the mean number of consecutive write runs is < 4. There are only 2.2× as many read runs as write runs (Sec. 2.3), but mean number of consecutive read runs is 4.3× the number consecutive write runs. This indicates that data is produced a few times, and then consumed many times over, true for most RW files. This observation suggests that scientific simulations often produce data during certain runs, which is then used as a driver input by several subsequent runs to explore different potential paths or analyze a simulated phenomena in detail. We note that consecutive write runs does not imply that all the previously written data is rewritten/lost. Some scientific workflows could append a file over two consecutive write runs and then, read a part of the file in the subsequent run.

Finding 4. *HPC files experience a few consecutive write runs and a long string of consecutive read runs on average. This insight can help leverage MPI "hints" [38] to guide the system about the type of I/O about to be executed. Partitioning*
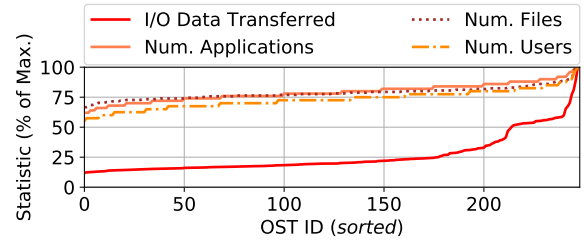
Figure 7: The amount of I/O data transferred by each OST is largely unequal, even though the number of files, applications, and users are more balanced due to capacity balancing.

of I/O servers [25] to separately serve RH files (which perform many consecutive reads) and RW files (for read and write runs) can boost I/O performance.

Multi-Application File Sharing. Taking the producer-consumer relationship one step further, it would be interesting to understand if the producer and the consumer are the same application or if they are different applications. From a methodological point of view, we note that all applications which access a file are run by the same user. So for any file, both producer and consumer applications belong to the same user. Also, a file is not considered to be shared by default among multiple users due to permission issues. Fig. 6(a) shows the CMF of the number of applications which access a file. Over 67% of files are accessed by at least 2 applications, thus indicating that files are often shared by multiple applications. Fig. 6(b) shows the CMF of the inter-arrival time of each application which performs I/O to a file. The mean inter-arrival time of each application is 31 hours, which is much lower than the mean inter-arrival time of individual read and write runs (>50 hours). Thus, for most files, 2 or more applications serve as the producer and the consumer, as opposed to a single application performing I/O to the file. This is consistent with our finding that a majority (86%) of files accessed by multiple applications are RW files (only 12% of these shared files are RH files and only 2% are WH files).
Finding 5. *HPC files are shared by multiple applications and each application performs both read and write I/O serving as both, the producer and the consumer. Inter-arrival times of these runs also indicate that the producer and the consumer are launched significantly apart in time - limiting the effectiveness of potential caching across applications.*

3.2 Characteristics of I/O Data Accesses

Per Run I/O Data Transfer. In Sec. 3.1, we studied how files get used over multiple runs. We now investigate how the data transaction characteristics change over these multiple runs. Fig. 6(c) shows a CMF of the amount of data transferred per run by read runs and write runs. We observe that on average, read runs transfer 17 GB of data per run, while write runs transfer 25 GB of data per run. In fact, 50% of read runs transfer less than 1 GB of data.
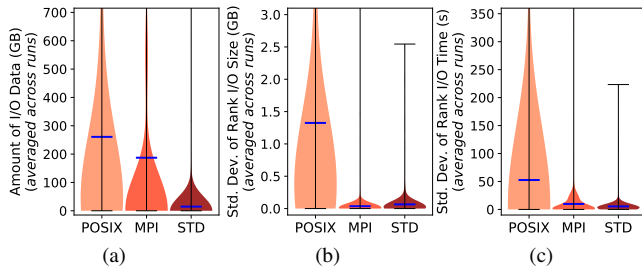
Figure 8: (a) POSIX and MPI I/O interfaces are used to transfer the most amount of data. (b) The variability in I/O size among different ranks of the same application is very small. (c) But, the variability in I/O time of individual ranks is large.

Finding 6. *While reads runs are more abundant than write runs and transfer more data in total, surprisingly, write runs transfer more amount of data than read runs per run. On average, write runs perform 1.4× the I/O of read runs per run. This finding can be exploited to manage I/O contention better at the system-level by limiting the number of concurrently executing write runs. Recall that our earlier finding indicates that HPC applications largely tend to perform only one type of I/O during one run and hence, "write runs" can easily be detected and classified.*

Spatial Load Imbalance. Now that we have found that different runs transfer different amount of data, the next question to investigate is how this difference affects the back-end OSTs. Fig. 7 shows the normalized I/O data transferred to/from each of the OSTs during the study period. Interestingly, there is a large spread in how much data is transferred by each OST. The least "active" OST is only 13% as active as the most active OST. On the other hand, when we look at the number of files on each OST, number of applications which use these files, and number of users which generate the files, we see that the spread is much lower.

Finding 7. *For the Lustre-based system studied in this work, OSTs are capacity-balanced to ensure approximately equal utilization at the file creation time, but that does not guarantee dynamic load-balance. Consequently, there is large inequality in terms of the amount of load (data transfer) which each OST observes over time - emphasizing the need for dynamic file migration (currently lacking in the Lustre file system), replication of read-only data, and caching.*

Intra-Run I/O Variability. Next, we look at how varying OST contention can affect the I/O time of concurrently running ranks (processes) within a run as these ranks could be performing I/O to different OSTs in parallel. For this analysis, we individually analyze the three different I/O interfaces used at Cori: POSIX I/O, MPI I/O, and STD I/O. First, we look at the amount of data transferred using each interface. Fig. 8(a) shows that POSIX is the most commonly used I/O interface transferring about 260 GB of data per run per file on average. Thereafter, MPI interface is used to transfer about 190 GB of
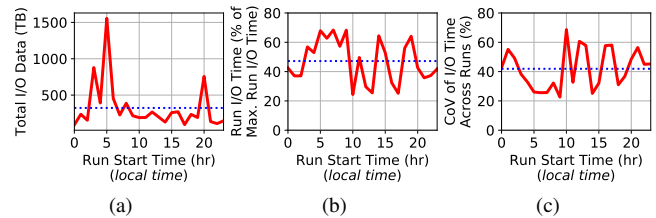
Figure 9: (a) A large amount of I/O data is transferred during 3am-5am local time. (b) Due to this, runs take the most amount of time to complete their I/O during the corresponding hours. (c) Also, variability in I/O time is lower when I/O time is higher and higher when I/O time is lower.

data per run per file on average. STD is the least commonly used interface, as is expected for parallel HPC applications.

Fig. 8(b) shows the standard deviation of the amount of data transferred across each rank performing I/O per run per file. On average, this standard deviation is very small across all three interfaces. For example, the average standard deviation of the amount of data transferred across POSIX I/O performing ranks is less than 1.5 GB, which is negligible compared to the average amount of data transferred using POSIX (260 GB). On the other hand, Fig. 8(c) shows the standard deviation of the I/O time across each rank performing I/O per run per file. This standard deviation is especially high for I/O performed using POSIX interface. This is because, typically when using the POSIX interface, each rank performs I/O to its own file, while when using the MPI I/O interface, all ranks perform I/O to a shared file. Because the default stripe width on the Cori supercomputer is 1, over 99% files are striped across only 1 OST. Therefore, if an application performs I/O to multiple files in parallel, they tend to perform I/O to multiple OSTs in parallel, as the files could be mapped to different OSTs. Thus, varying levels of resource contention at these OSTs can dramatically affect the I/O time of the individual ranks when using POSIX I/O.

Finding 8. *OST load imbalance leads to a high degree of variability in I/O time of ranks which are concurrently performing I/O, especially if the ranks are performing I/O to different OSTs, which is largely the case with POSIX I/O. This leads to the faster ranks having to wait for the slower ranks to finish I/O before they can resume computation, thus wasting precious compute cycles on the HPC system.*

Temporal Load Imbalance. Previously, we discovered that OST I/O imbalance and contention causes intra-run variability in I/O time. So the next step is to explore the temporal characteristics of I/O load. Fig. 9(a) shows the total amount of data which is transferred at different hours of the day. We observe that the largest amount of I/O activity is performed by runs which start between 3am and 5am local time. Note that Cori has users across the globe, so the specific local time (i.e., early morning) is not an indicator of when the local users are the most active. We plot the amount of data with respect to the

start time of the run which is sufficient for our analysis. We note that our following analysis does not necessarily establish a causal relationship between different factors, but instead attempts to explain the observed trends. In Fig. 9(b), we plot the I/O time of runs across different hours of the day. The I/O time of a run is plotted as percentage of the maximum I/O time among all runs which perform I/O to the same file to normalize it across files. However, we observe that runs started during 3am-5am and a few hours post 5am have the highest runtime due to the high I/O activity during this time. This is in spite of the fact that runs performing I/O to the same file have low variability in terms of the amount of data they transfer (as we will discuss later).

Interestingly, Fig. 9(c) shows that while the variability in I/O time is generally significantly high across all times (>20%), it is the lowest for runs which start during peak I/O activity periods. The CoV is calculated among runs belonging to the same file which start during the same hour of the day. The CoV of I/O time plot has a near opposite trend as that of the I/O time plot (Fig. 9(b)). In fact, the I/O time and CoV of I/O time have a Spearman Correlation Index of -0.94, which points to strong negative correlation. This indicates that when the I/O activity is highest, the variability in I/O time that the user can expect is slightly lower, i.e., if user A starts the same run every day during a high I/O activity period, they can expect less variability in the runs' I/O times (and therefore, runtimes) than user B who starts the same run every day during a low I/O activity period. Of course, the trade-off is that user A observes a higher I/O time on average than user B. This happens because when the I/O activity is high, the OSTs are heavily contended which may slow down all I/O. Hence, the effect of any variation in I/O time is small. However, when OSTs are not contended and I/O is faster, the effects of variation are more pronounced and noticeable.

Finding 9. *Temporal load imbalance causes I/O time of the same run to be different during different times of the day. Moreover, variability in I/O time is strongly negatively correlated with the I/O time during the time of the day. HPC systems need new techniques to mitigate the intra-run variability (i.e., ranks of the same application finishing at different times) which continues to have a considerable presence since the I/O variability is significant at all times (>20%).*

Inter-Run I/O Variability. The next question we address is that if there is temporal imbalance in storage system load, does it cause I/O time variability from one run to another? Note that the variability we addressed in Finding 9 was among runs starting during the same hour. Now we look at all runs accessing the same file regardless of their start times. First, we explore how much the amount of data transferred to/from the same file changes from one run to another. Fig. 10(a) shows the CMF of the CoV of the amount of I/O data transferred across runs for each file. Overall, more than 80% of files have a CoV of less than 5% which indicates a negligible
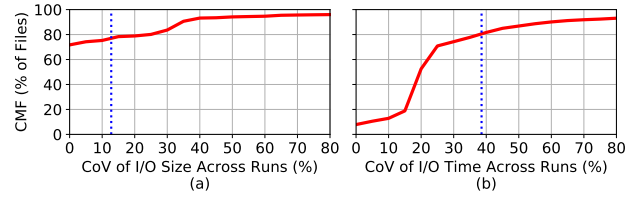
Figure 10: The change in the amount of data transferred across runs to read-only files is the smallest, but these files experience the highest variability across runs in terms of I/O time. Overall, the change in the amount of data is very small (mean CoV is 12%); however, the change in the amount of time it takes to transfer the data is much greater (mean CoV is 39%).

change in the amount of I/O data transferred from one run to another. This is especially true for RH files, and even true for RW files, which experience both, read runs and write runs, thus indicating that similar amount of data gets produced and consumed in a vast majority of cases. WH files exhibit the highest variability in the amount of data transferred (mostly write data in the case of WH files) with a mean CoV of 35% (results for different types of files are not shown for brevity).

Fig. 10(b) shows the CoV of I/O time for different runs for each file. Across all files, even though the amount of data does not change significantly from one run to another, the amount of time it takes to transfer this data experiences significant variability: the mean CoV of the I/O time across runs is 39%. RH files experience the most change in I/O time from one run to another with a mean CoV of 68%, even though they have the least change in the amount of data transferred. This is due to the fact that the OSTs experience different levels of contention at different times due to temporal load imbalance. In fact, because read runs transfer less amount of data on average than write runs (as we discussed in Finding 6), the effect of this load imbalance is especially prominent on their I/O time, which in turn has the largest impact on RH files.

Finding 10. *HPC files tend to experience similar amount of data transfer from one run to another, but they do experience a large variability in terms of the amount of time taken to transfer the data. This is especially true for ready-heavy files which have the least variability in I/O data, but the most variability in I/O time - indicating the need for special attention to RH files when mitigating I/O variability challenge.*

4 Scope of the Findings

While we have ensured that our results and insights are statistically significant, certain aspects of our study may limit the applicability and generalization ability of our analysis.

User Opt-Out. Cori users had the option to opt out of Darshan logging. However, the Darshan library is enabled by default for all users. Therefore, a large majority of users, especially the ones running I/O intensive applications, run Darshan during execution to understand their I/O behavior.

Time Period of Data Collection. Our study uses four months of data logs for analysis and is unable to detect trends longer

than four months. However, four months is a long period and all of the insightful findings such as read and write runs inter-arrival times, multiple application inter-arrival times, and temporal load imbalance are in the order of hours. We also note that the jobs on the Cori supercomputer do not exhibit significant seasonal behavior. That is, the I/O traffic remains relatively similar throughout the year, as also confirmed by previous studies [42]. Therefore, we do not expect our analysis and findings to be affected by the time period of the study.

Unavailable Information. Our study is restricted by the type of information traced by Darshan. Therefore, we are unable to study file size, file amendments/overwrites, number of nodes involved in I/O, and batch job I/O behavior. Information about random vs. sequential I/O type is available for POSIX I/O, but does not yield interesting results as we found that almost all of the I/O is sequential as is expected for HPC applications.

"What if?" Analysis. Our post-event analysis also bars us from posing "what if" questions such as what if a particular run is removed from analysis? How would it affect the I/O trends? Such questions are not possible to study retroactively in a parallel storage system as all concurrently running applications affect each other's I/O behavior in complex ways which cannot be decoupled easily.

Impact of Cori-specific environment and workloads. As expected, our findings are influenced by the nature of workloads which are executed at National Energy Research Scientific Computing Center (NERSC) and the NERSC system environment where Cori is hosted. Consequently, we caution that our findings cannot be generalized to other HPC systems as-is, but this work provides a methodological framework to conduct a study of this nature at other centers to confirm and refute the presented findings.

However, we also note that similarities between NERSC and other centers are likely since HPC users often tend to run workloads with similar characteristics [34]. Workloads running at NERSC are diverse in nature and correspond to a wide variety of scientific domains such as material science, cosmology, combustion, fluid dynamics, climate science, and quantum simulations. Prior studies have covered various aspects of these workloads [5, 31, 34, 53].

Increase in data analytics workloads may be the reason for read-heavy file I/O. Wide increase of such workloads on leading HPC centers has been observed in recent years [1, 12, 13]. NERSC has observed a rise in data analytics workloads in NERSC Exascale Science Applications Program (NE-SAP) [14]. Data and learning applications such as BD-CATS which run at NERSC are quite I/O-intensive. Interestingly, we also observed that some applications that generate large amounts of read data (QCD and quantum modeling of materials) do not necessarily come from the data analytics domain and have run at NERSC for many years. Finally, we note that the scope of this study is limited to only the NERSC system where the instrumentation was performed.

5 Related Work

In this section, we discuss and contrast some related work.

I/O Characterization Software. As HPC I/O has become more unstable and a bigger performance bottleneck over the last few years, much effort is geared toward developing I/O characterization tools for individual applications [10, 11, 22, 50, 55] and for the entire system [2, 23, 24, 51, 58]. Recent works focus on developing software for end-to-end characterization of I/O [15, 30, 31, 41, 48, 58]. These works deal with tool development and do not provide detailed analysis of I/O behavior, especially in terms of file access and reuse.

I/O Behavior Analysis. Most analysis works study the I/O behavior of individual applications and/or runs such as I/O periodicity, bandwidth characteristics, and inter/intra application execution I/O variability [19, 26, 29, 32–34, 37, 54, 56, 57, 59]. Variability and I/O characterization studies performed by some of these previous works are restricted to analyzing a few benchmarks as they do not have access to a system-level view of hundreds of concurrently running HPC applications. Apart from analyzing application-level I/O logs, works by Liu et al. and Madireddy et al. [27, 28, 35] also examine storage server logs to assess application I/O characteristics. In fact, many studies focus extensively on the storage system's I/O behavior [17, 21, 39, 51, 52] by exploring optimal file-system configurations or identifying system-level topology bottlenecks. Above works do not consider multi-fold interactions related to HPC files such as file re-access, multi-application file sharing, run classification and inter-arrival, spatial- and temporal- load imbalance, and intra- and inter- run variability.

6 Conclusion

Overall, our analysis of Darshan I/O logs on the Cori supercomputer reveals many previously unexplored and unexpected insights. We found that files which contribute the most to HPC I/O are not only re-accessed in more ways than one but are also shared across applications. They follow a producer-consumer relationship with runs which extensively write to the files and runs which extensively read them. We explored why and how these files have large intra- and inter-run variability not in terms of I/O size, but in terms of I/O time.

References

[1] NERSC 2017 Annual Report. `https://www.nersc.gov/assets/Uploads/2017NERSC-AnnualReport.pdf`, 2017.

[2] Anthony Agelastos, Benjamin Allan, Jim Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, et al. The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications. In *SC'14*, pages 154–165. IEEE, 2014.

[3] Gonzalo Pedro Rodrigo Alvarez, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. Towards Understanding Job Heterogeneity in HPC: A NERSC Case Study. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 521–526. IEEE, 2016.

[4] George Amvrosiadis, Ali R Butt, Vasily Tarasov, Erez Zadok, and Ming Zhao. Data Storage Research Vision 2025 Report. *Technical Report*, 2019.

[5] Brian Austin, Tina Butler, Richard Gerber, Cary Whitney, Nicholas Wright, Woo-Sun Yang, and Zhengji Zhao. Hopper Workload Analysis. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2014.

[6] Fayssal Benkhaldoun, Christophe Cérin, Imad Kissami, and Walid Saad. Challenges of Translating HPC Codes to Workflows for Heterogeneous and Dynamic Environments. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 858–863. IEEE, 2017.

[7] Wahid Bhimji, Deborah Bard, David Paul, Melissa Romanus, et al. Accelerating science with the NERSC Burst Buffer Early User Program. In *Cray User Group (CUG)*, 2016.

[8] A Brinkmann, K Mohror, and W Yu. Challenges and Opportunities of User-Level File Systemsfor HPC. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.

[9] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. Parallel I/O Prefetching using MPI File Caching and I/O Signatures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 44. IEEE Press, 2008.

[10] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and Improving Computational Science Storage Access through Continuous Characterization. *ACM Transactions on Storage (TOS)*, 7(3):8, 2011.

[11] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 Characterization of Petascale I/O Workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE, 2009.

[12] Steven WD Chien, Stefano Markidis, Vyacheslav Olshevsky, Yaroslav Bulatov, Erwin Laure, and Jeffrey S Vetter. TensorFlow Doing HPC. *arXiv preprint arXiv:1903.04364*, 2019.

[13] Steven WD Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luis Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure. Characterizing Deep-Learning I/O Workloads in TensorFlow. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 54–63. IEEE, 2018.

[14] Jack Deslippe, Doug Doerfler, Brian Friesen, Yun Helen He, Tuomas Koskela, Mathieu Lobet, Tareq Malas, Leonid Oliker, Andrey Ovsyannikov, Samuel Williams, et al. Analyzing Performance of Selected NESAP Applications on the Cori HPC System. In *High Performance Computing: ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers*, volume 10524, page 334. Springer, 2017.

[15] Sheng Di, Rinku Gupta, Marc Snir, Eric Pershey, and Franck Cappello. Logaider: A Tool for Mining Potential Correlations of HPC Log Events.

[16] Hassan Eslami, Anthony Kougkas, Maria Kotsifakou, Theodoros Kasampalis, Kun Feng, Yin Lu, William Gropp, Xian-He Sun, Yong Chen, and Rajeev Thakur. Efficient disk-to-disk sorting: A case study in the decoupled execution paradigm. In *Proceedings of the 2015 International Workshop on Data-Intensive Scalable Computing Systems*, page 2. ACM, 2015.

[17] Raghul Gunasekaran, Sarp Oral, Jason Hill, Ross Miller, Feiyi Wang, and Dustin Leverman. Comparative I/O Workload Characterization of Two Leadership Class Storage Clusters. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 31–36. ACM, 2015.

[18] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. I/O Acceleration with Pattern Detection. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 25–36. ACM, 2013.

[19] Dan Huang, Qing Liu, Jong Choi, Norbert Podhorszki, Scott Klasky, Jeremy Logan, George Ostrouchov, Xubin He, and Matthew Wolf. Can I/O Variability Be Reduced on QoS-Less HPC Storage Systems? *IEEE Transactions on Computers*, 68(5):631–645, 2018.

[20] Ye Jin, Xiaosong Ma, Mingliang Liu, Qing Liu, Jeremy Logan, Norbert Podhorszki, Jong Youl Choi, and Scott Klasky. Combining Phase Identification and Statistic Modeling for Automated Parallel Benchmark Generation. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):309–320, 2015.

[21] Youngjae Kim and Raghul Gunasekaran. Understanding I/O Workload Characteristics of a Peta-scale Storage System. *The Journal of Supercomputing*, 71(3):761–780, 2015.

[22] Michelle Koo, Wucherl Yoo, and Alex Sim. I/O Performance Analysis Framework on Measurement Data from Scientific Clusters. 2015.

[23] Julian M Kunkel, Michaela Zimmer, Nathanael Hübbe, Alvaro Aguilera, Holger Mickler, Xuan Wang, Andriy Chut, Thomas Bönisch, Jakob Lüttgau, Roman Michel, et al. The SIOX Architecture–Coupling Automatic Monitoring and Optimization of Parallel I/O. In *International Supercomputing Conference*, pages 245–260. Springer, 2014.

[24] Julian Martin Kunkel, Eugen Betke, Matt Bryson, Philip Carns, Rosemary Francis, Wolfgang Frings, Roland Laifer, and Sandra Mendez. Tools for Analyzing Parallel I/O. In *International Conference on High Performance Computing*, pages 49–70. Springer, 2018.

[25] Chih-Song Kuo, Aamer Shah, Akihiro Nomura, Satoshi Matsuoka, and Felix Wolf. How File Access Patterns Influence Interference Among Cluster Applications. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 185–193. IEEE, 2014.

[26] Qing Liu, Norbert Podhorszki, Jeremy Logan, and Scott Klasky. Runtime I/O Re-Routing+ Throttling on {HPC} Storage. In *Presented as part of the 5th {USENIX} Workshop on Hot Topics in Storage and File Systems*, 2013.

[27] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces. In *FAST*, volume 14, pages 213–228, 2014.

[28] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. Server-Side Log Data Analytics for I/O Workload Characterization and Coordination on Large Shared Storage Systems. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 819–829. IEEE, 2016.

[29] Glenn K Lockwood, Shane Snyder, Teng Wang, Suren Byna, Philip Carns, and Nicholas J Wright. A Year in the Life of a Parallel File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 74. IEEE Press, 2018.

[30] Glenn K Lockwood, Nicholas J Wright, Shane Snyder, Philip Carns, George Brown, and Kevin Harms. TOKIO on ClusterStor: Connecting Standard Tools to Enable Holistic I/O Performance Analysis. 2018.

[31] Glenn K Lockwood, Wucherl Yoo, Suren Byna, Nicholas J Wright, Shane Snyder, Kevin Harms, Zachary Nault, and Philip Carns. UMAMI: A Recipe for Generating Meaningful Metrics through Holistic I/O Performance Analysis. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pages 55–60. ACM, 2017.

[32] Uri Lublin and Dror G Feitelson. The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, 2003.

[33] Jakob Lüttgau, Shane Snyder, Philip Carns, Justin M Wozniak, Julian Kunkel, and Thomas Ludwig. Toward Understanding I/O Behavior in HPC Workflows. In *Proc. of Workshop in conjunction with ACM/IEEE Supercomputing Conference, Dallas, TX, USA*, 2018.

[34] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. A Multiplatform Study of I/O Behavior on Petascale Supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 33–44. ACM, 2015.

[35] Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan M Wild. Analysis and Correlation of Application I/O Performance and System-Wide I/O Activity. In *Networking, Architecture, and Storage (NAS), 2017 International Conference on*, pages 1–10. IEEE, 2017.

[36] Anirban Mandal, Paul Ruth, Ilya Baldin, Yufeng Xin, Claris Castillo, Mats Rynge, and Ewa Deelman. Evaluating i/o aware network management for scientific workflows on networked clouds. In *Proceedings of the Third International Workshop on Network-Aware Data Management*, page 2. ACM, 2013.

[37] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 409–425, 2018.

[38] John M May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann, 2001.

[39] S. Oral et al. Best Practices and Lessons Learned from Deploying and Operating Large-Scale Data-Centric Parallel File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 217–228. IEEE, 2014.

[40] Suraj Pandey, Karan Vahi, Rafael Ferreira da Silva, Ewa Deelman, Ming Jiang, Cyrus Harrison, Al Chu, and Henri Casanova. Event-Based Triggering and Management of Scientific Workflow Ensembles. In *HPC Asia*, 2018.

[41] Byung H Park, Saurabh Hukerikar, Ryan Adamson, and Christian Engelmann. Big Data Meets HPC Log Analytics: Scalable Approach to Understanding Systems at Extreme Scale. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 758–765. IEEE, 2017.

[42] Tirthak Patel, Suren Byna, Glenn K Lockwood, and Devesh Tiwari. Revisiting I/O Behavior in Large-Scale Storage Systems: The Expected and the Unexpected. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2019.

[43] Gonzalo P Rodrigo, P-O Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. Towards Understanding HPC Users and Systems: A NERSC Case Study. *Journal of Parallel and Distributed Computing*, 111:206–221, 2018.

[44] Robert Ross, Lee Ward, Philip Carns, Gary Grider, Scott Klasky, Quincey Koziol, Glenn K Lockwood, Kathryn Mohror, Bradley Settlemyer, and Matthew Wolf. Storage Systems and I/O: Organizing, Storing, and Accessing Data for Scientific Discovery. Technical report, USDOE Office of Science (SC)(United States), 2019.

[45] Mats Rynge, Scott Callaghan, Ewa Deelman, Gideon Juve, Gaurang Mehta, Karan Vahi, and Philip J Maechling. Enabling Large-Scale Scientific Workflows on Petascale Resources using MPI Master/Worker. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*, page 49. ACM, 2012.

[46] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. Adaptive load balancing in disk arrays. In *International Conference on Foundations of Data Organization and Algorithms*, pages 345–360. Springer, 1993.

[47] Seetharami Seelam, I-Hsin Chung, John Bauer, and Hui-Fang Wen. Masking I/O Latency using Application Level I/O Caching and Prefetching on Blue Gene systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[48] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. 2010.

[49] Evgenia Smirni and Daniel A. Reed. Lessons From Characterizing the Input/Output Behavior of Parallel Scientific Applications. *Performance Evaluation*, 33(1):27–44, 1998.

[50] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright. Modular HPC I/O Characterization with Darshan. In *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, pages 9–17. IEEE, 2016.

[51] Sudharshan S Vazhkudai, Ross Miller, Devesh Tiwari, Christopher Zimmer, Feiyi Wang, Sarp Oral, Raghul Gunasekaran, and Deryl Steinert. GUIDE: A Scalable Information Directory Service to Collect, Federate, and Analyze Logs for Operational Insights into a Leadership HPC Facility. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 45. ACM, 2017.

[52] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S Vazhkudai. Improving Large-Scale Storage System Performance via Topology-Aware and Balanced Data Placement. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 656–663. IEEE, 2014.

[53] Teng Wang, Suren Byna, Glenn K Lockwood, Shane Snyder, Philip Carns, Sunggon Kim, and Nicholas J Wright. A Zoom-in Analysis of I/O Logs to Detect Root Causes of I/O Performance Bottlenecks. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 102–111, 2019.

[54] Teng Wang, Shane Snyder, Glenn Lockwood, Philip Carns, Nicholas Wright, and Suren Byna. IOMiner: Large-Scale Analytics Framework for Gaining Knowledge from I/O Logs. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 466–476. IEEE, 2018.

[55] Steven A Wright, Simon D Hammond, Simon J Pennycook, Robert F Bird, JA Herdman, Ian Miller, A Vadgama, Abhir Bhalerao, and Stephen A Jarvis. Parallel File System Analysis through Application I/O Tracing. *The Computer Journal*, 56(2):141–155, 2012.

[56] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing Output Bottlenecks in a Supercomputer. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[57] Bing Xie, Yezhou Huang, Jeffrey S Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. Predicting Output Performance of a Petascale Supercomputer. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 181–192. ACM, 2017.

[58] Bin Yang, Xu Ji, Xiaosong Ma, Xiyang Wang, Tianyu Zhang, Xiupeng Zhu, Nosayba El-Sayed, Haidong Lan, Yibo Yang, Jidong Zhai, et al. End-to-End I/O Monitoring on a Leading Supercomputer. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 379–394, 2019.

[59] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Rob Ross, and Gabriel Antoniu. On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 750–759. IEEE, 2016.

[60] Zhou Zhou, Xu Yang, Dongfang Zhao, Paul Rich, Wei Tang, Jia Wang, and Zhiling Lan. I/O-Aware Batch Scheduling for Petascale Computing Systems. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 254–263. IEEE, 2015.

GIFT: A Coupon Based Throttle-and-Reward Mechanism
for Fair and Efficient I/O Bandwidth Management on Parallel Storage Systems

Tirthak Patel
Northeastern University

Rohan Garg
Nutanix

Devesh Tiwari
Northeastern University

Abstract

Large-scale parallel applications are highly data-intensive and perform terabytes of I/O routinely. Unfortunately, on a large-scale system where multiple applications run concurrently, I/O contention negatively affects system efficiency and causes unfair bandwidth allocation among applications. To address these challenges, this paper introduces GIFT, a principled dynamic approach to achieve fairness among competing applications and improve system efficiency.

1 Introduction

Problem Space and Gaps in Existing Approaches. Increase in computing power has enabled scientists to expedite the scientific discovery process, but scientific applications produce more and more analysis and checkpoint data, worsening their I/O bottleneck [7, 45]. Many applications spend 15-40% of their execution time performing I/O, which is expected to increase for exascale systems [12, 15, 22, 31, 53, 55]. Unfortunately, multiple concurrent applications on a large-scale system lead to severe I/O contention, limiting the usability of future HPC systems [11, 45].

Recognizing the importance of the problem, there have been numerous efforts to mitigate I/O contention from both I/O throughput and fairness perspectives [13, 14, 17, 25, 37, 42, 75, 76, 78, 88, 89]. Unfortunately, ensuring fairness and maximizing throughput are conflicting objectives, and it is challenging to strike a balance between them under I/O contention. For parallel HPC applications, the side-effect of I/O contention is further amplified because of the need for *synchronous I/O progress*. HPC applications are inherently tightly synchronized; during an I/O phase, MPI processes of an HPC application must wait for all processes to finish their I/O before resuming computation (i.e., synchronous I/O progress among MPI processes is required) [28, 31, 39, 57, 90].

MPI processes of an HPC application perform parallel I/O access to multiple back-end storage targets (e.g., an array of disks) concurrently. These back-end storage targets are shared among concurrently running applications and have different degree of sharing over time and hence, a varying level of contention. A varying level of I/O contention at the shared back-end parallel storage system makes different MPI processes progress at different rates and hence, leads to non-synchronous I/O progress. In Sec. 2, we quantify non-synchronous I/O progress as a key source of inefficiency in shared parallel storage systems. It results in (1) wastage of compute cycles on compute nodes, and (2) reduction in effective system I/O bandwidth (i.e., the bandwidth that contributes toward synchronous I/O progress), since full bandwidth is not utilized toward synchronous I/O progress.

Recent works have noted that non-synchronous I/O progress degrades application and system performances on modern supercomputers like Mira, Edison, Cori, and Titan [9, 31, 32, 39, 69, 83]. Thus, there is an emerging interest in improving the quality-of-service (QoS) of parallel storage systems [24, 80, 86]. Previous works have proposed rule-based or ad-hoc bandwidth allocation strategies for HPC storage [14, 17, 23, 36, 42, 88, 89]. However, existing approaches do not systematically implement synchronous I/O progress to balance the competing objectives: improving effective system I/O bandwidth and improving fairness.

To bridge this solution gap, *this paper describes GIFT, a coupon-based bandwidth allocation approach to ensure synchronous I/O progress of HPC applications while maximizing I/O bandwidth utilization and ensuring fairness among concurrent applications on parallel storage systems.*

Summary of the GIFT Approach. GIFT introduces two key ideas: (1) *Relaxing the fairness window:* GIFT breaks away from the traditional concept of instantaneous fairness at each I/O request, and instead, ensures fairness over multiple I/O phases and runs of an application. This opportunity is enabled by exploiting the observation that HPC applications have multiple I/O phases during a run and are highly repetitive, often exhibiting similar behavior across runs; and (2) *Throttle-and-reward approach for I/O bandwidth allocation:* GIFT opportunistically throttles the I/O bandwidth of certain applications at times in an attempt to improve the overall effective system I/O bandwidth (i.e., it minimizes the wasted I/O bandwidth that does not contribute toward synchronous I/O progress). GIFT's throttle-and-reward approach intelligently exploits *instantaneous* opportunities to improve effective system I/O bandwidth. Further, relaxing the fairness window enables GIFT to reward the "throttled" application at a *later point* to ensure fairness.
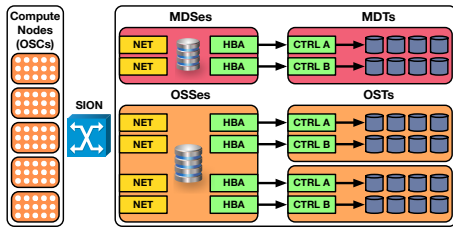
Figure 1: Overview of HPC storage system architecture.

First, GIFT allocates I/O bandwidth to all competing applications in a fair manner and ensures synchronous I/O progress among all processes of the same application at *all times* - this fundamental design principle eliminates the key source of parallel storage system inefficiencies (Sec. 3.1). This allows GIFT to estimate the amount of wasted I/O bandwidth (i.e., bandwidth which does not contribute toward the synchronous I/O progress). Then, GIFT exploits the "opportunity" to reduce the bandwidth waste by identifying and throttling the I/O bandwidth share of some applications and expanding the I/O bandwidth share of other applications (Sec. 3.2). To minimize the I/O bandwidth waste, GIFT uses constraint-based, linear programming to optimally allocate bandwidths to applications (Sec. 3.4). GIFT issues "coupons" to the throttled applications – the worth of these coupons is proportional to the degree of throttling. At a later point, GIFT "redeems" the previously issued coupons to throttled applications to ensure fairness (Sec. 3.3). In cases where GIFT cannot redeem issued coupons for an application, it rewards the application with proportional compute node-hours (credited from a bounded "system regret budget"). This system regret budget acts as a credit bank of compute node-hours, which GIFT uses to achieve fairness when coupons cannot be redeemed.

The contributions of GIFT include:

Design and Implementation. GIFT designs and develops an efficient and practical coupon-based management system for I/O bandwidth allocation among competing applications on shared parallel storage systems. GIFT develops new lightweight and effective techniques to identify throttle-friendly applications, determine the degree of throttling and expansion of I/O bandwidth share of competing applications, and redeem coupons to ensure fairness. GIFT shows that the usage of the "system regret budget" upon failure to redeem coupons is minimal, and that the compute node-hours required for the system regret budget are much less than compared to the increase in system throughput due to faster I/O. GIFT implements all the core ideas in a real-system prototype based on the FUSE file system, demonstrating that GIFT's' ideas can be realized in practice, open to the community for reproducibility, and do not require heroic optimization efforts or system-specific parameter tunings to realize the performance gains. GIFT is available at `https://github.com/GoodwillComputingLab/GIFT`.

Evaluation of GIFT. Our evaluation confirms that GIFT reduces the "bandwidth waste" caused by I/O contention on a HPC storage system, and thereby, improves the I/O bandwidth utilization toward synchronous I/O progress, application performance and fairness, and system job throughput. Our evaluation is based on extensive real system experimental results, guided by real-world, large-scale HPC system and application parameters, and supported by simulation results. GIFT is shown to improve the mean effective system I/O bandwidth by 17% and the mean application I/O time by 10%, compared to multiple competing schemes. GIFT is also shown to be effective under various scenarios including high contention levels and different application characteristics.

2 Background and Motivation

HPC Storage Systems. This section describes the key components of storage systems attached to large-scale HPC systems, such as Mira, Edison, Titan, Cori, and Stampede2 [1, 22, 54, 73]. HPC systems use parallel file systems, such as Lustre, Ceph, GPFS, and PVFS, to perform parallel I/O [58–60, 79]. For simplicity, this works targets widely-used Lustre-like HPC storage system. A Lustre-like architecture consists of multiple building blocks (Fig. 1). The most basic of these is an Object Storage Target (OST), a RAID array of disks. A file is typically distributed across multiple OSTs for parallelism and can be accessed in parallel from multiple MPI processes. The OSTs serve the Object Storage Servers (OSS), which are connected to the front-end compute nodes via an I/O network. Applications running on compute nodes communicate with the OSSes via file system clients. The Meta Data Server (MDS) is the starting point for all file metadata operations. MDS consults with the Meta Data Targets (MDT), which maintain the metadata of all I/O requests.

Day in the Life of an I/O Request in a HPC System. Large-scale applications run on multiple nodes and spawn multiple (MPI) processes. These processes periodically write (or read) analysis output and checkpoint data to (or from) the storage system – referred to as an I/O phase. Processes from the same application may perform I/O on separate files or stripe a single file across multiple OSTs for concurrent access [8].

We refer to an I/O operation (read/write) accessing one OST from an MPI process of an application as an I/O request. First, the file system client on the compute node issues a remote procedure call (RPC) to the MDS, which returns information about the file stripe and OST mappings. For a new file creation request, the MDS first assigns OSTs in a capacity-balanced manner. For existing files, the MDS returns previously assigned OST information to the file system client. Then, the file system client issues an I/O request over the network to the OSS corresponding to the target OST [81]. In practice, during the I/O phase, an HPC application issues multiple I/O requests from different MPI processes.

Table 1: I/O characteristics of large-scale HPC applications.

|	< 1 min	1-15 mins	> 15 mins
I/O Phase Length	HACC [63], Chombo-Crunch [52]	HIMMER [63], Chombo-Crunch [52], WRF [48], S3D [30, 33]	PTF [32], VPIC [9], Plasma Based Accelerators [19]
	< 5 min	**5-30 mins**	**30 mins - 3hr**
I/O Interval	GTC [33], Titan Apps [39], GYRO [33]	WRF [48], S3D, Chombo-Crunch [52], Titan Apps [39]	VPIC [9], CHIMERA [33], Chombo-Crunch [52], VULCAN [33]
	< 100 GB	**100 GB - 1 TB**	**> 1 TB**
I/O Output Size	GTC [33], POP [33], GYRO [33]	WRF [48], VULCAN [33], Titan Apps [39], HACC [63]	VPIC [9], XGC1 [57], HIMMER [63], S3D [30, 33]

(a) (b) (c)
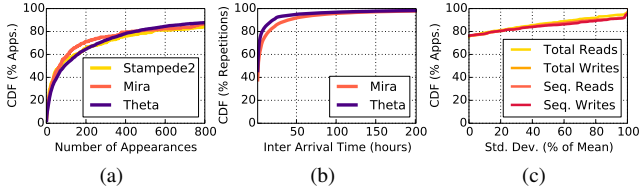
Figure 2: CDF of the (a) number of times that applications make appearances, (b) inter arrival times between each appearance, and (c) variation of I/O characteristics between two appearances.

I/O Phases of HPC Applications. HPC applications are typically long-running and perform I/O at regular intervals [28, 31, 39, 57, 90]. Their execution time ranges from a few hours to a few weeks [4, 5, 33, 57, 62, 83], and the compute period between two I/O phases can be from minutes to hours [9, 33, 39, 48, 52]. The I/O phases typically produce large amounts of data (up to hundreds of GBs) in the form of checkpoints and post-simulation results [8, 9, 33, 39, 48, 57, 62, 63]. Table 1 highlights the I/O characteristics of some popular HPC applications collected from multiple supercomputers. It shows that I/O phases can be as long as 30 min and the I/O interval (compute period) can be between 5 min and 3 h. Also, large amounts of data (100 GB - 5 TB) are transferred during each I/O phase. Next, we discuss some HPC I/O observations.

Observation 1. *HPC applications are highly repetitive in nature – that is, HPC applications typically run repeatedly and exhibit similar I/O behavior across their execution instances, though different applications have different I/O behavior.* Previous studies have shown that many HPC applications execute multiple times with similar execution characteristics [4, 5, 12, 22, 62, 63]. This is because scientific applications often model and simulate physical phenomena. This is an iterative process and requires repeated simulations for model refinement. Analysis of job scheduler logs for the last five years, two years, and one year from the leading supercomputers (Mira, Theta, and Stampede2) shows strong repetition (Fig. 2). More than 40% of the applications appear more than 200 times and about 15% of the applications appear more than 1000 times. Only less than 20% of the applications are run less than 5 times. Interestingly, we also found that the inter-arrival times between re-occurrences of HPC applications is relatively short on Mira and Theta (inter-arrival times
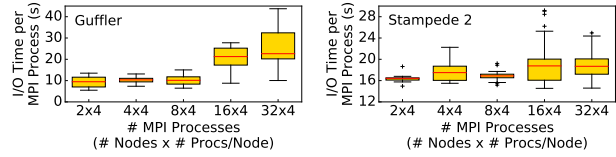
Figure 3: I/O variability among I/O performing processes of an HPC application on two HPC systems.

for Stampede2 were unavailable) (Fig. 2(b)). In fact, 80% of repetitions occur within 24 hours of each other.

Furthermore, Fig. 2(c) shows that applications exhibit only a small variation in their I/O characteristics across repetitions. This data was obtained by instrumenting HPC applications with Darshan on the Mira supercomputer [63, 83]. More than 80% of the applications that repeat more than five times show less than 5% standard deviation (as % of mean) in total amount of data read and written. We observe similar trends for different types of I/O requests (sequential and random).

Unfortunately, a shared storage back-end with no contention mitigation strategies results in severe contention among competing HPC applications [10, 28, 34, 47, 81, 85]. The I/O contention issue is further exacerbated by the need for *synchronous I/O progress* in HPC applications – an MPI process of an HPC application, exiting from an I/O phase, must wait for the slower processes to also finish their I/O [28, 31, 39, 57, 90]. Previous studies have noted that OSTs are the most contended resource on the I/O storage path (i.e., compute node, I/O routers, and OSSes) [10, 34, 47, 81, 85], since they have the lowest bandwidth among the different resources. *We note that the Meta Data Server (MDS) attempts to capacity-balance the OSTs by mapping files uniformly across OSTs, but since the MDS has no knowledge of future access patterns, its decisions cannot avoid runtime I/O contention on OSTs caused due to access patterns.* Next, we provide experimental evidence to demonstrate the impact of I/O contention and how it affects synchronous I/O progress of HPC applications.

Observation 2. *MPI processes from the same application experience significantly different I/O progress during an I/O phase – resulting in non-synchronous I/O progress across processes. This problem cannot be solved by simply identifying and speeding up a straggler process.* To demonstrate the effects of non-synchronous I/O progress, we performed a set of IOR benchmark [41] experiments on a local, production HPC system, Engaging. Engaging consists of over 100 compute nodes, and runs a production Lustre parallel file system with 44 OSTs, 44 OSSes, and 1 MDS. We ran IOR with different number of MPI processes, with each MPI process writing to a different OST. Other concurrently running applications were not controlled. We performed these experiments multiple times and from different compute nodes to eliminate transient and spatial biases. From Fig. 3, we observe that the

I/O time of different MPI processes can vary significantly (up to 4x) across runs and the number of nodes (2-32 nodes, with 4 MPI processes per node). This non-synchronous I/O progress is attributed to the difference in degrees of contention encountered by different MPI processes on their respective OSTs. Similar experiments on Stampede2 showed up to 83% variation in I/O time. Previous studies have reported similar results on non-synchronous I/O progress of MPI processes on other large-scale supercomputers including Cori, Mira, Edison, and Hopper [9, 40, 63, 83]. On further analysis, we discovered that often different processes finish at very different speeds (covering a large spectrum), and the ordering of processes in terms of their completion time changes significantly across different runs, because the I/O contention at different OSTs changes over time. *This shows that the non-synchronous I/O progress problem is not the same as the traditional straggler problem – and hence, cannot be solved by simply identifying and speeding up a straggler MPI process or OST.*

Observation 3. *Non-synchronous progress among MPI processes is caused due to unmanaged, varying I/O contention at the OSTs in the HPC storage back-end. Naïve strategies to ensure synchronous I/O progress cannot find the right balance between competing objectives: maximizing effective I/O bandwidth and fairness among applications.* To further analyze the I/O contention behavior, we ran another set of IOR experiments on Engaging, measuring the observed I/O bandwidth at each OST. Each experiment consists of writing to a particular OST from one process. Fig. 4 shows the contention (defined as the inverse of bandwidth) faced on a few OSTs (other OSTs show similar trends). Results of this simple experiment show that the degree of contention is different on each OST and varies over time. Unfortunately, allocating I/O bandwidth among competing applications to achieve conflicting objectives (fairness, effective system I/O bandwidth, synchronous I/O progress) is non-trivial. To achieve fairness, POFS (Per-OST Fair Share) scheme allocates I/O bandwidth to all competing applications equally on each individual OST (as shown in Fig. 5). But, this fair scheme may generate non-synchronous I/O progress and lead to lower effective system I/O bandwidth (i.e., sum of all bandwidths that contribute toward synchronous I/O progress). For example, under POFS, a part of the bandwidth assigned to all applications on OST3 and a part of the bandwidth assigned to A on OST1 are wasted. This is because additionally allocated bandwidths do not contribute toward synchronous I/O progress.

To ensure synchronous I/O progress, one can allocate bandwidth on each OST determined by the fair allocation on the bottlenecked OST. In Fig. 5, BSIP (Basic Synchronous I/O Progress) scheme performs such an allocation. Essentially, BSIP scheme allocates the I/O bandwidth to an application as determined by its most contended OST (e.g., A's allocations on other OSTs is determined by its bottlenecked or the most-contended OST (i.e.,OST2)). Unfortunately, this
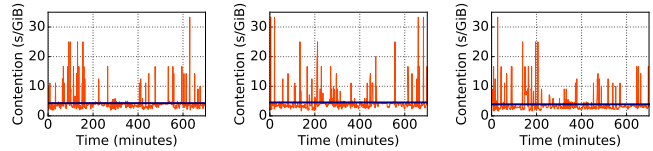
Figure 4: I/O contention on 3 of the 44 OSTs on Engaging (blue line indicates the mean contention level).
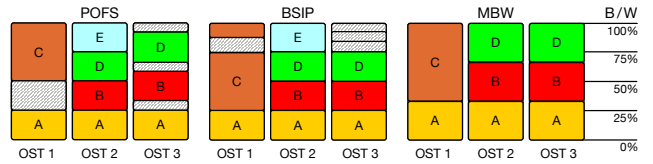
Figure 5: Bandwidth allocation among five applications spanning on three OSTs with (1) Per-OST Fair Share (POFS), (2) Basic Synchronized I/O Progress (BSIP), and (3) Minimum Bandwidth Wastage (MBW) schemes. Checkered boxes indicate bandwidth waste (not contributing toward synchronous I/O progress).

scheme also creates bandwidth gaps on less contended OSTs and lowers effective system I/O bandwidth because the bandwidth share is limited by the most-contended OST. On the other hand, a greedy approach to minimize bandwidth gaps by preferentially allocating bandwidth to applications that maximize effective system I/O bandwidth, while still ensuring synchronous I/O progress results in unfair allocations. Fig. 5 illustrates such a scheme, referred as MBW (Minimum Bandwidth Wastage), which minimize bandwidth gaps by allocating more bandwidth to certain applications and unfairly hurting other applications (e.g., it reduces the bandwidth share of application E to zero in Fig. 5). In summary, allocating I/O bandwidth among competing applications presents challenging trade-offs and GIFT strikes a balance between them as described in the next section.

3 GIFT: Design and Implementation

3.1 Overview of GIFT

First, GIFT enforces synchronous I/O progress among processes of an application by allocating bandwidth using the BSIP scheme (Fig. 5). BSIP determines the bandwidth allocation to an application according to its most contended OST. As shown in Fig. 5, BSIP scheme can create bandwidth gaps on OSTs, GIFT attempts to "fill" these bandwidth gaps by carefully throttling the bandwidth share of some applications and expanding the bandwidth share of some other applications, such that a net gain in the overall effective system I/O bandwidth is achieved. This requires identifying which applications to throttle, when to throttle, whom to expand, and how to compensate throttled applications for fairness. GIFT uses a simple and low-overhead approach to dynamically identify "throttle-friendly applications": applications which GIFT can throttle with high confidence of rewarding the stolen band-

width at a later point. The later point could be during the same I/O phase, a later I/O phase during the same run, or a future run of the same application (Sec. 3.2). GIFT issues "coupons" to throttled application which can be redeemed at later points. At regular intervals (also referred as "decision instance"), GIFT considers all throttle-friendly applications (i.e., applications which can redeem a high fraction of issued coupons - "high redemption rates") and solves a linear programming (LP) based optimization problem to maximize the effective I/O bandwidth (Sec. 3.4). This step determines which applications are throttled, which ones are expanded, and by how much. Expanded applications (which can also include throttle-friendly applications) get more than their fair share of the bandwidth, which reduces the bandwidth wastage.

Finally, GIFT bounds the unfairness toward throttle-friendly applications by using a dynamic limiting strategy (Sec. 3.2). GIFT periodically assess its fairness and compensates for the unfair treatment in the form of compute time (i.e., node-hours on the HPC system). GIFT also bounds the node-hours given out to a maximum specified "system regret budget" of compute node-hours. Algorithm 1 outlines the steps that GIFT takes at the start of every decision instance.

Algorithm 1 GIFT Decision Algorithm.

1: $X \leftarrow$ All apps performing I/O
2: $\forall i \in X$, Determine fair share of bandwidth as per $b_{i,bsip}$
3: Redeem previously issued coupons if possible (Sec. 3.3)
4: $\quad\uparrow$ Redemption rate of apps with redeemed coupons
5: Determine the set of throttle-friendly apps Y (Sec. 3.2)
6: Allocate bandwidth using LP optimization (Sec. 3.4)
7: Issue coupons to throttled apps $\subseteq Y$
8: $\quad\downarrow$ Redemption rate of apps with issued coupons

---

3.2 Identifying Throttle-friendly Applications

To identify throttle-friendly applications, GIFT throttles, issues coupons, and observes the coupon redemption rate of throttled applications. Redemption rate can be estimated with high accuracy if the whole system state (e.g., information about all concurrently running applications, their OST mapping, I/O phase length, etc.) is stored with every coupon issuance and redemption event. However, this can impose a high storage and access overhead. Also, note that, some application's OST-level I/O behavior might change over a long period (e.g., the number of OSTs, and OST mappings), causing the application's throttle-friendly status to change.

Therefore, GIFT uses the concept of receding window at the application-level that captures the recent history of an application's coupon redemption behavior (Sec. 3.5 and 4 show it is both lightweight and effective). The recent coupon redemption behavior of an application is estimated at the start of every decision instance by taking the ratio of the coupons redeemed to the last N coupons issued, where N denotes the

Table 2: GIFT model parameters.

N	Length of the receding window of applications (unit: number of coupons issued)
τ	Minimum redemption rate required for an application to be eligible for throttling and for the system to throttle applications (unit: ratio)
B_{thres}	Upper threshold of the factor by which each application's I/O request can be throttled

length of the receding window (Table 2). For fairness and simplicity, length of the receding window (N) is kept the same for all applications, although each application may take a different amount of time to accumulate N coupons depending upon its OST mappings, I/O phase length, and system I/O contention level, etc. At the start of decision instance, k, the coupon redemption rate of an application i is expressed as $c_i(k) = n_i(k)/N$, where $n_i(k)$ is the number of coupons redeemed (out of N) by application i. GIFT considers an application throttle-friendly, if its redemption rate is greater than a set threshold τ: $Y(k) = \{i \in X(k), \text{ if } c_i(k) \geq \tau\}$, where $X(k)$ is the set of all applications performing I/O and $Y(k)$ is the set of throttle-friendly applications. As the receding window moves forward, more coupons are issued only until $c_i(k) \geq \tau$. Once the redemption rate breaches the τ limit, GIFT avoids issuing more coupons to the application until it redeems its existing coupons and its redemption rate goes above τ. Using this method, GIFT ensures that unfairness is bounded for each application in case the application's redemption rate cannot go over the threshold. GIFT gives out compute node-hours as regret for unfairly treated applications periodically - this period is referred as "regret assessment period" and, as Sec. 4 shows, it can be much larger to allow applications sufficient time for redeeming the coupons.

Throttling applications based on threshold-based redemption rate at the application-level helps constrain the "regret" the system experiences from giving out node-hours (out of the system's regret budget) for unfair treatment toward one single application. But, in a system with multiple applications, the system's "cumulative" regret in terms of compute node-hours given to *all* applications can still grow sufficiently large. To address this challenge, GIFT employs a receding window at the system-level too, where it tracks the aggregate redemption rate of coupons issued by the system to *all* the applications, in order to minimize the "system regret budget" level. GIFT makes sure that the system only hands out coupons until its redemption rate is above τ (same threshold as the one used for the applications). However, unlike applications' redemption rates, GIFT resets the system's redemption rate at the end of each regret assessment period. This prevents the system's redemption rate from being saturated at τ because of non-throttle-friendly applications which never get redeemed, which can cause GIFT to miss the opportunity of throttling even throttle-friendly applications. Our evaluation (Sec. 4) shows that GIFT's approach of using τ at the system- and application- level helps keep the outstanding node-hours

("system regret budget") to a reasonably low level (e.g., less than 7% of the total gain in compute node-hours obtained via system throughput improvement due to GIFT). We also observed that keeping the same τ for applications and system is simple and effective; a higher τ at the system-level does not yield additional improvements.

Finally, we note that GIFT carefully chooses the length of receding window (N) to balance competing trade-offs: bound on unfairness toward applications vs. stability of application's status (throttle-friendly or non-throttle-friendly). If N is too large, it increases the upper bound on unfairness toward individual applications (i.e., possibility of higher number of coupons that cannot be redeemed). If N is small, an application's redemption rate $c_i(k)$ can vary erratically as the window glides, and the application's status can toggle frequently between throttle-friendly and non-throttle-friendly. GIFT achieves stable behavior by maintaining the variance of the mean redemption rate of the receding window to be small. For samples within a given receding window, the maximum variance occurs when half of the coupons can be successfully redeemed, and the other half cannot be redeemed. Hence, the maximum possible variance is $v^2 = 0.25$ (independent of N). The variance of the mean redemption rate is defined as $\sigma^2 = \frac{v^2}{N}$, which is bounded by $\sigma^2 \leq \frac{0.25}{N}$. Statistically, σ less than 0.001 can achieve reasonable stability [49]. GIFT's choice of receding window length is guided by this principle. In fact, GIFT's evaluation demonstrates that its improvements are not sensitive to the choice of parameters N (receding window size) and τ (redemption rate threshold), and that GIFT performs effectively well without the need to fine-tune.

3.3 Coupon Redemption Policy

Recall that redeeming previously issued coupons is critical to ensuring fairness. GIFT does not simply attempt to redeem an application's coupons the very next I/O phase after they were issued. This is because if redeeming a coupon requires throttling another application, then it would lead to a zero-sum result in terms of improvements in efficiency (e.g., effective system bandwidth). Thus, GIFT redeems coupons only when it does not require throttling applications. Before performing optimal bandwidth allocation and picking applications to throttle, GIFT first attempts to redeem coupons of previously throttled applications (Algorithm 1 line 3).

Coupons are redeemed when GIFT finds gaps on the OSTs on which a coupon-bearing application is running. After making the basic fair synchronous-I/O progress (BSIP) bandwidth allocation, GIFT searches through the coupon database of active applications. If all of the OSTs on which the coupon-bearing application is performing I/O have a bandwidth gap, then the coupon is redeemed either partially (if the gap is less than the coupon value) or fully or multiple coupons can also be redeemed (if the gap is large enough). By redeeming coupons in this manner, GIFT avoids throttling other appli-
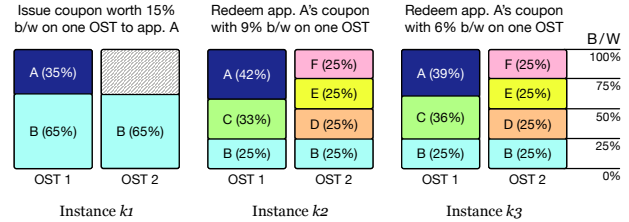
Figure 6: GIFT redeems coupons in a manner which is fair and efficient, without throttling other applications.

cations. Also, GIFT, by design, allows coupons to be issued and redeemed on different OSTs for any given application.

GIFT intelligently allocates spare bandwidth toward redeeming coupons to maintain fairness and efficiency. We note that redeeming coupons without throttling other applications requires availability of "spare I/O bandwidth". One may reason that since spare I/O bandwidth is available, applications would have naturally been allocated higher I/O bandwidth allocation, irrespective of GIFT's I/O bandwidth allocation policies. Consequently, why should GIFT refer to this additional allocated I/O bandwidth as "coupon redemption" and claim this as a mechanism to achieve fairness? Below, we discuss a simple example to illustrate the wide range of choices to allocate spare I/O bandwidth. But, GIFT carefully allocates this spare bandwidth such that (1) it redeems previously issued coupons (i.e., maintains fairness over longer term), but (2) without throttling any application at the current decision instance, otherwise it would cause more unfairness and lead to a zero-sum result in terms of efficiency.

As shown in Fig. 6, let us consider a simple example: two OSTs and bandwidth allocation decisions at three decision instances ($k1$, $k2$ and $k3$). At instance $k1$, OST1 is shared by two applications (A and B), but OST2 is only serving application B. The fair share of application A is 50% on OST1. But, if A was given its fair share on OST1, then half of the bandwidth on OST2 would be wasted since it would have not contributed toward synchronous I/O progress even if it was allocated to application B. Therefore, GIFT decides to throttle application A to reduce the overall I/O bandwidth waste. Application A's share on OST1 is reduced to 35% and a corresponding coupon is issued, and application B's share on both OSTs is increased to 65% which results in 15% reduction in I/O bandwidth waste on OST2.

At instance $k2$, OST1 is shared by three applications (A, B, and C), and OST2 is now shared by four applications (B, D, E and F). Note that application B's bandwidth share is decided by its bottlenecked OST (OST2). Application B's share on OST1 and OST2 is 25% – this ensures synchronous I/O progress and is not unfair to application B and other applications on OST1 or OST2. Due to application B's bottleneck on OST2, 9% of spare bandwidth is available on OST1. The fair share for application A and C on OST1 is 33% each. A GIFT-less approach that does not issue coupons to maintain fairness over longer time windows, would equally divide this spare

bandwidth on OST1 (9%) to both application A and C. However, GIFT decides to allocate this spare bandwidth fully to application A (increases its share to 42%, *partially* redeeming a coupon issued to application A at instance *k*1). Application C is still treated fairly even though it is not allocated any part of the spare bandwidth. Application C's fair share was 33% and it still receives it. At instance *k*3 (same OST sharing scenario as instance *k*2), application A receives 6% of the spare bandwidth (completely redeeming the coupon issued at *k*1) and the remaining 3% bandwidth can be allocated in any way (it is allocated to application C in this case).

In summary, application A was throttled in the past to increase the effective system I/O bandwidth utilization. Application A was kind then, and is later picked to receive the reward (larger share in the available spare bandwidth), without being unfair to C or throttling any other application below its fair share. This way, GIFT's decision to throttle A in the past proves to be useful. Using a throttle-and-reward approach, GIFT reduces the overall bandwidth utilization over these three time steps, while ensuring fairness to other applications and maintaining synchronous I/O progress. A GIFT-less BSIP approach (instantaneous fairness and synchronous I/O ensuring allocation at each decision instance but without throttle-and-reward approach) would have been fair but incurred 50% bandwidth waste on OST2 at instance *k*1; in comparison GIFT incurs only 35% bandwidth waste, while remaining fair over multiple decision instances. These are the kind of opportunities that GIFT detects and exploits. Such situations are not deterministic or predictable, which is why GIFT learns using the concepts of redemption rate and system regret budget.

Lastly, we note that GIFT can track coupon issuance and redemption at the user-level if the same application is being shared across multiple users and maintaining fairness at the user-level is deemed more appropriate. This will simply require including and tracking different types of identifiers per I/O request. GIFT can be extended to support different variations of "fair share" instead of being limited to treating all applications equally important. This can be achieved via encoding and tracking relative priority levels, or weights.

3.4 Optimal Bandwidth Allocation

Once a set of throttle-friendly applications is determined and coupons are redeemed, GIFT proceeds to make the bandwidth allocations to maximize the effective bandwidth. Inputs to this step include the set of throttle-friendly applications, the set of all applications concurrently performing I/O, and the set of OSTs being used by each application.

First, GIFT calculates the fair share of each application on the OSTs it is performing I/O on, to ensure synchronous I/O progress. These allocations are the same as in the BSIP scheme (Fig. 5). Next, GIFT maximizes the effective I/O bandwidth by adjusting the bandwidth of all applications subject to multiple constraints: (1) only throttle-friendly appli-

cations are allowed a lower bandwidth assignment than their fair share, (2) the total effective bandwidth is always equal to or greater than what is achieved by the BSIP scheme, and (3) *the gains from reducing the bandwidth wastage should be more than the worth of issued coupons (i.e., bandwidth waste with BSIP - bandwidth waste with GIFT > aggregate worth of coupons)*. GIFT formulates and solves this problem as a constraint-based, linear programming (LP) bandwidth allocation optimization problem, as discussed below.

Bandwidth allocation LP optimization: GIFT accounts for constraints from both, the applications' and system's perspectives. For the applications, at each decision instance *k*:

- All I/O requests (r_i) of application *i* issued across all assigned OSTs (S_i) should get the same bandwidth in order to facilitate synchronized I/O progress, i.e., for application *i*, $b_{ij} = b_i \forall j \in S_i$, where b_{ij} is bandwidth allocated to application *i*'s I/O request on OST *j* and b_i is the bandwidth allocated to application *i*'s I/O request running on the most contented OST.
- The final bandwidth allocation b_i should be s.t.

  (a) $b_{i,bsip}(1 - B_{thres}) \leq b_i \leq 1$ if $i \in Y$
  (b) $b_{i,bsip} \leq b_i \leq 1$ otherwise

The second constraint essentially allows GIFT to reduce the bandwidth share of a throttle-friendly application (belonging to set *Y*) by a configurable parameter (B_{thres}) (Table 2). Higher values of B_{thres} create more opportunity of reducing bandwidth wastage, but also result in higher coupon values. Our evaluation shows that GIFT delivers performance for a wide range of B_{thres} values and does not require tuning.

From the system's perspective, the bandwidth allocation at each OST is constrained by its full capacity. That is, $\forall j \in Z$, where *Z* is the set of all OSTs, if L_j is the set of applications served by *j*, then $\sum_{i \in L_j} b_i \leq 1$. With these constraints in mind, at every instance, *k*, we have the following polynomial-time optimization problem: maximize the effective system I/O bandwidth by making allocations b_i for each application *i*:

$$\text{maximize} \sum_{j \in Z} \sum_{i \in L_j} b_i \qquad (1)$$

We make two important remarks: (1) throttle-friendly applications are not always necessarily throttled. In fact, if it is optimal to give more bandwidth to a throttle-friendly application (i.e., expand a throttle-friendly application), given a set of contending applications, then the GIFT's LP-based optimization solution does so. (2) At any time instance, the throttling decision is not limited to picking only one candidate. In fact, the GIFT's LP-based optimization solution might select to throttle multiple throttle-friendly applications simultaneously and expand multiple applications (including throttle-friendly applications) if it leads to highest effective system I/O bandwidth while honoring the constraints.

3.5 GIFT Implementation

To evaluate GIFT, we implemented it using FUSE [67] as the base file system. Our prototype extends FUSE to capture the functionality of a parallel file system. The architecture of the GIFT implementation is similar to that of a Lustre-based HPC storage system (Sec. 2). Compute nodes mount the remote partition through FUSE. A local service daemon acts as a file system client on each compute node and monitors the mounted partition. An application's requests for file system operations are intercepted by the service daemon and executed on remote storage targets through RPC calls over the network. The file system client forwards file metadata requests to a remote metadata service (MDS), which decides the remote storage target (OST) mappings of a file. Once a file is open, data requests are directly sent over the network to the appropriate OST without involving the MDS. Each I/O request is augmented with metadata about application identity. A local service daemon (OSS) running on the storage node persists the application's data to the OST. Similar to Lustre, our implementation uses two separate network channels: "Lnet" for internal messages (for example, heartbeat, control messages, etc.) and "Dnet" for application data.

The MDS daemon broadcasts a heartbeat message to all the OSSes at a user-configurable time interval. Each OSS responds to the heartbeat message with a list of currently active data requests. The OSSes send a set of <application, I/O requests> tuples for each application they are serving. The MDS uses this data to look up its "coupon" table, make redemption decisions, and determine a set of throttle-friendly applications. Then, it makes a LP optimal bandwidth allocation decision and sends a set of <application, bandwidth allocation> tuples to each OSS. The optimal bandwidth allocation algorithm is implemented using the COIN-OR CLP [43] library. The `blkio` control group (cgroup) is used to enforce bandwidth limits. GIFT uses the MDS as a centralized coordination and decision-making service for all OSSes. OSSes incurring transient failures can be synchronized at the next decision instance. GIFT uses a 1 second timeout and makes a new decision if more than 80% of the OSSes respond. GIFT's decision instance interval is configurable and set to 10 seconds by default, that is decisions are made every 10 sec (Sec. 4). Note that GIFT operates and makes decisions at the system-level without requiring any input from the user applications or changing user applications.

We chose FUSE instead of a production parallel file system such as Lustre or GPFS to implement GIFT's core ideas because the current underlying implementation of bandwidth control support provided in Lustre and GPFS cannot be used for GIFT purposes. This is because the current bandwidth control support does not guarantee synchronous I/O progress and may create imbalance across contended OSTs – a key source of inefficiency that GIFT attempts to solve. GPFS provides bandwidth control only for maintenance tasks [24]. We

experimented with recent QoS control features of Lustre as provided by LIME and other frameworks (TBF-NRS algorithm) [56, 80, 86], but found that fairness mechanism does not work as expected because the QoS support does not account for the OST mappings. Even simple experiments such as running a few applications with equal QoS support results in significant performance differences (up to 25%) because of varying level of contention at OST level which leads to non-synchronous progress – these issues and OST mapping information is not accounted by existing early QoS support features. GIFT solves these issues.

4 Evaluation

Methodology. GIFT is evaluated on a real system using system and application characteristics of supercomputers Mira, Theta, and Stampede2. GIFT's experimental setup includes 64 OSSes (and corresponding 64 OSTs) and one MDS running on a cluster with Intel Xeon E5-2686 v4 servers – similar to the Stampede2 OSS and MDS configuration. A total of 192 file system clients are connected to OSSes. The servers and clients are connected to each other via Ethernet with a measured peak bandwidth of 4.5 GB/s. Each OST is connected to a single HDD with a peak bandwidth of 102 MB/s. Experiments are driven by an application set of 250 applications, where applications are executed with repetitions as per the typical number of distinct applications submitted on Stampede2 during a week [1, 16]. The characteristics of applications, such as number of nodes, total compute time and amount of I/O data, are taken from applications running on Stampede2 [16, 62, 66]. Number of MPI processes, and length of compute interval and I/O intervals is based on Darshan logs from Mira and Theta [83]. We use a transparent checkpointing library (DMTCP [3]) to produce periodic I/O from HPC applications such as CoMD [51], SNAP [87], and miniFE [26]. The application arrival times follow a Gamma distribution [1, 44] and are scheduled on the system using an FCFS strategy with easy-backfilling, as used by contemporary HPC schedulers [65]. For practical repeatability, the real-system evaluation scales down the compute and I/O phases to get one week's system wall clock time to finish within a few days. We also evaluate GIFT using simulations to gain deeper insights into GIFT's performance on large-scale systems. The simulations allow us to study aspects of GIFT which are too time consuming to be feasible for a representative real-system evaluation. Specifically, we use simulations to explore the effect of GIFT model parameters and high contention on GIFT performance – these explorations require hundreds of runs to cover the full parameter space. The simulations use the same parameters as the real-system evaluation, but the default application set size is increased to 500 and the simulated time period is 25 days of system wall clock time. As discussed later, the simulation results support the real-system evaluation results and demonstrate the robustness of GIFT.
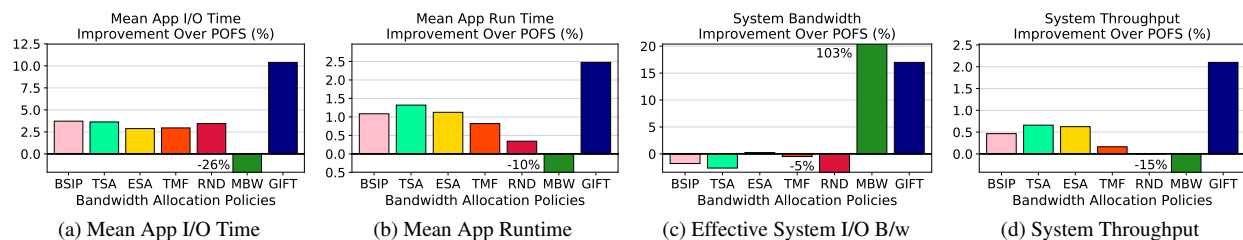
Figure 7: GIFT's implementation provides improvement for both application- and system- level objectives (higher is better).

Scheduling Policies. We evaluate GIFT against seven competing I/O scheduling policies: Per-OST Fair Share (POFS), Basic Synchronous I/O Progress (BSIP), Minimum Bandwidth Wastage (MBW), Throttle Small Applications (TSA), Expand Small Applications (ESA), Throttle Most Frequent Applications (TMF), and Throttle Randomly (RND). POFS, BSIP, and MBW are implemented as discussed in Sec. 2. TSA attempts to increase the effective system bandwidth by throttling small applications, while ESA attempts to improve the system throughput by increasing the bandwidth allocation for longer-running, smaller applications that generally do small I/O [2, 4, 5]. We also compare against other simple, intuitive strategies such as TMF and RND, which pick the "most frequently appearing" and "random" applications for bandwidth throttling, respectively. POFS is used as the baseline policy.

Objective Metrics. *Application I/O Time* is the amount of time spent in I/O by an application during its run. *Application Run Time* is the run time of the application. *Effective System Bandwidth* is the average effective I/O bandwidth during the run of an application set, defined as overall system bandwidth minus the wasted bandwidth (Sec. 2). *System Throughput* is the number of jobs completed per unit time.

GIFT's real-system implementation provides better application- and system- level performances. First, our results show that GIFT outperforms all competing techniques significantly. Fig. 7 (a)-(d) show that GIFT performs better for mean application I/O time, mean application runtime, effective system bandwidth, and system throughput, respectively. The mean application I/O time with GIFT is 10% better than with POFS, and 3.5% better than the next best technique, BSIP. Interestingly, when applications are throttled based on their characteristics (TSA, ESA, and TMF), or are arbitrarily throttled (RND), the performance remains similar to that of BSIP. This shows that naïve, rule-based techniques cannot match the performance delivered by the GIFT approach.

GIFT also improves the effective system bandwidth by more than 17% compared to POFS and other techniques, except MBW. Expectedly, MBW improves the effective system bandwidth the highest because it solely focuses on this metric. Next, we note that by compromising fairness one could design techniques that solely focus on improving system throughput (e.g., favor small jobs). GIFT does not compromise fairness,
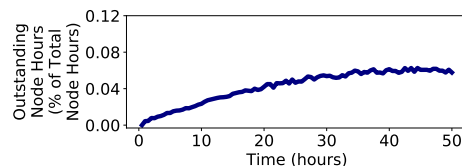
Figure 8: GIFT implementation bounds outstanding node-hours using application- and system-level redemption rate thresholds.

and it neither directly manipulates nor aims to improve the system job throughput, but by virtue of reducing I/O bandwidth waste and mean application I/O time, GIFT yields 2% improvement in system throughput. We note that even a small improvement in system throughput leads to large monetary savings in operational cost of HPC systems [18, 71, 84].

Next, we recall that GIFT gives out compute node-hours as regret, but it is minimal compared to the system throughput improvement it enables (2% savings in total compute node-hours). Fig. 8 shows that GIFT gave out less than 0.06% hours of total compute node-hours from the system regret budget in a more than two-day long experimental run – this result shows that application- and system-level redemption rate thresholds keep the system regret budget under control. Even if one were to award outstanding node-hours every day, GIFT would give out only 0.12% of node-hours, which is much smaller than the gains in system throughput (2%); this trend is also later supported by simulation results.

Next, we discuss the effectiveness of GIFT in terms of fairness. First, recall that the design of GIFT introduces two ideas: (1) opportunistically rewarding applications, and (2) compensating unfairness in I/O performance via additional compute hours. These ideas do not naturally align with the traditional notion of fairness - where a scheme tends to distribute the "benefits" equally among all applications and the "currency" of fairness measurement remains the same. In contrast, GIFT is designed to distribute the benefit opportunistically among applications because, as discussed earlier, distributing the benefits equally among all applications leads to benefit (system bandwidth) wastage due to non-synchronous I/O progress. GIFT achieves fairness by compensating I/O unfairness with compute resources. Therefore, GIFT's performance cannot be directly compared with POFS to establish its fairness effectiveness. Nevertheless, we provide this comparison for completeness and to demonstrate that GIFT is not unfair.

Fig. 9(a) and (b) show that GIFT implementation provides similar fairness in terms of both the I/O and runtime performance as the baseline fairness strategy (POFS). First, as expected, GIFT indeed provides better performance than POFS for many applications. In fact, GIFT is able to improve the I/O performance of one-third of the applications by more than 20%, while competing techniques cannot. But, this improvement is not evenly distributed among all beneficiary applications. This is because, as noted earlier, GIFT rewards certain applications opportunistically by increase their I/O bandwidth if it helps reduce the overall bandwidth waste. We note that these decisions are not systematically biased toward preferring certain applications over others.

Therefore, next, we focus on applications that receive worse performance than under the POFS scheme. This set of application provides us a better quantification of "unfairness" of GIFT and other competing schemes. First we note that other competing schemes, besides GIFT, tend to provide worse performance than POFS for a large fraction of applications compared to POFS - indicating that they are not consciously fairness aware. To further quantify this better, we use a more intuitive and traditional way to measure unfairness - the fraction of applications that achieve worse performance than POFS. As Fig. 9(c) shows GIFT outperforms other schemes in this metric as well (32% for GIFT vs. more than 45% for all other schemes, and 76% for MBW which aggressively focuses only on performance and not fairness). More importantly, even though 32% of the all the applications under GIFT achieve worse performance POFS, we calculated that the average magnitude of I/O time degradation for applications performing worse than POFS is approx. 1.2%. This shows that GIFT is able to provide a similar fair performance compared to our baseline fairness scheme (POFS). These are applications which get throttled initially but are unable to redeem coupons, for which they get compensated in node-hours. Finally, we note, unlike other competing schemes, GIFT indeeds compensates these applications via compute resources and hence, achieves fairness over the long term.

GIFT improves performance across different parameters and the required system regret budget level needed to award outstanding hours is fairly low even under pessimistic scenarios. To study the impact of model parameters on GIFT performance accurately, we perform a simulation-based exploration. First, we briefly present the simulation results for the same objective metrics as the real system evaluation. We find that GIFT's simulation results support and closely match the trends observed with the real system evaluation (Fig. 10 vs. Fig. 7). Fig. 10 shows that compared to POFS, GIFT improves the mean application I/O time by 15% and effective system bandwidth by 25%. Similarly, GIFT improves the mean application run time by more than 4% and system throughput by approx. 2%. We note thatthe absolute improvement values are higher than real system evaluation because
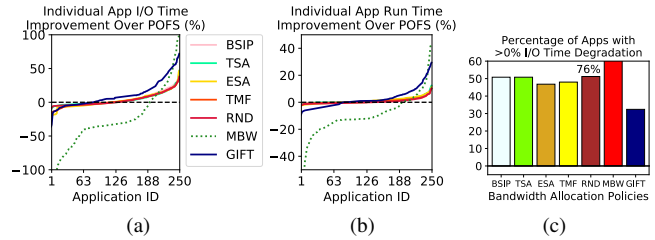
Figure 9: GIFT implementation provides I/O and runtime performance fairness to individual applications.
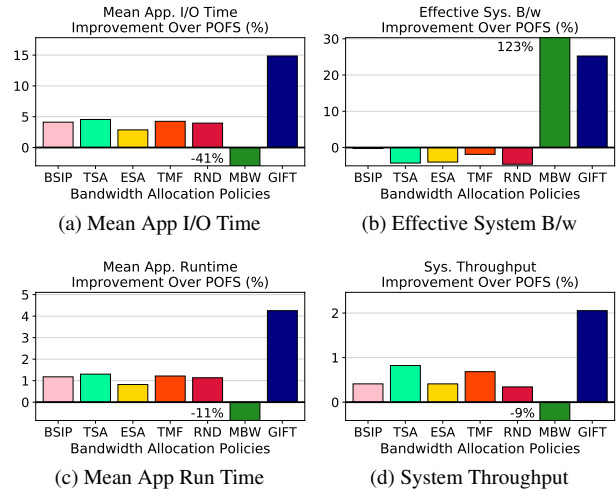
Figure 10: GIFT simulation results support GIFT real-system-based implementation results and show significant improvements.
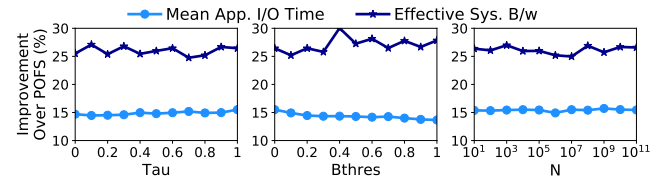
Figure 11: GIFT improves performance across different values of throttle-and-reward parameters.

simulation study covers a longer time frame (25 days) and a larger application set (500); this provides more opportunities for GIFT to make better throttle-and-reward decisions.

Next, our results (Fig. 11) illustrate that GIFT performs effectively across the parameter space and does not require tuning. Recall that τ is the minimum redemption rate for the system to throttle and for an application to be considered throttle-friendly. Therefore, it is expected that at higher values of τ, the I/O time would improve slightly. GIFT also continues to provide significant improvement in effective system bandwidth, even with high τ values. Recall that B_{thres} is the maximum factor by which an application's bandwidth can be throttled. Fig. 11(b) shows that GIFT is effective at different B_{thres} values. Note that GIFT increases the effective system bandwidth by as much as 5% points for higher B_{thres} values. This trend is expected: a higher B_{thres} value implies higher

(a) Cumulative Num. Samples   (b) Outstanding Node-Hours

Figure 12: GIFT is able to collect high cumulative number of samples and bound the node-hours awarded.

(a) Num of Appearances   (b) Num of I/O Intervals   (c) Per Interval I/O Size

Figure 13: Applications with all types of characteristics experience improvement in I/O performance with GIFT.

throttling power, and hence, better opportunities to fill the bandwidth gap. However, this also causes slight reduction in I/O time improvement (2% points). Next, Fig. 11(c) shows the impact of parameter N (the length of the receding window) on GIFT performance. Increasing N does not impact I/O time but it improves effective system bandwidth slightly due to better stability from one decision instance to the next. Overall, GIFT does better than POFS across a wide range of N values.

Studying GIFT's characteristics over time, Fig. 12(a) shows that GIFT collects a large number of samples as time progresses for both, default parameter configuration ($\tau = 0.8, B_{thres} = 0.1$) and extreme cases ($\tau = 0.1, B_{thres} = 0.1$ and $\tau = 0.8, B_{thres} = 0.8$). The sample collection continues in order to adjust to application characteristics and learn about new applications. Fig. 12(b) also shows that the number of outstanding node-hours is quite low at all times due to effectiveness of GIFT's redemption rate thresholds – therefore, indicating that only a small system regret budget is needed. Fig. 12(b) also shows that even under a pessimistic parameter selection ($\tau = 0.1, B_{thres} = 0.1$, low redemption rate threshold for applications to be considered throttle-friendly), GIFT needs a low number of outstanding node-hours at all times (less than 20 hours at any instance, although the corresponding 2% improvement in system throughput translates to a gain of more than 5,800 node-hours). Even if outstanding node-hours are awarded daily, the system regret budget needs to be only 360 node-hours over 24 days, much less than the 5,800 node-hours gained with 2% system throughput improvement.

GIFT provides performance improvement for applications of different characteristics, under high I/O contention, and device bandwidth (SSD vs. HDD). We performed simulation based exploration to understand how GIFT

(a) App I/O Time   (b) App Runtime   (c) Effective I/O B/w
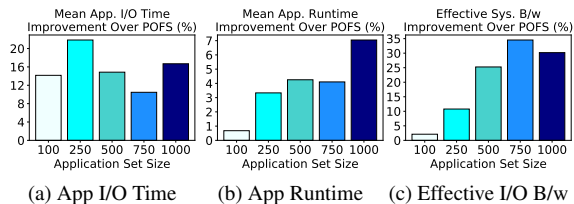
Figure 14: GIFT performs better than POFS at all contention levels.

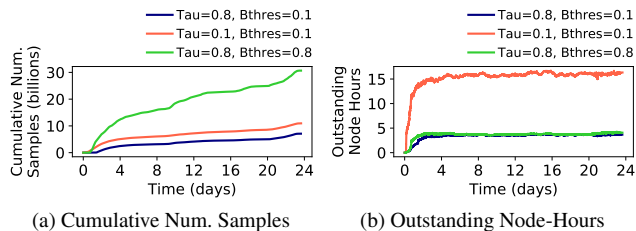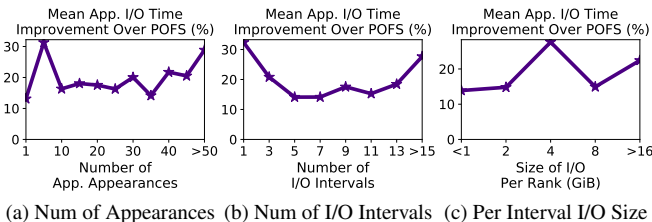performs when key application characteristics are varied: number of appearances of an application, number of I/O intervals, and the size of I/O per I/O interval per MPI process (rank). We found (Fig. 13 (a)-(c)) that GIFT continues to provide a significant improvement in application I/O performance as we vary the number of appearances of an application, number of I/O intervals, and the per-interval I/O size across a wide range. Our results (Fig. 14) also show that GIFT's performance benefits actually improve as we increase the contention level from 100-applications set to 1,000-applications set; this is expected because a higher-level of contention increases the chances for GIFT to exercise throttle and reward. For 1,000 application-set GIFT improves mean application I/O time by up to 16%, mean application run time by up to 7%, and effective system bandwidth by up to 30%. Finally, although GIFT does not rely on specific storage device characteristics to provide benefits, we studied the effect of device bandwidth (e.g., SSD vs. HDD) on the limits of GIFT performance. As expected, we did not find the GIFT performance improvements to be sensitive to the underlying storage device.

GIFT implementation is low-overhead and scalable on a real system. GIFT has two sources of overhead: computation and communication. MDS incurs the computation overhead due to solving an LP optimization problem. Communication overhead is incurred due to message exchanges between the OSTs and MDS. To obtain pessimistic estimates on the GIFT implementation overhead on a real system, we increase the number of OSTs from 32 to 200 and increase the application set size to 1,000 – amplifying the degree of GIFT overheads. We measured that the CPU overhead on the MDS increased from 1 ms to 5 ms which is negligible compared to decision instance interval (10 seconds); GIFT produces similar results with similar decision instance interval lengths, however choosing too small interval (e.g., 1 second) can make overhead effects visible and choosing very large interval (e.g., 10 minute) can lead missed opportunities for throttle-and-reward. The volume of messages between the MDS and the OSTs is also minimal (less than 4 MB over two days) and occurs on a non-critical network path. In our real system experiments, we measured that overall GIFT's implementation imposes a negligible overhead on I/O performance even under pessimistic scenarios (less than 0.01%).

5 Discussion

Relationship between I/O bandwidth improvements and system throughput. We note that GIFT does not actively manipulate the I/O bandwidth allocation to directly improve the system throughput. It is trivial to improve the system throughput - for example, by allocating more I/O bandwidth share to short-running jobs which can significantly increase the system throughput at the expense of fairness. Nevertheless, as our results show, GIFT is able to improve the overall system throughput. This is because GIFT eliminatea I/O bandwidth inefficiencies by increasing the I/O bandwidth toward synchronous progress which reduces the overall I/O time and run time of applications. Reducing the overall run time of applications by judiciously utilizing the available I/O bandwidth, in turn, leads to completion of more jobs per unit time (i.e., system throughput increases).

Why traditional notions of measuring fairness alone may be not be adequate for assessing the effectiveness of GIFT. A conventional notion of fairness measures the amount of equal opportunity among all participants. In the case of GIFT, this translates to providing equal bandwidth to all jobs concurrently performing I/O on the same OST (i.e., POFS). However, this does not lead to effective equal bandwidth division since jobs may not be able to leverage the full I/O bandwidth due to non-synchronous I/O progress. While, GIFT does not enforce this fair opportunity at every decision instance, it does enforce it as a constraint in the long run. Thus, GIFT enforces fair opportunity as a constraint.

Another conventional notion of fairness measures the amount of equal performance among all participants. For example, calculating the difference between maximum and minimum performances, or the standard deviation of performances, or Jain's Fairness Index [27]. Fairness can be viewed at as all jobs having equal I/O performance. In practice, this is difficult to enforce and impractical to achieve in a diverse and dynamic I/O environment of an HPC storage system. Job I/O performance depends on a variety of job-specific aspects which are not in control of GIFT (GIFT only performs time-divided bandwidth allocation) such as number of OSTs across which a file is stripped, size of I/O, type and pattern of I/O, I/O interface (POSIX, MPIIO, STDIO), etc. Thus, while GIFT enforces equal opportunity in terms of bandwidth (resource) allocation as hard constraint, it cannot enforce overall equal I/O performance.

In the case of GIFT, one could argue that fairness can be defined as all applications having equal improvement as compared to POFS. However, this definition is not meaningful since POFS already performs instantaneous fair allocation, thus, I/O performance with POFS is fair and attempting to achieve "fair improvement from a fair performance" does not have practical value for end users. Therefore, as discussed in Sec. 4, GIFT's fairness is better quantified by focusing on the applications which achieves worse performance than POFS. If the improvement over POFS is positive, then GIFT is considered fair for such beneficiary applications, but the improvement over POFS among such beneficiary applications is not equal. This is because GIFT rewards certain applications opportunistically by increasing their I/O bandwidth if it helps reduce the overall bandwidth waste. Finally, GIFT compensates unfairness in one type of resource allocation by allocating another type of resource - this feature makes GIFT fairness fundamentally different than traditional notions.

6 Related Work

Many prior works have focused on identifying the root causes of contention and characterizing the I/O bottlenecks [2, 10, 12, 22, 28, 31, 34, 38, 39, 46, 47, 63, 68, 81, 82, 85]. These works do not propose mitigation techniques. Studies focusing on application-level techniques [13, 35, 42, 56, 61, 89, 91, 92], such as CALCioM [14], rely on application modifications and cooperation for coordinating I/O transactions among applications. Client-side solutions, which coordinate I/O requests to and from the client-attached burst buffers or requests handlers [6, 25, 29, 36, 37, 76–78], end up underutilizing the back-end bandwidth due to the lack of a storage-system view. In general, client-side techniques are complementary to GIFT and can be used to further enhance application performance. On the other hand, server-side solutions aim to efficiently schedule the I/O requests from the server nodes to the disk targets [17, 20, 21, 50, 64, 69, 70, 72, 74, 90]. For example, IOrchestrator [88] uses spacial locality of I/O requests to unfairly prioritize the most disk efficient requests. Note that none of these studies consider the distributed and synchronous I/O behavior of HPC applications. This paper introduced, GIFT, a new I/O bandwidth allocation approach to ensure synchronous I/O progress for HPC application while maximizing I/O throughput and ensuring fairness.

7 Conclusion

Improving effective system I/O bandwidth, providing fairness among applications, and ensuring synchronous I/O progress are three major challenges in parallel storage systems, but no existing approaches have considered them as a joint problem. GIFT identifies and solves this new problem using a throttle-and-reward approach - yielding significant improvements (17% in mean effective system I/O bandwidth and 10% in the mean application I/O time). GIFT is available at `https://github.com/GoodwillComputingLab/GIFT`.

References

[1] *Stampede2 User Guide*, 2018 (accessed January 10, 2019). `https://portal.tacc.utexas.edu/user-guides/stampede2`.

[2] Gonzalo Pedro Rodrigo Alvarez, Per-Olov Östberg, Erik Elmroth, Katie Antypas, Richard Gerber, and Lavanya Ramakrishnan. Towards Understanding Job Heterogeneity in HPC: A NERSC Case Study. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 521–526. IEEE, 2016.

[3] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *Parallel and Distributed Processing Symposium (IPDPS), 2009 IEEE International*, pages 1–12. IEEE, 2009.

[4] Katie Antypas, BA Austin, TL Butler, RA Gerber, Cary Whitney, Nick Wright, Woo-Sun Yang, and Zhengji Zhao. NERSC Workload Analysis on Hopper. Technical report, Technical report, LBNL Report, 2013.

[5] Brian Austin, Tina Butler, Richard Gerber, Cary Whitney, Nicholas Wright, Woo-Sun Yang, and Zhengji Zhao. Hopper Workload Analysis. 2014.

[6] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Ruth Aydt, Quincey Koziol, Marc Snir, et al. Taming Parallel I/O Complexity with Auto-Tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 68. ACM, 2013.

[7] John Bent, Sorin Faibish, Jim Ahrens, Gary Grider, John Patchett, Percy Tzelnic, and Jon Woodring. Jitter-Free Co-Processing on a Prototype Exascale Storage Stack. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, 2012.

[8] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 21. ACM, 2009.

[9] Suren Byna, A Uselton, D Knaak Prabhat, and Y He. Trillion Particles, 120,000 cores, and 350 TBs: Lessons Learned from a Hero I/O Run on Hopper. In *Cray user group meeting*, 2013.

[10] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. On the Performance Variation in Modern Storage Stacks. In *FAST*, pages 329–344, 2017.

[11] Franck Cappello. Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *IJHPCA*, 23(3):212–226, 2009.

[12] Christopher S Daley, Devarshi Ghoshal, Glenn K Lockwood, Sudip Dosanjh, Lavanya Ramakrishnan, and Nicholas J Wright. Performance Characterization of Scientific Workflows for the Optimal use of Burst Buffers. *Future Generation Computer Systems*, 2017.

[13] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, and Leigh Orf. Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-Free I/O. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 155–163. IEEE, 2012.

[14] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O Interference in HPC Systems Through Cross-Application Coordination. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 155–164. IEEE, 2014.

[15] Elmootazbellah N Elnozahy and James S Plank. Checkpointing for Peta-scale Systems: A Look into the Future of Practical Rollback-Recovery. *TDSC 2004*, 1(2):97–108, 2004.

[16] Thomas Furlani. XDMoD Value Analytics. 2018.

[17] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the I/O of HPC Applications under Congestion. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1013–1022. IEEE, 2015.

[18] Richard Gerber, James Hack, Katherine Riley, Katie Antypas, Richard Coffey, Eli Dart, Tjerk Straatsma, Jack Wells, Deborah Bard, Sudip Dosanjh, et al. Crosscut Report: Exascale Requirements Reviews, March 9–10, 2017–Tysons Corner, Virginia. An Office of Science Review Sponsored by: Advanced Scientific Computing Research, Basic Energy Sciences, Biological and Environmental Research, Fusion Energy Sciences, High Energy Physics, Nuclear Physics. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States); Argonne . . . , 2018.

[19] Richard A Gerber and Harvey Wasserman. Large Scale Computing and Storage Requirements for High Energy Physics. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 20102.

[20] Ajay Gulati, Arif Merchant, and Peter J Varman. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *ACM SIGMETRICS*

Performance Evaluation Review, volume 35, pages 13–24. ACM, 2007.

[21] Ajay Gulati, Arif Merchant, and Peter J Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 437–450. USENIX Association, 2010.

[22] Raghul Gunasekaran, Sarp Oral, Jason Hill, Ross Miller, Feiyi Wang, and Dustin Leverman. Comparative I/O Workload Characterization of Two Leadership Class Storage Clusters. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 31–36. ACM, 2015.

[23] Jun He, Duy Nguyen, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Reducing File System Tail Latencies with Chopper. In *FAST*, volume 15, pages 119–133, 2015.

[24] John Hearns, Marc A Kaplan, and Egonle Bo. Limit / fair share of gpfs bandwidth, Jan 2018.

[25] Stephen Herbein, Dong H Ahn, Don Lipari, Thomas RW Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. Scalable I/O-aware Job Scheduling for Burst Buffer Enabled HPC Clusters. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 69–80. ACM, 2016.

[26] M Heroux and S Hammond. MiniFE: Finite Element Solver.

[27] Raj Jain, Arjan Durresi, and Gojko Babic. Throughput Fairness Index: An Explanation. In *ATM Forum contribution*, volume 99, 1999.

[28] Ye Jin, Xiaosong Ma, Mingliang Liu, Qing Liu, Jeremy Logan, Norbert Podhorszki, Jong Youl Choi, and Scott Klasky. Combining Phase Identification and Statistic Modeling for Automated Parallel Benchmark Generation. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):309–320, 2015.

[29] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance Differentiation for Storage Systems using Adaptive Control. *ACM Transactions on Storage (TOS)*, 1(4):457–480, 2005.

[30] Seong Jo Kim. Parallel I/O Profiling and Optimization in HPC Systems. 2014.

[31] Youngjae Kim and Raghul Gunasekaran. Understanding I/O Workload Characteristics of a Peta-scale Storage System. *The Journal of Supercomputing*, 71(3):761–780, 2015.

[32] Michelle Koo, Wucherl Yoo, and Alex Sim. I/O Performance Analysis Framework on Measurement Data from Scientific Clusters. 2015.

[33] Douglas Kothe and Ricky Kendall. Computational Science Requirements for Leadership Computing. *Oak Ridge National Laboratory, Technical Report*, 2007.

[34] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O Performance Challenges at Leadership Scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 40. ACM, 2009.

[35] Han Deok Lee, Young Jin Nam, Kyong Jo Jung, Seok Gan Jung, and Chanik Park. Regulating I/O Performance of Shared Storage with a Control Theoretical Approach. In *MSST*, pages 105–117, 2004.

[36] Yan Li, Xiaoyuan Lu, Ethan L Miller, and Darrell DE Long. Ascar: Automating Contention Management for High-Performance Storage Systems. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pages 1–16. IEEE, 2015.

[37] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the Role of Burst Buffers in Leadership-Class Storage Systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[38] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces. In *FAST*, volume 14, pages 213–228, 2014.

[39] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. Server-Side Log Data Analytics for I/O Workload Characterization and Coordination on Large Shared Storage Systems. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 819–829. IEEE, 2016.

[40] Glenn K Lockwood, Wucherl Yoo, Suren Byna, Nicholas J Wright, Shane Snyder, Kevin Harms, Zachary Nault, and Philip Carns. UMAMI: A Recipe for Generating Meaningful Metrics Through Holistic I/O Performance Analysis. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pages 55–60. ACM, 2017.

[41] William Loewe, T McLarty, and C Morrone. IOR Benchmark, 2012.

[42] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. Managing Variability in the IO Performance of Petascale Storage Systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–12. IEEE, 2010.

[43] Robin Lougee-Heimer. The Common Optimization INterface for Operations Research: Promoting Opensource Software in the Operations Research Community. *IBM Journal of Research and Development*, 47(1):57–66, 2003.

[44] Uri Lublin and Dror G Feitelson. The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, 2003.

[45] Robert Lucas. Top Ten Exascale Research Challenges. In *DOE ASCAC Subcommittee Report*, 2014.

[46] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. A Multiplatform Study of I/O Behavior on Petascale Supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 33–44. ACM, 2015.

[47] Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert Latham, Robert Ross, Shane Snyder, and Stefan M Wild. Analysis and Correlation of Application I/O Performance and System-Wide I/O Activity. In *Networking, Architecture, and Storage (NAS), 2017 International Conference on*, pages 1–10. IEEE, 2017.

[48] George S Markomanolis, Bilel Hadri, Rooh Khurram, and Saber Feki. Scientific Applications Performance Evaluation on Burst Buffer. In *International Conference on High Performance Computing*, pages 701–711. Springer, 2017.

[49] Deirdre N McCloskey, Stephen T Ziliak, et al. The Standard Error of Regressions. *Journal of economic literature*, 34(1):97–114, 1996.

[50] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: Quality-of-Service in Large Disk Arrays. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 245–254. ACM, 2011.

[51] Jamaludin Mohd-Yusof, S Swaminarayan, and TC Germann. Co-Design for Molecular Dynamics: An Exascale Proxy Application, 2013.

[52] Andrey Ovsyannikov, Melissa Romanus, Brian Van Straalen, Gunther H Weber, and David Trebotich. Scientific Workflows at Satawarp-Apeed: Accelerated Data-Intensive Science using NERSC's Burst Buffer. In *Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS), 2016 1st Joint International Workshop on*, pages 1–6. IEEE, 2016.

[53] Tirthak Patel, Suren Byna, Glenn K Lockwood, and Devesh Tiwari. Revisiting I/O Behavior in Large-Scale Storage Systems: The Expected and the Unexpected. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2019.

[54] Torben Kling Petersen. HPC Storage Current Status and Futures.

[55] Yingjin Qian, Xi Li, Shuichi Ihara, Andreas Dilger, Carlos Thomaz, Shilong Wang, Wen Cheng, Chunyan Li, Lingfang Zeng, Fang Wang, et al. LPCC: Hierarchical Persistent Client Caching for Lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 88. ACM, 2019.

[56] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. A Configurable Rule Based Classful Token Bucket Filter Network Request Scheduler for the Lustre File System. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 6. ACM, 2017.

[57] Robert Ross, Robert Ross, Gary Grider, Gary Grider, Evan Felix, Evan Felix, Mark Gary, Mark Gary, Scott Klasky, Scott Klasky, et al. Storage Systems and Input/Output to Support Extreme Scale Science. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015.

[58] Robert B Ross, Rajeev Thakur, et al. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*, pages 391–430, 2000.

[59] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[60] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[61] David Shue, Michael J Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *OSDI*, volume 12, pages 349–362. USENIX, 2012.

[62] Nikolay A Simakov, Joseph P White, Robert L DeLeon, Steven M Gallo, Matthew D Jones, Jeffrey T Palmer, Benjamin Plessinger, and Thomas R Furlani. A Workload Analysis of NSF's Innovative HPC Resources Using XDMoD. *arXiv preprint arXiv:1801.04306*, 2018.

[63] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright. Modular HPC I/O Characterization with Darshan. In *Extreme-Scale Programming Tools (ESPT), Workshop on*, pages 9–17. IEEE, 2016.

[64] Huaiming Song, Yanlong Yin, Xian-He Sun, Rajeev Thakur, and Samuel Lang. Server-Side I/O Coordination for Parallel File Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 17. ACM, 2011.

[65] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P Sadayappan. Characterization of Backfilling Strategies for Parallel Job Scheduling. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pages 514–519. IEEE, 2002.

[66] Dan Stanzione, Bill Barth, Niall Gaffney, Kelly Gaither, Chris Hempel, Tommy Minyard, S Mehringer, Eric Wernert, H Tufo, D Panda, et al. Stampede 2: The Evolution of an XSEDE Supercomputer. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, page 15. ACM, 2017.

[67] Miklos Szeredi. FUSE: Filesystem in Userspace. https://fuse.sourceforge.net/, 2005. Online (accessed January 10, 2019).

[68] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking File System Benchmarking: It* IS* Rocket Science. In *HotOS*, volume 13, pages 1–5, 2011.

[69] Sagar Thapaliya, Purushotham Bangalore, Jay Lofstead, Kathrn Mohror, and Adam Moody. IO-Cop: Managing Concurrent Accesses to Shared Parallel File System. In *Parallel Processing Workshops (ICCPW), 2014 43rd International Conference on*, pages 52–60. IEEE, 2014.

[70] 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.

[71] Edward Walker. The Real Cost of a CPU Hour. *Computer*, (4):35–41, 2009.

[72] Chien-Min Wang, Tse-Chen Yeh, and Guo-Fu Tseng. Provision of Storage QoS in Distributed File Systems for Clouds. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 189–198. IEEE, 2012.

[73] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S Vazhkudai. Improving Large-scale Storage System Performance via Topology-Aware and Balanced Data Placement. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 656–663. IEEE, 2014.

[74] Hui Wang and Peter J Varman. Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-Aware Allocation. In *FAST*, volume 14, pages 229–242, 2014.

[75] Jingjing Wang, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Controlled Contention: Balancing Contention and Reservation in Multicore Application Scheduling. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 946–955. IEEE, 2015.

[76] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An Ephemeral Burst-buffer File System For Scientific Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 69. IEEE Press, 2016.

[77] Teng Wang, Sarp Oral, Michael Pritchard, Bin Wang, and Weikuan Yu. Trio: Burst Buffer Based I/O Orchestration. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 194–203. IEEE, 2015.

[78] Teng Wang, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, and Weikuan Yu. Burstmem: A High-Performance Burst Buffer System for Scientific Applications. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 71–79. IEEE, 2014.

[79] 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. USENIX Association, 2006.

[80] Li Xi and Zeng Lingfang. LIME: A Framework for Lustre Global QoS Management. *Lustre Administrator and Developer Workshop*, 2018.

[81] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing Output Bottlenecks in a Supercomputer. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[82] Bing Xie, Yezhou Huang, Jeffrey S Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. Predicting Output Performance of a Petascale Supercomputer. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 181–192. ACM, 2017.

[83] Cong Xu, Shane Snyder, Vishwanath Venkatesan, Philip Carns, Omkar Kulkarni, Suren Byna, Roberto Sisneros, and Kalyana Chadalavada. DXT: Darshan eXtended Tracing. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[84] Fan Yang and Andrew A Chien. Extreme Scaling of Supercomputing with Stranded Power: Costs and Capabilities. *arXiv preprint arXiv:1607.02133*, 2016.

[85] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Rob Ross, and Gabriel Antoniu. On the Root Causes of Cross-application I/O Interference in HPC Storage Systems. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 750–759. IEEE, 2016.

[86] L Zeng, J Kaiser, A Brinkmann, T Süß, L Xi, Q Yingjin, and S Ihara. Providing QoS-Mechanisms for Lustre through Centralized Control Applying the TBF-NRS. *Lustre User Group*, 2017.

[87] Joe Zerr and Randal Baker. SNAP. `http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/snap/`, 2018 (accessed January 10, 2019).

[88] Xuechen Zhang, Kei Davis, and Song Jiang. IOrchestrator: Improving the Performance of Multi-node I/O Systems via Inter-server Coordination. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[89] Xuechen Zhang, Kei Davis, and Song Jiang. Opportunistic Data-driven Execution of Parallel Programs for Efficient I/O Services. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 330–341. IEEE, 2012.

[90] Zhou Zhou, Xu Yang, Dongfang Zhao, Paul Rich, Wei Tang, Jia Wang, and Zhiling Lan. I/O-Aware Batch Scheduling for Petascale Computing Systems. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 254–263. IEEE, 2015.

[91] Timothy Zhu, Michael A Kozuch, and Mor Harchol-Balter. WorkloadCompactor: Reducing Datacenter Cost while Providing Tail Latency SLO Guarantees. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 598–610. ACM, 2017.

[92] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

Scalable Parallel Flash Firmware for Many-core Architectures

Jie Zhang[1], Miryeong Kwon[1], Michael Swift[2], Myoungsoo Jung[1]
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)[1], University of Wisconsin at Madison[2]
http://camelab.org

Abstract

NVMe is designed to unshackle flash from a traditional storage bus by allowing hosts to employ many threads to achieve higher bandwidth. While NVMe enables users to fully exploit all levels of parallelism offered by modern SSDs, current firmware designs are not scalable and have difficulty in handling a large number of I/O requests in parallel due to its limited computation power and many hardware contentions.

We propose `DeepFlash`, a novel manycore-based storage platform that can process more than a million I/O requests in a second (1MIOPS) while hiding long latencies imposed by its internal flash media. Inspired by a parallel data analysis system, we design the firmware based on many-to-many threading model that can be scaled horizontally. The proposed `DeepFlash` can extract the maximum performance of the underlying flash memory complex by concurrently executing multiple firmware components across many cores within the device. To show its extreme parallel scalability, we implement `DeepFlash` on a many-core prototype processor that employs dozens of lightweight cores, analyze new challenges from parallel I/O processing and address the challenges by applying concurrency-aware optimizations. Our comprehensive evaluation reveals that `DeepFlash` can serve around 4.5 GB/s, while minimizing the CPU demand on microbenchmarks and real server workloads.

1   Introduction

Solid State Disks (SSDs) are extensively used as caches, databases, and boot drives in diverse computing domains [37, 42, 47, 60, 74]. The organizations of modern SSDs and flash packages therein have undergone significant technology shifts [11, 32, 39, 56, 72]. In the meantime, new storage interfaces have been proposed to reduce overheads of the host storage stack thereby improving the storage-level bandwidth. Specifically, NVM Express (NVMe) is designed to unshackle flash from a traditional storage interface and enable users to take full advantages of all levels of SSD internal parallelism [13, 14, 54, 71]. For example, it provides streamlined commands and up to 64K deep queues, each with up to 64K entries. There is massive parallelism in the backend where

requests are sent to tens or hundreds of flash packages. This enables assigning queues to different applications; multiple deep NVMe queues allow the host to employ many threads thereby maximizing the storage utilization.

An SSD should handle many concurrent requests with its massive internal parallelism [12, 31, 33, 34, 61]. However, it is difficult for a single storage device to manage the tremendous number of I/O requests arriving in parallel over many NVMe queues. Since highly parallel I/O services require simultaneously performing many SSD internal tasks, such as address translation, multi-queue processing, and flash scheduling, the SSD needs multiple cores and parallel implementation for a higher throughput. In addition, as the tasks inside the SSD increase, the SSD must address several scalability challenges brought by garbage collection, memory/storage contention and data consistency management when processing I/O requests in parallel. These new challenges can introduce high computation loads, making it hard to satisfy the performance demands of diverse data-centric systems. Thus, the high-performance SSDs require not only a powerful CPU and controller but also an efficient flash firmware.

We propose `DeepFlash`, a manycore-based NVMe SSD platform that can process more than one million I/O requests within a second (1MIOPS) while minimizing the requirements of internal resources. To this end, we design a new flash firmware model, which can extract the maximum performance of hundreds of flash packages by concurrently executing firmware components atop a manycore processor. The layered flash firmware in many SSD technologies handles the internal datapath from PCIe to physical flash interfaces as a single heavy task [66, 76]. In contrast, `DeepFlash` employs a many-to-many threading model, which multiplexes any number of threads onto any number of cores in firmware.

Specifically, we analyze key functions of the layered flash firmware and decompose them into multiple modules, each is scaled independently to run across many cores. Based on the analysis, this work classifies the modules into a queue-gather stage, a trans-apply stage, and a flash-scatter stage, inspired by a parallel data analysis system [67]. Multiple threads on the queue-gather stage handle NVMe queues, while each thread on the flash-scatter stage handles many flash devices on a channel bus. The address translation between logical

block addresses and physical page numbers is simultaneously performed by many threads at the trans-apply stage. As each stage can have different numbers of threads, contention between the threads for shared hardware resources and structures, such as mapping table, metadata and memory management structures can arise. Integrating many cores in the scalable flash firmware design also introduces data consistency, coherence and hazard issues. We analyze new challenges arising from concurrency, and address them by applying concurrency-aware optimization techniques to each stage, such as parallel queue processing, cache bypassing and background work for time-consuming SSD internal tasks.

We evaluate a real system with our hardware platform that implements `DeepFlash` and internally emulates low-level flash media in a timing accurate manner. Our evaluation results show that `DeepFlash` successfully provides more than 1MIOPS with a dozen of simple low-power cores for all reads and writes with sequential and random access patterns. In addition, `DeepFlash` reaches 4.5 GB/s (above 1MIOPS), on average, under the execution of diverse real server workloads. The main contributions of this work are summarized as below:

• **Many-to-many threading firmware.** We identify scalability and parallelism opportunities for high-performance flash firmware. Our many-to-many threading model allows future manycore-based SSDs to dynamically shift their computing power based on different workload demands without any hardware modification. `DeepFlash` splits all functions from the existing layered firmware architecture into three stages, each with one or more thread groups. Different thread groups can communicate with each other over an on-chip interconnection network within the target SSD.

• **Parallel NVMe queue management.** While employing many NVMe queues allows the SSD to handle many I/O requests through PCIe communication, it is hard to coordinate simultaneous queue accesses from many cores. `DeepFlash` dynamically allocates the cores to process NVMe queues rather than statically assigning one core per queue. Thus, a single queue is serviced by multiple cores, and a single core can service multiple queues, which can deliver full bandwidth for both balanced and unbalanced NVMe I/O workloads. We show that this parallel NVMe queue processing exceeds the performance of the static core-per-queue allocation by 6x, on average, when only a few queues are in use. `DeepFlash` also balances core utilization over computing resources.

• **Efficient I/O processing.** We increase the parallel scalability of many-to-many threading model by employing non-blocking communication mechanisms. We also apply simple but effective lock and address randomization methods, which can distribute incoming I/O requests across multiple address translators and flash packages. The proposed method minimizes the number of hardware core to achieve 1MIOPS. Putting all it together, `DeepFlash` improves bandwidth by 3.4× while significantly reducing CPU requirements, compared to conventional firmware. Our `DeepFlash` requires only
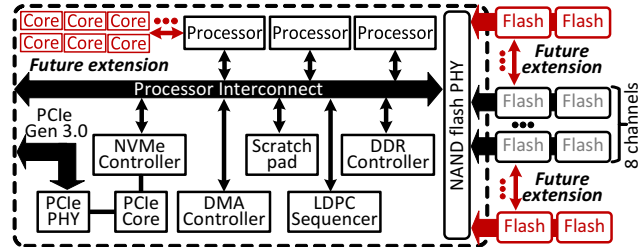
Figure 1: Overall architecture of an NVMe SSD.

a dozen of lightweight in-order cores to deliver 1MIOPS.

2 Background

2.1 High Performance NVMe SSDs

Baseline. Figure 1 shows an overview of a high-performance SSD architecture that Marvell recently published [43]. The host connects to the underlying SSD through four Gen 3.0 PCIe lanes (4 GB/s) and a PCIe controller. The SSD architecture employs three embedded processors, each employing two cores [27], which are connected to an internal DRAM controller via a processor interconnect. The SSD employs several special-purpose processing elements, including a low-density parity-check (LDPC) sequencer, data transfer (DMA) engine, and scratch-pad memory for metadata management. All these multi-core processors, controllers, and components are connected to a flash complex that connects to eight channels, each connecting to eight packages, via flash physical layer (PHY). We select this multicore architecture description as our reference and extend it, since it is only documented NVMe storage architecture that employs multiple cores at this juncture, but other commercially available SSDs also employ a similar multi-core firmware controller [38, 50, 59].

Future architecture. The performance offered by these devices is by far below 1MIOPS. For higher bandwidth, a future device can extend storage and processor complexes with more flash packages and cores, respectively, which are highlighted by red in the figure. The bandwidth of each flash package is in practice tens of MB/s, and thus, it requires employing more flashes/channels, thereby increasing I/O parallelism. This flash-side extension raises several architectural issues. First, the firmware will make frequent SSD-internal memory accesses that stress the processor complex. Even though the PCIe core, channel and other memory control logic may be implemented, metadata information increases for the extension, and its access frequency gets higher to achieve 1MIOPS. In addition, DRAM accesses for I/O buffering can be a critical bottleneck to hide flash's long latency. Simply making cores faster may not be sufficient because the processors will suffer from frequent stalls due to less locality and contention at memory. This, in turn, makes each core bandwidth lower, which should be addressed with higher parallelism on computation parts. We will explain the current architecture and show why it is non-scalable in Section 3.
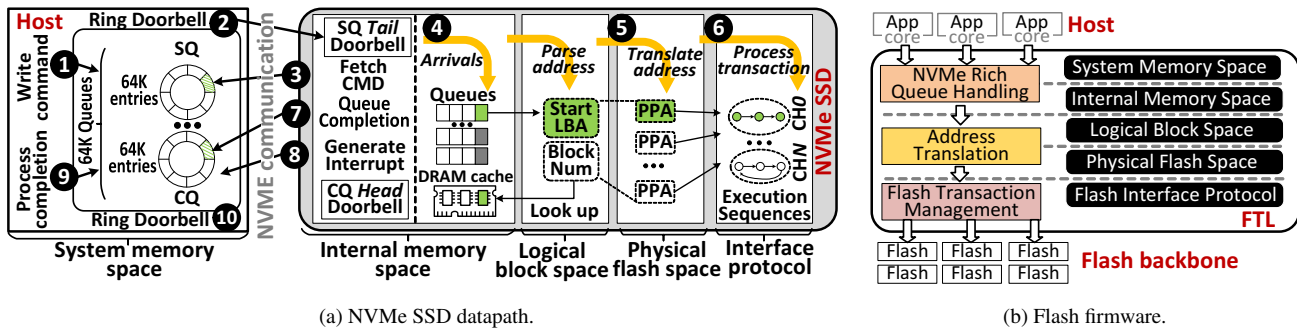
(a) NVMe SSD datapath.   (b) Flash firmware.

Figure 2: Datapath from PCIe to Flash and overview of flash firmware.

Datapath from PCIe to flash. To understand the source of scalability problems, it requires being aware of the internal datapath of NVMe SSDs and details of the datapath management. Figure 2a illustrates the internal datapath between PCIe and NV-DDR [7, 53], which is managed by NVMe [16] and ONFi [69] protocols, respectively. NVMe employs multiple device-side *doorbell* registers, which are designed to minimize handshaking overheads. Thus, to issue an I/O request, applications submit an NVMe command to a *submission queue (SQ)* (❶) and notify the SSD of the request arrival by writing to the doorbell corresponding to the queue (❷). After fetching a host request from the queue (❸), flash firmware, known as *flash translation layer* (FTL), parses the I/O operation, metadata, and data location of the target command (❹). The FTL then translates the *physical page address (PPA)* from the host's *logical block address (LBA)* (❺). In the meantime, the FTL also orchestrates data transfers. Once the address translation is completed, the FTL moves the data, based on the I/O timing constraints defined by ONFi (❻). A *completion queue (CQ)* is always paired with an SQ in the NVMe protocol, and the FTL writes a result to the CQ and updates the tail doorbell corresponding to the host request. The FTL notifies the queue completion to the host (❼) by generating a message-signaled interrupt (MSI) (❽). The host can finish the I/O process (❾) and acknowledge the MSI by writing the head doorbell associated with the original request (❿).

2.2 Software Support

Flash firmware. Figure 2b shows the processes of the FTL, which performs the steps ❸ ∼ ❽. The FTL manages NVMe queues/requests and responds to the host requests by processing the corresponding doorbell. The FTL then performs address translations and manages memory transactions for the flash media. While prior studies [34, 48, 49] distinguish host command controls and flash transaction management as the host interface layer (HIL) and flash interface layer (FIL), respectively, in practice, both modules are implemented as a layered firmware calling through functions of event-based codes with a single thread [57, 65, 70]. The performance of the layered firmware is not on the critical path as flash latency is several orders of magnitude longer than one I/O command

processing latency. However, SSDs require a large number of flash packages and queues to handle more than a thousand requests per *m*sec. When increasing the number of underlying flash packages, the FTL requires powerful computation not only to spread I/O requests across flash packages but also to process I/O commands in parallel. We observe that, compute latency keeps increasing due to non-scalable firmware and takes 93.6% of the total I/O processing time in worst case.

Memory spaces. While the FTL manages the logical block space and physical flash space, it also handles SSD's internal memory space and accesses to host system memory space (cf. Figure 2b). SSDs manage internal memory for caching incoming I/O requests and the corresponding data. Similarly, the FTL uses the internal memory for metadata and NVMe queue management (e.g., SQs/CQs). In addition, the FTL requires accessing the host system memory space to transfer actual data contents over PCIe. Unfortunately, a layered firmware design engages in accesses to memory without any constraint and protection mechanism, which can make the data inconsistent and incoherent in simultaneous accesses. However, computing resources with more parallelism must increase to achieve more than 1MIOPS, and many I/O requests need processing simultaneously. Thus, all shared memory spaces of a manycore SSD platform require appropriate concurrency control and resource protection, similar to virtual memory.

3 Challenges to Exceeding 1MIOPS

To understand the main challenges in scaling SSD firmware, we extend the baseline SSD architecture in a highly scalable environment: Intel many-integrated cores (MIC) [18]. We select this processor platform, because its architecture uses a simple in-order and low-frequency core model, but provides a high core count to study parallelism and scalability. The platform internally emulates low-level flash modules with hardware-validated software[1], so that the flash complex can be extended by adding more emulated channels and flash resources: the number of flash (quad-die package, QDP) varies from 2 to 512. Note that MIC is a prototyping platform used

---

[1]This emulation framework is validated by comparing with Samsung Z-SSD prototype [4], multi-stream 983 DCT (Proof of Concept), 850 Pro [15] and Intel NVMe 750 [25]. **The software will be publicly available.**
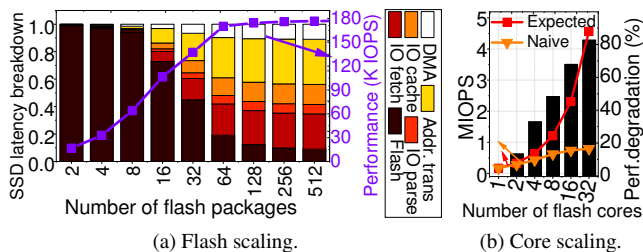
(a) Flash scaling.   (b) Core scaling.

Figure 3: Perf. with varying flash packages and cores.

Figure 4: Many-to-many threading firmware model.

for *only* exploring the limit of scalability, rather than as a suggestion for actual SSD controller.

Flash scaling. The bandwidth of a low-level flash package is several orders of magnitude lower than the PCIe bandwidth. Thus, SSD vendors integrate many flash packages over multiple channels, which can in parallel serve I/O requests managed by NVMe. Figure 3a shows the relationship of bandwidth and execution latency breakdown with various number of flash packages. In this evaluation, we emulate an SSD by creating a layered firmware instance in a single MIC core, in which two threads are initialized to process the tasks of HIL and FTL, respectively. We also assign 16 MIC cores (one core per flash channel) to manage flash interface subsystems. We evaluate the performance of the configured SSD emulation platform by testing 4KB sequential writes. For the breakdown analysis, we decompose total latency into i) NVMe management (`I/O parse` and `I/O fetch`), ii) `I/O cache`, iii) `address translation` (including flash scheduling), vi) NVMe data transfers (`DMA`) and v) flash operations (`Flash`). One can observe from the figure that the SSD performance saturates at 170K IOPS with 64 flash packages, connected over 16 channels. Specifically, the flash operations are the main contributor of the total execution time in cases where our SSD employs tens of flash packages (73% of the total latency). However, as the number of flash packages increases (more than 32), the layered firmware operations on a core become the performance bottleneck. NVMe management and address translation account for 41% and 29% of the total time, while flash only consumes 12% of the total cycles.

There are two reasons that flash firmware turns into the performance bottleneck with many underlying flash devices. First, NVMe queues can supply many I/O requests to take advantages of the SSD internal parallelism, but a single-core SSD controller is insufficient to fetch all the requests. Second, it is faster to parallelize I/O accesses across many flash chips than performing address translation only on one core. These new challenges make it difficult to fully leverage the internal parallelism with the conventional layered firmware model.

Core scaling. To take flash firmware off the critical path in scalable I/O processing, one can increase computing power with the execution of many firmware instances. This approach can allocate a core per NVMe SQ/CQ and initiate one layered firmware instance in each core. However, we observe that this naive approach cannot successfully address the burdens brought by flash firmware. To be precise, we evaluate IOPS
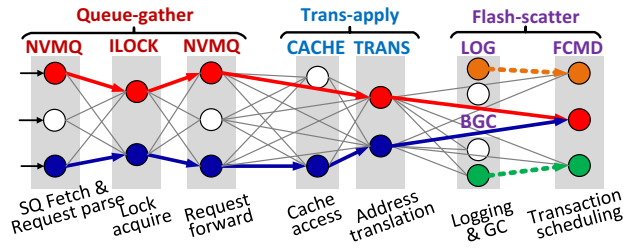
with varying number of cores, ranging from 1 to 32. Figure 3b compares the performance of aforementioned naive manycore approach (e.g., `Naive`) with the system that expects perfect parallel scalability (e.g., `Expected`). `Expected`'s performance is calculated by multiplying the number of cores with IOPS of `Naive` built on a single core SSD. One can observe from this figure that `Naive` can only achieve 813K IOPS even with 32 cores, which exhibits 82.6% lower performance, compared to `Expected`. This is because contention and consistency management for the memory spaces of internal DRAM (cf. Section 5.3) introduces significant synchronization overheads. In addition, the FTL must serialize the I/O requests to avoid hazards while processing many queues in parallel. Since all these issues are not considered by the layered firmware model, it should be re-designed by considering core scaling.

The goal of our new firmware is to fully parallelize multiple NVMe processing datapaths in a highly scalable manner while minimizing the usage of SSD internal resources. `DeepFlash` requires only 12 in-order cores to achieve 1M or more IOPS.

4 Many-to-Many Threading Firmware

Conventional FTL designs are unable to fully convert the computing power brought by a manycore processor to storage performance, as they put all FTL tasks into a single large block of the software stack. In this section, we analyze the functions of the traditional FTLs and decompose them into seven different function groups: 1) *NVMe queue handling (NVMQ)*, 2) *data cache (CACHE)*, 3) *address translation (TRANS)*, 4) *index lock (ILOCK)*, 5) *logging utility (LOG)*, 6) *background garbage collection utility (BGC)*, and 7) *flash command and transaction scheduling (FCMD)*. We then reconstruct the key function groups from the ground up, keeping in mind concurrency, and deploy our reworked firmware modules across multiple cores in a scalable manner.

4.1 Overview

Figure 4 shows our `DeepFlash`'s many-to-many threading firmware model. The firmware is a set of modules (i.e., *threads*) in a request-processing network that is mapped to a set of processors. Each thread can have a firmware operation, and the task can be scaled by instantiating into multiple parallel threads, referred to as *stages*. Based on different data

Figure 5: Firmware architecture.

(a) Data contention (1:N). (b) Unbalanced task. (c) I/O hazard.

Figure 6: Challenges of NVMQ allocation (SQ:NVMQ).

processing flows and tasks, we group the stages into *queue-gather*, *trans-apply*, and *flash-scatter* modules. The queue-gather stage mainly parses NVMe requests and collects them to the SSD-internal DRAM, whereas the trans-apply stage mainly buffers the data and translates addresses. The flash-scatter stage spreads the requests across many underlying flash packages and manages background SSD-internal tasks in parallel. This new firmware enables scalable and flexible computing, and highly parallel I/O executions.

All threads are maximally independent, and I/O requests are always processed from left to right in the thread network, which reduces the hardware contentions and consistency problems, imposed by managing various memory spaces. For example, two i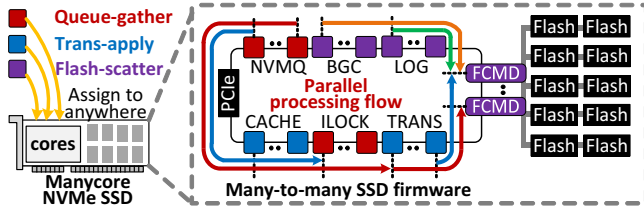ndependent I/O requests are processed by two different network paths (which are highlighted in Figure 4 by red and blue lines, respectively). Consequently, it can simultaneously service incoming I/O requests as many network paths on as DeepFlash can create. In contrast to the other threads, background threads are asynchronous with the incoming I/O requests or host-side services. Therefore, they create their own network paths (dashed lines), which perform SSD internal tasks at background. Since each stage can process a different part of an I/O request, DeepFlash can process multiple requests in a pipeline manner. Our firmware model also can be simply extended by adding more threads based on performance demands of the target system.

Figure 5 illustrates how our many-to-many threading model can be applied to and operate in the many-core based SSD architecture of DeepFlash. While the procedure of I/O services is managed by many threads in the different data processing paths, the threads can be allocated in any core in the network, in a parallel and scalable manner.

4.2   Queue-gather Stage

NVMe queue management. For high performance, NVMe supports up to 64K queues, each up to 64K entries. As shown in Figure 6a, once a host initiates an NVMe command to an SQ and writes the corresponding doorbell, the firmware fetches the command from the SQ and decodes a non-contiguous set of host physical memory pages by referring a kernel list structure [2], called a *physical region page* (PRP) [23]. Since the length of the data in a request can vary, its data can be delivered by multiple data frames, each of which is usually 4KB. While all command information can be retrieved by the device-level registers and SQ, the contents
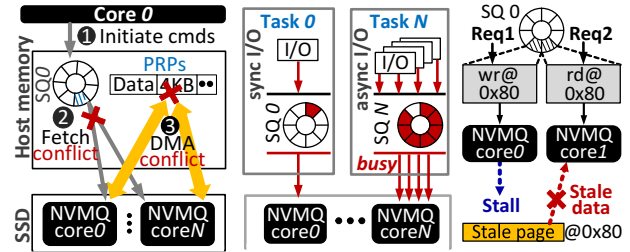
of such data frames exist across non-contiguous host-side DRAM (for a single I/O request). The firmware parses the PRP and begins DMA for multiple data frames per request. Once all the I/O services associated with those data frames complete, firmware notifies the host of completion through the target CQ. We refer to all the tasks, related to this NVMe command and queue management, as *NVMQ*.

A challenge to employ many cores for parallel queue processing is that, multiple NVMQ cores may simultaneously fetch a same set of NVMe commands from a single queue. This in turn accesses the host memory by referring a same set of PRPs, which makes the behaviors of parallel queue accesses undefined and non-deterministic (Figure 6a). To address this challenge, one can make each core handle only a set of SQ/CQ, and therefore, there is no contention, caused by simultaneous queue processing or PRP accesses (Figure 6b). In this "static" queue allocation, each NVMQ core fetches a request from a different queue, based on the doorbell's queue index and brings the corresponding data from the host system memory to SSD internal memory. However, this static approach requires that the host balance requests across queues to maximize the resource utilization of NVMQ threads. In addition, it is difficult to scale to a large number of queues. DeepFlash addresses these challenges by introducing a dynamic I/O serialization, which allows multiple NVMQ threads to access each SQ/CQ in parallel while avoiding a consistency violation. Details of NVMQ will be explained in Section 5.1. **I/O mutual exclusion.** Even though the NVMe specification does not regulate the processing ordering of NVMe commands in a range from where the head pointer indicates to the entry that the tail pointer refers to [3], users may expect that the SSD processes the requests in the order that users submitted. However, in our DeepFlash, many threads can simultaneously process I/O requests in any order of accesses. It can make the order of I/O processing different with the order NVMe queues (and users) expected, which may in turn introduce an I/O hazard or a consistency issue. For example, Figure 6c shows a potential problem brought by parallel I/O processing. In this figure, there are two different I/O requests from the same NVMe SQ, request-1 (a write) and request-2 (a read), which create two different paths, but target to the same PPA. Since these two requests are processed by different NVMQ threads, the request-2 can be served from the target slightly earlier than the request-1. The request-1 then will be
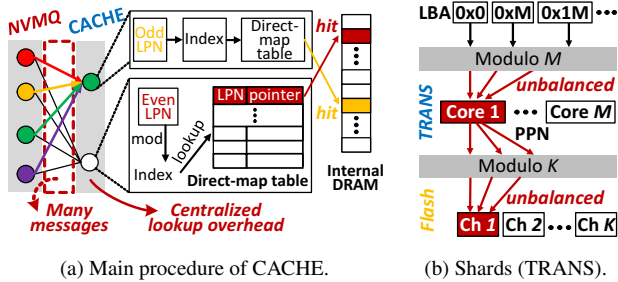
(a) Main procedure of CACHE. (b) Shards (TRANS).

Figure 7: Challenge analysis in CACHE and TRANS.

stalled, and the request-2 will be served with stale data. During this phase, it is also possible that any thread can invalidate the data while transferring or buffering them out of order.

While serializing the I/O request processing with a strong ordering can guarantee data consistency, it significantly hurts SSD performance. One potential solution is introducing a locking system, which provides a lock per page. However, per-page lock operations within an SSD can be one of the most expensive mechanisms due to various I/O lengths and a large storage capacity of the SSD. Instead, we partition physical flash address space into many *shards*, whose access granularity is greater than a page, and assign an index-based lock to each shard. We implement the index lock as a red-black tree and make this locking system as a dedicated thread (ILOCK). This tree helps ILOCK quickly identify which lock to use, and reduces the overheads of lock acquisition and release. Nevertheless, since NVMQ threads may access a few ILOCK threads, it also can be resource contention. DeepFlash optimizes ILOCK by redistributing the requests based on lock ownership (cf., Section 5.2). Note that there is no consistency issue if the I/O requests target different LBAs. In addition, as most OSes manage the access control to prevent different cores from accessing the same files [19, 41, 52], I/O requests from different NVMe queues (mapping to different cores) access different LBAs, which also does not introduce the consistency issue. Therefore, DeepFlash can solve the I/O hazard by guaranteeing the ordering of I/O requests, which are issued to the same queue and access the same LBAs, while DeepFlash can process other I/O requests out of order.

4.3 Trans-apply Stage

Data caching and buffering. To appropriately handle NVMe's parallel queues and achieve more than 1MIOPS, it is important to utilize the internal DRAM buffer efficiently. Specifically, even though modern SSDs enjoy the massive internal parallelism stemming from tens or hundreds of flash packages, the latency for each chip is orders of magnitude longer than DRAM [22, 45, 46], which can stall NVMQ's I/O processing. DeepFlash, therefore, incorporates CACHE threads that incarnate SSD internal memory as a burst buffer by mapping LBAs to DRAM addresses rather than flash ones. The data buffered by CACHE can be drained by striping requests across many flash packages with high parallelism.

As shown in Figure 7a, each CACHE thread has its own mapping table to record the memory locations of the buffered requests. CACHE threads are configured with a traditional direct-map cache to reduce the burden of table lookup or cache replacement. In this design, as each CACHE thread has a different memory region to manage, NVMQ simply calculates the index of the target memory region by modulating the request's LBA, and forwards the incoming requests to the target CACHE. However, since all NVMQ threads possibly communicate with a CACHE thread for every I/O request, it can introduce extra latency imposed by passing messages among threads. In addition, to minimize the number of cores that DeepFlash uses, we need to fully utilize the allocated cores and dedicate them to each firmware operation while minimizing the communication overhead. To this end, we put a cache tag inquiry method in NVMQ and make CACHE threads fully handle cache hits and evictions. With the tag inquiry method, NVMQ can create a bypass path, which can remove the communication overheads (cf. Section 5.3).

Parallel address translation. The FTL manages physical blocks and is aware of flash-specific behavior such as erase-before-write and asymmetric erase and read/write operation unit (block vs. page). We decouple FTL address translation from system management activities such as garbage collection or logging (e.g., journaling) and allocate the management to multiple threads. The threads that perform this simplified address translation are referred to as *TRANS*. To translate addresses in parallel, it needs to partition both LBA space and PPA space and allocate them to each TRANS thread.

As shown in Figure 7b, a simple solution is to split a single LBA space into m numbers of address chunks, where m is the number of TRANS threads, and map the addresses by wrapping around upon reaching m. To take advantage of channel-level parallelism, it can also separate a single PPA space into k shards, where k is the number of underlying channels, and map the shards to each TRANS with arithmetic modulo k. While this address partitioning can make all TRANS threads operate in parallel without interference, unbalanced I/O accesses can activate a few TRANS threads or channels. This can introduce a poor resource utilization and many resource conflicts and stall a request service on the fly. Thus, we randomize the addresses when partitioning the LBA space with simple XOR operators. This can scramble LBA and statically assign all incoming I/O requests across different TRANS threads in an evenly distributed manner. We also allocate all the physical blocks of the PPA space to each TRANS in a round-robin fashion. This block-interleaved virtualization allows us to split the PPA space with finer granularity.

4.4 Flash-scatter Stage

Background task scheduling. The datapath for garbage collection (GCs) can be another critical path to achieve high bandwidth as it stalls many I/O services while reclaiming
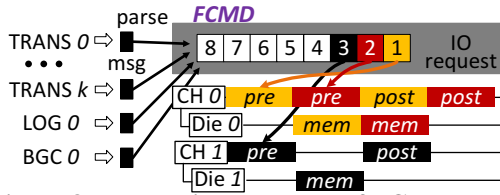
Figure 8: The main procedure of FCMD cores.

flash block(s). In this work, GCs can be performed in parallel by allocating separate core(s), referred to as *BGC*. BGC(s) records the block numbers that have no more entries to write when TRANS threads process incoming I/O requests. BGC then merges the blocks and update the mapping table of corresponding TRANS in behind I/O processing. Since a thread in TRANS can process address translations during BGC's block reclaims, it would introduce a consistency issue on mapping table updates. To avoid conflicts with TRANS threads, BGC reclaims blocks and updates the mapping table at background when there is no activity in NVMQ and the TRANS threads complete translation tasks. If the system experiences a heavy load and clean blocks are running out, our approach performs on-demand GC. To avoid data consistency issue, we only block the execution of the TRANS thread, which is responsible for the address translation of the reclaiming flash block.

Journalling. SSD firmware requires journalling by periodically dumping the local metadata of TRANS threads (e.g., mapping table) from DRAM to a designated flash. In addition, it needs to keep track of the changes, which are not dumped yet. However, managing consistency and coherency for persistent data can introduce a burden to TRANS. Our `DeepFlash` separates the journalling from TRANS and assigns it to a LOG thread. Specifically, TRANS writes the LPN-to-PPN mapping information of a FTL page table entry (PTE) to out-of-band (OoB) of the target flash page [64] in each flash program operation (along with the per-page data). In the meantime, LOG periodically reads all metadata in DRAM, stores them to flash, and builds a checkpoint in the background. For each checkpoint, LOG records a version, a commit and a page pointer indicating the physical location of the flash page where TRANS starts writing to. At a boot time, LOG checks sanity by examining the commit. If the latest version is staled, LOG loads a previous version and reconstructs mapping information by combining the checkpointed table and PTEs that TRANS wrote since the previous checkpoint.

Parallel flash accesses. At the end of the `DeepFlash` network, the firmware threads need to i) compose flash transactions respecting flash interface timing and ii) schedule them across different flash resources over the flash physical layer (PHY). These activities are managed by separate cores, referred to as *FCMD*. As shown in Figure 8, each thread in FCMD parses the PPA translated by TRANS (or generated by BGC/LOG) into the target channel, package, chip and plane numbers. The threads then check the target resources' availability and compose flash transactions by following the underlying flash interface protocol. Typically, memory tim-
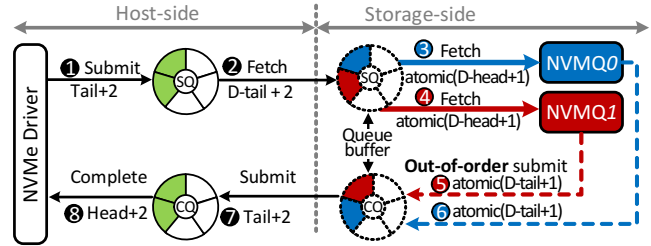
Figure 9: Dynamic I/O serialization (DIOS).

ings within a flash transaction can be classified by *pre-dma*, *mem-op* and *post-dma*. While pre-dma includes operation command, address, and data transfer (for writes), post-dma is composed by completion command and another data transfer (for reads). Memory operations of the underlying flash are called mem-op in this example. FCMD(s) then scatters the composed transactions over multiple flash resources. During this time, all transaction activities are scheduled in an interleaved way, so that it can maximize the utilization of channel and flash resources. The completion order of multiple I/O requests processed by this transaction scheduling can be spontaneously an out-of-order.

In our design, each FCMD thread is statically mapped to one or more channels, and the number of channels that will be assigned to the FCMD thread is determined based on the SSD vendor demands (and/or computing power).

5 Optimizing DeepFlash

While the baseline `DeepFlash` architecture distributes functionality with many-to-many threading, there are scalability issues. In this section, we will explain the details of thread optimizations to increase parallel scalability that allows faster, more parallel implementations.

5.1 Parallel Processing for NVMe Queue

To address the challenges of the static queue allocation approach, we introduce the *dynamic I/O serialization (DIOS)*, which allows a variable ratio of queues to cores. DIOS decouples the fetching and parsing processes of NVMe queue entries. As shown in Figure 9, once a NVMQ thread fetches a batch of NVMe commands from a NVMe queue, other NVMQ threads can simultaneously parse the fetched NVMe queue entries. This allows all NVMQ threads to participate in processing the NVMe queue entries from the same queue or multiple queues. Specifically, DIOS allocates a storage-side SQ buffer (per SQ) in a shared memory space (visible to all NVMQ threads) when the host initializes NVMe SSD. If the host writes the tail index to the doorbell, a NVMQ thread fetches multiple NVMe queue entries and copies them (not actual data) to the SQ buffer. All NVMQ threads then process the NVMe commands existing in the SQ buffer in parallel. The batch copy is performed per 64 entries or till the tail for SQ and CQ points a same position. Similarly, DIOS creates

a CQ buffer (per CQ) in the shared memory. NVMQ threads update the CQ buffer instead of the actual CQ as an out of order, and flush the NVMe completion messages from the CQ buffer to the CQ in batch. This allows multiple threads update an NVMe queue in parallel without a modification of the NVMe protocol and host side storage stack. Another technical challenge for processing a queue in parallel is that the head and tail pointers of SQ and CQ buffers are also shared resources, which requires a protection for simultaneous access. `DeepFlash` offers DIOS's head (*D-head*) and tail (*D-tail*) pointers, and allows NVMQ threads to access SQ and CQ through those pointers, respectively. Since D-head and D-tail pointers are managed by gcc atomic built-in function, `__sync_fetch_and_add` [21], and the core allocation is performed by all NVMQ threads, in parallel, the host memory can be simultaneously accessed but at different locations.

5.2   Index Lock Optimization

When multiple NVMQ threads contend to acquire or release the same lock due to their same target address range, it can raise two technical issues: i) lock contention and ii) low resource utilization of NVMQ. As shown in Figure 10a, an ILOCK thread sees all incoming lock requests (per page by LBA) through its message queue. This queue sorts the messages based on SQ indices, and each message maintains thread request structure that includes an SQ index, NVMQ ID, LBA, and lock request information (e.g., acquire and release). Since the order of queue's lock requests is non-deterministic, in a case of contention on acquisition, it must perform I/O services by respecting the order of requests in the corresponding SQ. Thus, the ILOCK thread infers the SQ order by referring to the SQ index in the message queue if the target LBA with the lock request has a conflict. It then checks the red-black (RB) tree whose LBA-indexed node contains the lock number and owner ID that already acquired the corresponding address. If there is no node in the lock RB tree, the ILOCK thread allocates a node with the request's NVMQ ID. When ILOCK receives a release request, it directly removes the target node without an SQ inference process. If the target address is already held by another NVMQ thread, the lock requester can be stalled until the corresponding I/O service is completed. Since low-level flash latency takes hundreds of microseconds to a few milliseconds, the stalled NVMQ can hurt overall performance. In our design, ILOCK returns the owner ID for all lock acquisition requests rather than returning simply acquisition result (e.g., false or fail). The NVMQ thread receives the ID of the owning NVMQ thread, and can forward the request there to be processed rather than communicating with ILOCK again. Alternatively, the NVMQ thread can perform other tasks, such as issuing the I/O service to TRANS or CACHE. The insight behind this *forwarding* is that if another NVMQ owns the corresponding lock of request, then forwards the request to owner and stop further communication with ILOCK.

(a) The main procedure of ILOCK.

(b) The example of CACHE bypassing.
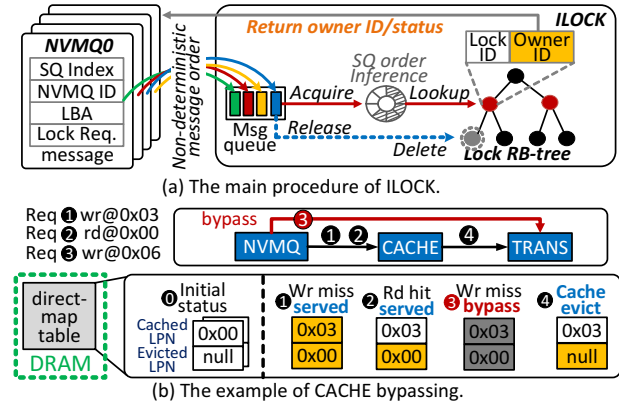
Figure 10: Optimization details.

This, in turn, can free the NVMQ thread from waiting for the lock acquisition, which increases the parallelism of DIOS.

5.3   Non-blocking Cache

To get CACHE off the critical path, we add a direct path between NVMQ and TRANS threads and make NVMQ threads access CACHE threads "only if" there is data in CACHE. We allocate direct-map table(s) in a shared memory space to accommodate the cache metadata so that NVMQ threads can lookup the cache metadata on their own and send I/O requests only if there is a hit. However, this simple approach may introduce inconsistency between the cache metadata of the direct-map table and target data of the CACHE. When a write evicts a dirty page from the burst buffer, the metadata of such evicted page is removed from the direct-map table immediately. However, the target data of the evicted page may still stay in the burst buffer, due to the long latency of a flash write. Therefore, when a dirty page is in progress of eviction, read requests, which target for the same page, may access stale data from the flash. To coordinate the direct-map table and CACHE correctly, we add "evicted LPN" field in each map table entry that presents the page number, being in eviction (cf. Figure 10b). In this example of the figure, we assume the burst buffer is a direct mapped cache with 3 entries. The request (Req ❶) evicts the dirty page at LPN 0x00. Thus, NVMQ records the LPN of Req ❶ in the cached LPN field of the direct-map table and moves the address of the evicted page to its evicted LPN field. Later, as the LPN of Req ❷ (the read at 0x00) matches with the evicted LPN field, Req ❷ is served by CACHE instead of accessing the flash. If CACHE is busy in evicting the dirty page at LPN 0x00, Req ❸ (the write at 0x06) has to be stalled. To address this, we make Req ❸ directly bypass CACHE. Once the eviction successfully completes, CACHE clears the evicted LPN field (❹).

To make this *non-blocking cache* more efficient, we add a simple randomizing function to retrieve the target TRANS index for NVMQ and CACHE threads, which can evenly distribute their requests in a static manner. This function performs an XOR operation per bit for all the bit groups

and generates the target TRANS index, which takes less than 20 ns. The randomization allows queue-gather stages to issue requests to TRANS by addressing load imbalance.

6 Evaluation

Implementation platform. We set up an accurate SSD emulation platform by respecting the real NVMe protocol, the timing constraints for flash backbone and the functionality of a flexible firmware. Specifically, we emulate a manycore-based SSD firmware by using a MIC 5120D accelerator that employs 60 lightweight in-order cores (4 hardware threads per core) [28]. The MIC cores operate at 1GHz and are implemented by applying low power techniques such as short in-order pipeline. We emulate the flash backbone by modelling various flash latencies, different levels of parallelism (i.e., channel/way/flash) and the request conflicts for flash resources. Our flash backbone consists of 16 channels, each connecting 16 QDP flash packages [69]; we observed that the performance of both read and write operations on the backbone itself is not the bottleneck to achieve more than 1 MIOPS. The NVMe interface on the accelerator is also fully emulated by wrapping Intel's symmetric communications interface (SCIF) with an NVMe emulation driver and controller that we implemented. The host employs a Xeon 16-core processor and 256 GB DRAM, running Linux kernel 2.6.32 [62]. It should be noted that this work uses MIC to explore the scalability limits of the design; the resulting software can run with fewer cores if they are more powerful, but the design can now be about what is most economic and power efficient, rather than whether the firmware can be scalable.

Configurations. `DeepFlash` is the emulated SSD platform including all the proposed designs of this paper. Compared to `DeepFlash`, `BaseDeepFlash` does not apply the optimization techniques (described in Section 5). We evaluate the performance of a real Intel customer-grade SSD (`750SSD`) [25] and high-performance NVMe SSD (`4600SSD`) [26] for a better comparison. We also emulate another SSD platform (`ManyLayered`), which is an approach to scale up the layered firmware on many cores. Specifically, `ManyLayered` statically splits the SSD hardware resources into multiple subsets, each containing the resources of one flash channel and running a layered firmware independently. For each layered firmware instance, `ManyLayered` assigns a pair of threads: one is used for managing flash transaction, and another is assigned to run HIL and FTL. All these emulation platforms use "12 cores" by default. Lastly, we also test different flash technologies such as SLC, MLC, TLC, each of which latency characteristics are extracted from [44], [45] and [46], respectively. By default, the MLC flash array in pristine state is used for our evaluations. The details of SSD platform are in Table 1.

Workloads. In addition to microbenchmarks (reads and writes with sequential and random patterns), we test diverse server workloads, collected from Microsoft Production Server

(MPS) [35], FIU SRCMap [63], Enterprise, and FIU IOD-edup [40]. Each workload exhibits various request sizes, ranging from 4KB to tens of KB, which are listed in Table 1. Since all the workload traces are collected from the narrow-queue SATA hard disks, replaying the traces with the original times-tamps cannot fully utilize the deep NVMe queues, which in turn conceals the real performance of SSD [29]. To this end, our trace replaying approach allocates 16 worker threads in the host to keep issuing I/O requests, so that the NVMe queues are not depleted by the SSD platforms.

6.1 Performance Analysis

Microbenchmarks. Figure 11 compares the throughput of the five SSD platforms with I/O sizes varying from 4KB to 32KB. Overall, `ManyLayered` outperforms `750SSD` and `4600SSD` by $1.5\times$ and 45%, on average, respectively. This is because `ManyLayered` can partially take the benefits of many-core computing and parallelize I/O processing across multiple queues and channels over the static resource partitioning. `BaseDeepFlash` exhibits poor performance in cases that the request size is smaller than 24KB with random patterns. This is because threads in NVMQ/ILOCK keep tight inter-thread communications to appropriately control the consistency over locks. However, for large requests (32KB), `BaseDeepFlash` exhibits good performance close to `ManyLayered`, as multiple pages in large requests can be merged to acquire one range lock, which reduces the communication (compared to smaller request sizes), and thus, it achieves higher bandwidth.

We observe that `ManyLayered` and `BaseDeepFlash` have a significant performance degradation in random reads and random writes (cf. Figures 11b and 11d). `DeepFlash`, in contrast, provides more than 1MIOPS in all types of I/O requests; 4.8 GB/s and 4.5 GB/s bandwidth for reads and writes, respectively. While those many-core approaches suffer from many core/flash-level conflicts (`ManyLayered`) and lock/sync issues (`BaseDeepFlash`) on the imbalanced random workloads, `DeepFlash` scrambles the LBA space and evenly distributes all the random I/O requests to different TRANS threads with a low overhead. In addition, it applies cache bypass and lock forwarding techniques to mitigate the long stalls, imposed by lock inquiry and inter-thread communication. This can enable more threads to serve I/O requests in parallel.

As shown in Figure 12, `DeepFlash` can mostly activate 6.3 cores that run 25 threads to process I/O services in parallel, which is better than `BaseDeepFlash` by 127% and 63% for reads and writes, respectively. Note that, for the random writes, the bandwidth of `DeepFlash` is sustainable (4.2 GB/s) by activating only 4.5 cores (18 threads). This is because although many cores contend to acquire ILOCK which makes more cores stay in idle, the burst buffer successfully overcomes the long write latency of the flash.

Figure 12e shows the active core decomposition of `DeepFlash`. As shown in the figure, reads require 23% more

Table 1: H/W configurations and Important workload characteristics of the workloads that we tested.

Host		Workloadsets	Microsoft,Production Server							FIU IODedup		
CPU/mem	Xeon 16-core processor/256GB, DDR4	Workloads	24HR	24HRS	BS	CFS	DADS	DAP	DDR	cheetah	homes	webonline
Storage platform/firmware		Read Ratio	0.06	0.13	0.11	0.82	0.87	0.57	0.9	0.99	0	0
Controller	Xeon-phi, 12 cores by default	Avg length (KB)	7.5	12.1	26.3	8.6	27.6	63.4	12.2	4	4	4
FTL/buffer	hybrid, n:m=1:8, 1 GB/512 MB	Randomness	0.3	0.4937	0.87	0.94	0.99	0.38	0.313	0.12	0.14	0.14
Flash	16 channels/16 pkgs per channel/1k blocks per die	Workloadsets	FIU SRCMap							Enterprise		
array	512GB(SLC),1TB(MLC),1.5TB(TLC)	Workloads	ikki	online	topgun	webmail	casa	webresearch	webusers	madmax	Exchange	
SLC	R: 25us, W: 300us, E: 2ms, Max: 1.4 MIOPS	Read Ratio	0	0	0	0	0	0	0	0.002	0.24	
MLC	R: 53us, W: 0.9ms, E: 2.3ms, Max: 1.3 MIOPS	Avg length (KB)	4	4	4	4	4	4	4	4.005	9.2	
TLC	R: 78us, W: 2.6ms, E: 2.3ms, Max: 1.1 MIOPS	Randomness	0.39	0.17	0.14	0.21	0.65	0.11	0.14	0.08	0.84	

Figure 11: Performance comparison.
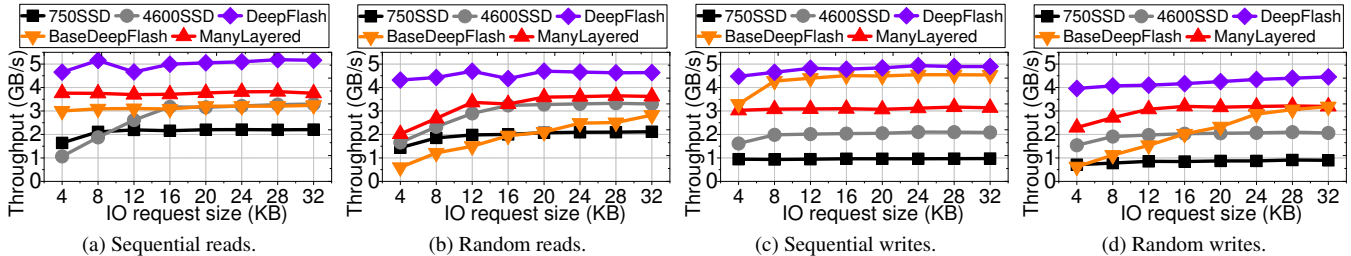(a) Sequential reads. (b) Random reads. (c) Sequential writes. (d) Random writes.

Figure 13: Overall throughput analysis.

Flash activities, compared to writes, as they are accommodated in internal DRAM. In addition, NVMQ requires 1.5% more compute resources to process write requests than processing read requests, which can offer slightly worse bandwidth on writes, compared to that of reads. Note that background activities such as garbage collection and logging are not invoked during this evaluation as we configured the emulation platform as a pristine SSD.

Server workload traces. Figure 13 illustrates the throughput of server workloads. As shown in the figure, `BaseDeepFlash` exhibits 1.6, 2.7, 1.1, and 2.9 GB/s, on average, for MPS, SR-CMap, Enterprise and IODedup workload sets, respectively, and `DeepFlash` improves those of `BaseDeepFlash`, by 260%, 64%, 299% and 35%, respectively. `BaseDeepFlash` exhibits a performance degradation, compared to `ManyLayered` with MPS. This is because MPS generates multiple lock contentions, due to more small-size random accesses than other workloads (c.f. Table 1). Interestingly, while `DeepFlash` outperforms other SSD platforms in most workloads, its performance is not as good under *DDR* workloads (slightly better than `BaseDeepFlash`). This is because FCMD utilization is

lower than 56% due to the address patterns of *DDR*. However, since all NVMQ threads parse and fetch incoming requests in parallel, even for such workloads, `DeepFlash` provides 3 GB/s, which is 42% better than `ManyLayered`.

6.2 CPU, Energy and Power Analyses

CPU usage and different flashes. Figures 14a and 14b show sensitivity analysis for bandwidth and power/energy, respectively. In this evaluation, we collect and present the performance results of all four microbenchmarks by employing a varying number of cores (2∼19) and different flash technologies (SLC/MLC/TLC). The overall SSD bandwidth starts to saturate from 12 cores (48 hardware threads) for the most case. Since TLC flash exhibits longer latency than SLC/MLC flash, TLC-based SSD requires more cores to reduce the firmware latency such that it can reach 1MIOPS. When we increase the number of threads more, the performance gains start to diminish due to the overhead of exchanging many messages among thread groups. Finally, when 19 cores are employed, SLC, MLC, and TLC achieve the maximum bandwidths that all the underlying flashes aggregately expose, which are 5.3, 4.8, and 4.8 GB/s, respectively.

Power/Energy. Figure 14b shows the energy breakdown of each SSD stage and the total core power. The power and energy are estimated based on an instruction-level energy/power model of Xeon Phi [55]. As shown in Figure 14b, `DeepFlash` with 12 cores consumes 29 W, which can satisfy the power delivery capability of PCIe [51]. Note that while this power con-

(a) Sequential reads. (b) Random reads. (c) Sequential writes. (d) Random writes. (e) Core decomposition.
Figure 12: Dynamics of active cores for parallel I/O processing.

(a) Bandwidth.   (b) Breakdown.   (c) Cores.
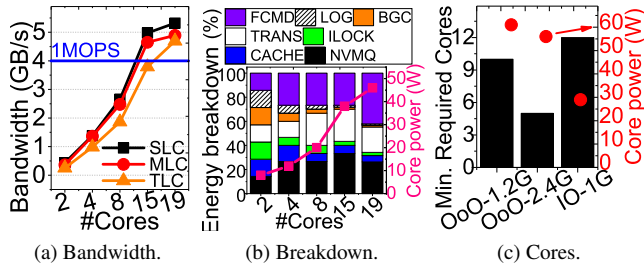
Figure 14: Resource requirement analysis.

(a) NVMQ performance.   (b) IOPS per NVMQ thread.

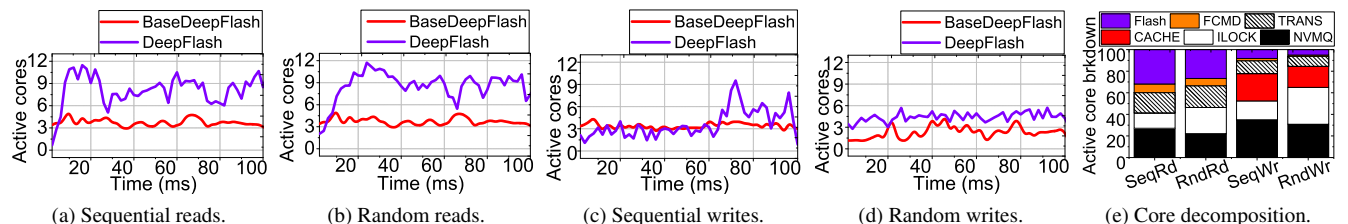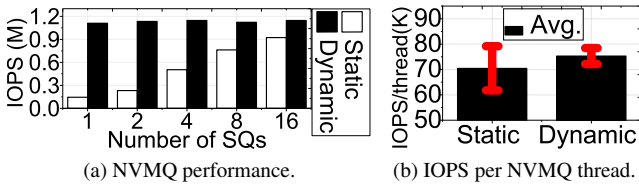**Figure 15: Performance on different queue allocations.**

sumption is higher than existing SSDs (20 ∼ 30W [30,58,73]), power-efficient manycores [68] can be used to reduce the power of our prototype. When we break down energy consumed by each stage, FCMD, TRANS and NVMQ consume 42%, 21%, and 26% of total energy, respectively, as the number of threads increases. This is because while CACHE, LOG, ILOCK, and BGC require more computing power, most cores should be assigned to handle a large flash complex, many queues and frequent address translation for better scalability. **Different CPUs.** Figure 14c compares the minimum number of cores that DeepFlash requires to achieve 1MIOPS for both reads and writes. We evaluate different CPU technologies: i) OoO-1.2G, ii) OoO-2.4G and iii) IO-1G. While IO-1G uses the default in-order pipeline 1GHz core that our emulation platform employs, OoO-1.2G and OoO-2.4G employ Intel Xeon CPU, an out-of-order execution processor [24] with 1.2 and 2.4GHz CPU frequency, respectively. One can observe from the figure that a dozen of cores that DeepFlash uses can be reduced to five high-frequency cores (cf. OoO-2.4G). However, due to the complicated core logic (e.g., reorder buffer), OoO-1.2G and OoO-2.4G consume 93% and 110% more power than IO-1G to achieve the same level of IOPS.

6.3 Performance Analysis of Optimization

In this analysis, we examine different design choices of the components in DeepFlash and evaluate their performance impact on our proposed SSD platform. The following experiments use the configuration of DeepFlash by default. **NVMQ.** Figures 15a and 15b compare NVMQ's IOPS and per-thread IOPS, delivered by a non-optimized queue allocation (i.e., Static) and our DIOS (i.e., Dynamic), respectively. Dynamic achieves the bandwidth goal, irrespective of the number of NVMe queues that the host manages, whereas Static requires more than 16 NVMe queues to achieve 1MIOPS (cf. Figure 15a). This implies that the host also requires more cores since the NVMe allocates a queue per host

(a) ILOCK impact.   (b) CACHE IOPS.
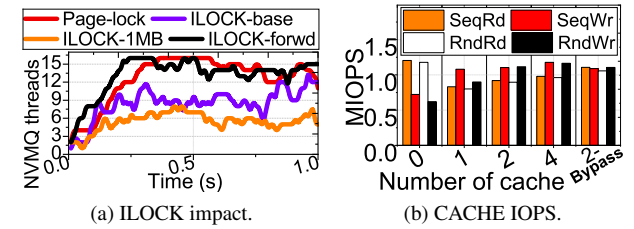
Figure 16: ILOCK and CACHE optimizations.

(a) LOG/BGC.   (b) BGC overhead.
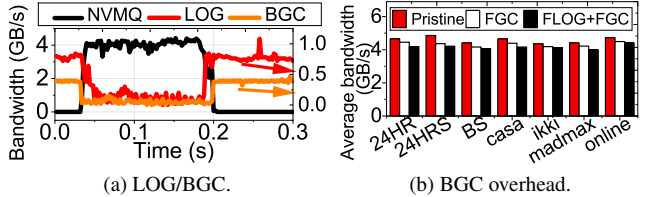
Figure 17: Background task optimizations.

CPU core [10]. Furthermore, the per-thread IOPS of Dynamic (with 16 queues) is better than Static by 6.9% (cf. Figure 15b). This is because Dynamic can fully utilize all NVMQ threads when the loads of different queues are unbalanced; the NVMQ performance variation of Dynamic (between min and max) is only 12%, whereas that of Static is 48%. **ILOCK.** Figure 16a compares the different locking systems. Page-lock is a page-granular lock, while ILOCK-base is ILOCK that has no ownership forwarding. ILOCK-forwd is the one that DeepFlash employs. While ILOCK-base and ILOCK-forwd use a same granular locking (256KB), ILOCK-1MB employs 1MB for its lock range but has no forwarding. Page-lock can activate NVMQ threads more than ILOCK-1MB by 82% (Figure 16a). However, due to the overheads imposed by frequent lock node operations and RB tree management, the average lock inquiry latency of Page-lock is as high as 10 us, which is 11× longer than that of ILOCK-forwd. In contrast, ILOCK-forwd can activate the similar number of NVMQ threads as Page-lock, and exhibits 0.93 us average lock inquiry latency. **CACHE.** Figure 16b illustrates CACHE performance with multiple threads varying from 0 to 4. "2-Bypass" employs the bypass technique (with only 2 threads). Overall, the read performance (even with no-cache) is close to 1MIOPS, thanks to massive parallelism in back-end stages. However, write performance with no-cache is only around 0.65 MIOPS, on average. By enabling a single CACHE thread to buffer data in SSD internal DRAM rather than underlying flash media, write bandwidth increases by 62%, compared to the system of no-cache. But single CACHE thread reduces read bandwidth by 25%, on average, due to communication overheads (between CACHE and NVMQ) for each I/O service. Even with more CACHE threads, performance gains diminish due to communication overhead. In contrast, DeepFlash's 2-Bypass can be ideal as it requires fewer threads to achieve 1MIOPS. **Background activities.** Figure 17a shows how DeepFlash coordinates NVMQ, LOG and BGC threads to avoid contentions on flash resources and maximize SSD performance.

As shown in the figure, when NVMQ actively parses and fetches data (between 0.04 and 0.2 s), LOG stops draining the data from internal DRAM to flash, since TRANS needs to access their meta information as a response of NVMQ's queue processing. Similarly, BGC also suspends the block reclaiming since data migration (associated to the reclaim) may cause flash-level contentions, thereby interfering NVMQ's activities. As `DeepFlash` can minimize the impact from LOG and BGC, the I/O access bandwidth stays above 4 GB/s. Once NVMQ is in idle, LOG and BGC reactivate their work.

STEADY-STATE performance. Figure 17b shows the impact of on-demand garbage collection (FGC) and journalling (FLOG) on the performance of `DeepFlash`. The results are compared to the ideal performance of `DeepFlash` (`Pristine`), which has no GC and LOG activities. Compared to `Pristine`, the performance of FGC degrades by 5.4%, while FLOG+FGC decreases the throughput by 8.8%, on average. The reason why there is negligible performance loss is that on-demand GC only blocks single TRANS thread that manages the reclaimed flash block, while the remaining TRANS threads keep serving the I/O requests. In the meantime, LOG works in parallel with TRANS, but consumes the usage of FCMD to dump data.

7 Related Work and Discussion

OS optimizations. To achieve higher IOPS, host-level optimization on multicore systems [8, 36, 75] have been studied. Bjorling *et al.* changes Linux block layer in OS and achieves 1MIOPS on the high NUMA-factor processor systems [8]. Zheng *et al.* redesigns buffer cache on file systems and reinvent overhead and lock-contention in a 32-core NUMA machine to achieve 1MIOPS [75]. All these systems exploit heavy manycore processors on the host and buffer data atop SSDs to achieve higher bandwidth.

Industry trend. To the best of our knowledge, while there are no manycore SSD studies in literature, industry already begun to explore manycore based SSDs. Even though they do not publish the actual device in publicly available market, there are several devices that partially target to 1MIOPS. For example, FADU is reported to offer around 1MIOPS (only for sequential reads with prefetching) and 539K IOPS (for writes) [20]; Samsung PM1725 offers 1MIOPS (for reads) and 120K IOPS (for writes). Unfortunately, there are no information regarding all industry SSD prototypes and devices in terms of hardware and software architectures. We believe that future architecture requires brand-new flash firmware for scalable I/O processing to reach 1MIOPS.

Host-side FTL. LightNVM [9], including CNEX solution [1], aims to achieve high performance (∼1MIOPS) by moving FTL to the host and optimizing user-level and host-side software stack. But their performance are achieved by evaluating only specific operations (like reads or sequential accesses). In contrast, `DeepFlash` reconstructs device-level software/hardware with an in-depth analysis and offers 1MIOPS for all microbenchmarks (read, write, sequential and random) with varying I/O sizes. In addition, our solution is orthogonal to (and still necessary for) host-side optimizations.

Emulation. There is unfortunately no open hardware platform, employing multiple cores and flash packages. For example, OpenSSD has two cores [59], and Dell/EMC's Openchannel SSD (only opens to a small and verified community) also employs 4∼8 NXP cores on a few flash [17]. Although this is an emulation study, we respected all real NVMe/ONFi protocols and timing constraints for SSD and flash, and the functionality and performance of flexible firmware are demonstrated by a real lightweight many-core system.

Scale-out vs. scale-up options. A set of prior work proposes to architect the SSD as the RAID0-like scale-out option. For example, Amfeltec introduces an M.2-to-PCIe carrier card, which can include four M.2 NVMe SSDs as the RAID0-like scale-up solution [5]. However, this solution only offers 340K IOPS due to the limited computing power. Recently, CircuitBlvd overcomes such limitation by putting eight carrier cards into a storage box [6]. Unfortunately, this scale-out option also requires two extra E5-2690v2 CPUs (3.6GHz 20 cores) with seven PCIe switches, which consumes more than 450W. In addition, these scale-out solutions suffer from serving small-sized requests with a random access-pattern (less than 2GB/sec) owing to frequent interrupt handling and I/O request coordination mechanisms. In contrast, `DeepFlash`, as an SSD scale-up solution, can achieve promising performance of random accesses by eliminating the overhead imposed by such RAID0 design. In addition, compared to the scale-out options, `DeepFlash` employs fewer CPU cores to execute only SSD firmware, which in turn reduces the power consumption.

8 Conclusion

In this work, we designed scalable flash firmware inspired by parallel data analysis systems, which can extract the maximum performance of the underlying flash memory complex by concurrently executing multiple firmware components within a single device. Our emulation prototype on a manycore-integrated accelerator reveals that it simultaneously processes beyond 1MIOPS, while successfully hiding long latency imposed by internal flash media.

9 Acknowledgement

References

[1] CNEX Labs. https://www.cnexlabs.com.

[2] Microsoft SGL Description. https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/using-scatter-gather-dma.

[3] Nvm express. http://nvmexpress.org/wp-content/uploads/NVM-Express-1_3a-20171024_ratified.pdf.

[4] Ultra-low Latency with Samsung Z-NAND SSD. http://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf, 2017.

[5] Squid carrier board family pci express gen 3 carrier board for 4 m.2 pcie ssd modules. https://amfeltec.com/pci-express-gen-3-carrier-board-for-m-2-ssd/, 2018.

[6] Cinabro platform v1. https://www.circuitblvd.com/post/cinabro-platform-v1, 2019.

[7] Jasmin Ajanovic. PCI express 3.0 overview. In *Proceedings of Hot Chip: A Symposium on High Performance Chips*, 2009.

[8] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block IO: introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, page 22. ACM, 2013.

[9] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *FAST*, pages 359–374, 2017.

[10] Keith Busch. Linux NVMe driver. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130812_PreConfD_Busch.pdf, 2013.

[11] Adrian M Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K Gupta, Allan Snavely, and Steven Swanson. Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[12] Adrian M Caulfield, Laura M Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. *ACM Sigplan Notices*, 44(3):217–228, 2009.

[13] Wonil Choi, Myoungsoo Jung, Mahmut Kandemir, and Chita Das. Parallelizing garbage collection with i/o to improve flash resource utilization. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 243–254, 2018.

[14] Wonil Choi, Jie Zhang, Shuwen Gao, Jaesoo Lee, Myoungsoo Jung, and Mahmut Kandemir. An in-depth study of next generation interface for emerging non-volatile memories. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2016 5th*, pages 1–6. IEEE, 2016.

[15] cnet. Samsung 850 Pro SSD review. https://www.cnet.com/products/samsung-ssd-850-pro/, 2015.

[16] Danny Cobb and Amber Huffman. NVM Express and the PCI Express SSD revolution. In *Intel Developer Forum. Santa Clara, CA, USA: Intel*, 2012.

[17] Jae Do. SoftFlash: Programmable storage in future data centers. https://www.snia.org/sites/default/files/SDC/2017/presentations/Storage_Architecture/Do_Jae_Young_SoftFlash_Programmable_Storage_in_Future_Data_Centers.pdf, 2017.

[18] Alejandro Duran and Michael Klemm. The Intel® many integrated core architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 365–366. IEEE, 2012.

[19] FreeBSD. Freebsd manual pages: flock. https://www.freebsd.org/cgi/man.cgi?query=flock&sektion=2, 2011.

[20] Anthony Garreffa. Fadu unveils world's fastest SSD, capable of 5gb/sec. http://tiny.cc/eyzdcz, 2016.

[21] Arthur Griffith. *GCC: the complete reference*. McGraw-Hill, Inc., 2002.

[22] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 2–2. USENIX Association, 2012.

[23] Amber Huffman. NVM Express, revision 1.0 c. *Intel Corporation*, 2012.

[24] Intel. Intel Xeon Processor E5 2620 v3. http://tiny.cc/a1zdcz, 2014.

[25] Intel. Intel SSD 750 series. http://tiny.cc/qyzdcz, 2015.

[26] Intel. Intel SSD DC P4600 Series. *http://tiny.cc/dzzdcz*, 2018.

[27] Xabier Iturbe, Balaji Venu, Emre Ozer, and Shidhartha Das. A triple core lock-step (TCLS) ARM® Cortex®-R5 processor for safety-critical and ultra-reliable applications. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pages 246–249. IEEE, 2016.

[28] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[29] 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 the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 61–74, 2014.

[30] Myoungsoo Jung. Exploring design challenges in getting solid state drives closer to cpu. *IEEE Transactions on Computers*, 65(4):1103–1115, 2016.

[31] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T Kandemir. Hios: A host interface i/o scheduler for solid state disks. *ACM SIGARCH Computer Architecture News*, 42(3):289–300, 2014.

[32] Myoungsoo Jung and Mahmut Kandemir. Revisiting widely held SSD expectations and rethinking system-level implications. In *ACM SIGMETRICS Performance Evaluation Review*, volume 41, pages 203–216. ACM, 2013.

[33] Myoungsoo Jung and Mahmut T Kandemir. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 524–535. IEEE, 2014.

[34] Myoungsoo Jung, Ellis H Wilson III, and Mahmut Kandemir. Physically addressed queueing (PAQ): improving parallelism in solid state disks. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 404–415. IEEE Computer Society, 2012.

[35] Bruce Worthington Qi Zhang Kavalanekar, Swaroop and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *IISWC*, 2008.

[36] Byungseok Kim, Jaeho Kim, and Sam H Noh. Managing array of ssds when the storage device is no longer the performance bottleneck. In *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[37] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8(4):14, 2012.

[38] Nathan Kirsch. Phison E12 high-performance SSD controller. *http://tiny.cc/91zdcz*, 2018.

[39] Sungjoon Koh, Junhyeok Jang, Changrim Lee, Miryeong Kwon, Jie Zhang, and Myoungsoo Jung. Faster than flash: An in-depth study of system challenges for emerging ultra-low latency ssds. *arXiv preprint arXiv:1912.06998*, 2019.

[40] Ricardo Koller et al. I/O deduplication: Utilizing content similarity to improve I/O performance. *TOS*, 2010.

[41] Linux. Mandatory file locking for the linux operating system. *https://www.kernel.org/doc/Documentation/filesystems/mandatory-locking.txt*, 2007.

[42] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):5, 2017.

[43] marvell. Marvell 88ss1093 flash memory controller. *https://www.marvell.com/storage/assets/Marvell-88SS1093-0307-2017.pdf*, 2017.

[44] Micron. Mt29f2g08aabwp/mt29f2g16aabwp NAND flash datasheet. 2004.

[45] Micron. Mt29f256g08cjaaa/mt29f256g08cjaab NAND flash datasheet. 2008.

[46] Micron. Mt29f1ht08emcbbj4-37:b/mt29f1ht08emhbbj4-3r:b NAND flash datasheet. 2016.

[47] Yongseok Oh, Eunjae Lee, Choulseung Hyun, Jongmoo Choi, Donghee Lee, and Sam H Noh. Enabling cost-effective flash based caching with an array of commodity ssds. In *Proceedings of the 16th Annual Middleware Conference*, pages 63–74. ACM, 2015.

[48] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: software-defined flash for web-scale internet storage systems. *ACM SIGPLAN Notices*, 49(4):471–484, 2014.

[49] Seon-yeong Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng, and Joonwon Lee. Exploiting internal parallelism of flash-based SSDs. *IEEE Computer Architecture Letters*, 9(1):9–12, 2010.

[50] Chris Ramseyer. Seagate SandForce SF3500 client SSD controller detailed. *http://tiny.cc/f2zdcz*, 2015.

[51] Tim Schiesser. Correction: PCIe 4.0 won't support up to 300 watts of slot power. *http://tiny.cc/52zdcz*, 2017.

[52] Windows SDK. Lockfileex function. *https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-lockfileex*, 2018.

[53] Hynix Semiconductor et al. Open NAND flash interface specification. *Technical Report ONFI*, 2006.

[54] Narges Shahidi, Mahmut T Kandemir, Mohammad Arjomand, Chita R Das, Myoungsoo Jung, and Anand Sivasubramaniam. Exploring the potentials of parallel garbage collection in ssds for enterprise storage systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 561–572. IEEE, 2016.

[55] Yakun Sophia Shao and David Brooks. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 389–394. IEEE, 2013.

[56] Mustafa M Shihab, Jie Zhang, Myoungsoo Jung, and Mahmut Kandemir. Revenand: A fast-drift-aware resilient 3d nand flash design. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(2):1–26, 2018.

[57] Ji-Yong Shin, Zeng-Lin Xia, Ning-Yi Xu, Rui Gao, Xiong-Fei Cai, Seungryoul Maeng, and Feng-Hsiung Hsu. FTL design exploration in reconfigurable high-performance SSD for server applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 338–349. ACM, 2009.

[58] S Shin and D Shin. Power analysis for flash memory SSD. *Work-shop for Operating System Support for Non-Volatile RAM (NVRAMOS 2010 Spring)(Jeju, Korea, April 2010)*, 2010.

[59] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. Cosmos openSSD: A PCIe-based open source SSD platform. *Proc. Flash Memory Summit*, 2014.

[60] Wei Tan, Liana Fong, and Yanbin Liu. Effectiveness assessment of solid-state drive used in big data services. In *Web Services (ICWS), 2014 IEEE International Conference on*, pages 393–400. IEEE, 2014.

[61] 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 *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 49–66, 2018.

[62] Linus Torvalds. Linux kernel repo. *https://github.com/torvalds/linux*, 2017.

[63] Akshat Verma, Ricardo Koller, Luis Useche, and Raju Rangaswami. SRCMap: Energy proportional storage using dynamic consolidation. In *FAST*, volume 10, pages 267–280, 2010.

[64] Shunzhuo Wang, Fei Wu, Zhonghai Lu, You Zhou, Qin Xiong, Meng Zhang, and Changsheng Xie. Lifetime adaptive ecc in nand flash page management. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1253–1556. IEEE, 2017.

[65] Qingsong Wei, Bozhao Gong, Suraj Pathak, Bharadwaj Veeravalli, LingFang Zeng, and Kanzo Okada. WAFTL: A workload adaptive flash translation layer with data partition. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–12. IEEE, 2011.

[66] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. ANViL: Advanced virtualization for modern non-volatile memory devices. In *FAST*, pages 111–118, 2015.

[67] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 230–243. ACM, 2001.

[68] Norbert Werner, Guillermo Payá-Vayá, and Holger Blume. Case study: Using the xtensa lx4 configurable processor for hearing aid applications. *Proceedings of the ICT. OPEN*, 2013.

[69] ONFI Workgroup. Open NAND flash interface specification revision 3.0. *ONFI Workgroup, Published Mar*, 15:288, 2011.

[70] 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*, pages 253–266. ACM, 2012.

[71] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 6. ACM, 2015.

[72] Jie Zhang, Gieseo Park, Mustafa M Shihab, David Donofrio, John Shalf, and Myoungsoo Jung. Open-NVM: An open-sourced fpga-based nvm controller for low level memory characterization. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 666–673. IEEE, 2015.

[73] Jie Zhang, Mustafa Shihab, and Myoungsoo Jung. Power, energy, and thermal considerations in SSD-based I/O acceleration. In *HotStorage*, 2014.

[74] Yiying Zhang, Gokul Soundararajan, Mark W Storer, Lakshmi N Bairavasundaram, Sethuraman Subbiah, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau.

Warming up storage-level caches with bonfire. In *FAST*, pages 59–72, 2013.

[75] Da Zheng, Randal Burns, and Alexander S Szalay. Toward millions of file system iops on low-cost, commodity hardware. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 69. ACM, 2013.

[76] You Zhou, Fei Wu, Ping Huang, Xubin He, Changsheng Xie, and Jian Zhou. An efficient page-level FTL to optimize address translation in flash memory. In *Proceedings of the Tenth European Conference on Computer Systems*, page 12. ACM, 2015.

A Study of SSD Reliability in Large Scale Enterprise Storage Deployments

Stathis Maneas
University of Toronto

Kaveh Mahdaviani
University of Toronto

Tim Emami
NetApp

Bianca Schroeder
University of Toronto

Abstract

This paper presents the first large-scale field study of NAND-based SSDs in *enterprise* storage systems (in contrast to drives in distributed data center storage systems). The study is based on a very comprehensive set of field data, covering 1.4 million SSDs of a major storage vendor (NetApp). The drives comprise three different manufacturers, 18 different models, 12 different capacities, and all major flash technologies (SLC, cMLC, eMLC, 3D-TLC). The data allows us to study a large number of factors that were not studied in previous works, including the effect of firmware versions, the reliability of TLC NAND, and correlations between drives within a RAID system. This paper presents our analysis, along with a number of practical implications derived from it.

1 Introduction

System reliability is arguably one of the most important aspects of a storage system, and as such a large body of work exists on the topic of storage device reliability. Much of the older work is focused on hard disk drives (HDDs) [2, 26–28], but as more data is being stored on solid state drives (SSDs), the focus has recently shifted to the reliability of SSDs. In addition to a large amount of work on SSDs in lab conditions under controlled experiments [3, 5–11, 13, 18–21, 31, 32, 36], more recently, the first field studies reporting on SSD reliability in deployed production systems have appeared [22, 23, 29, 34]. These studies are based on data collected at data centers at Facebook, Microsoft, Google, and Alibaba, where drives are deployed as part of large distributed storage systems. However, we observe that there still are a number of critical gaps in the existing literature that this work is striving to bridge:

• There are no studies that focus on *enterprise storage systems*. The drives, workloads, and reliability mechanisms in these systems can differ significantly from those in cloud data centers. For example, the drives used in enterprise storage systems include high-end drives and reliability is ensured through (single, double or triple parity) RAID, instead of replication or distributed storage codes.

• We also observe that existing studies do not cover some of the most important characteristics of failures that are required for building realistic failure models, in order to compute metrics such as the mean time to data loss. This includes, for example, a breakdown of the reasons for drive replacements, including the scope of the underlying problem and the corresponding repair action (RAID reconstruction versus draining the drive), and most importantly, an understanding of the correlations between drives in the same RAID group.

In this paper, we work toward closing these gaps and provide the first field study of a large population of SSDs deployed in NetApp's *enterprise storage systems*. Our study is based on telemetry data for a sample of the total NetApp SSD population over a period of 30 months. Specifically, our study's SSD population comprises of almost 1.4 million drives and includes drives from three different manufacturers, 18 different models, 12 different capacities, and four different flash technologies, i.e., SLC, cMLC (*consumer-class*), eMLC (*enterprise-class*), and 3D-TLC. The data collected for these drives is very rich, and includes information on drive replacements (including reasons for replacements), bad blocks, usage, drive age, firmware versions, drive role (e.g., data, parity or spare), among a number of other things. This paper presents the results from our analysis with a focus to close the gaps in existing work.

2 Background

2.1 Description of the Systems

The basis of our study is telemetry data from a large population of NetApp storage systems deployed in the field. The systems, also referred to as *filers*, employ the WAFL file system [17] and NetApp's Data ONTAP operating system [24], which uses software RAID to provide resiliency against drive failures. The RAID subsystem can be configured to use SSDs in a Raid-TEC [16] (triple Parity), RAID-DP [12] (double Parity), or RAID-4 [25] (single Parity) configuration. The SSDs within a RAID group are homogeneous (same manufacturer, model, and capacity); the drives' deployment time can vary

(from a few months to several years), but most SSDs within a RAID group were deployed at the same time. Systems run on custom Fabric-Attached Storage (**FAS**) hardware and use drives manufactured by other companies. They serve data over the network using file-based protocols such as NFS and CIFS/SMB, and/or block-based protocols, such as iSCSI.

Filers vary widely in their hardware configurations, in terms of CPU, memory, and number of drives. They are divided into two groups: one that uses SSDs as an intermediate write-back caching layer on top of HDDs, and another consisting of flash-only systems (called All Flash FAS (**AFF**)).

2.2 Description of the Data

The majority of all NetApp systems in the field send weekly NetApp Active IQ® bundles (previously called AutoSupport), which track a very large set of system and device parameters, but do not contain copies of the customers' actual data. This information is collected and automatically analyzed for corrective action and for detecting potential issues.

Our study is based on mining this collection of NetApp Active IQ messages. More precisely, our data set consists of 10 snapshots, each of which is based on parsing the entire body of NetApp Active IQ support messages at 10 different points in time: Jan/Jun 2017, Jan/May/Aug/Dec 2018, and Feb/Mar/April/May 2019. Each snapshot contains monitoring data for every filer (and its drives) and consists of all those NetApp Active IQ messages that were collected before the end of the corresponding month. Moreover, the data set provides information on filers and their configuration, including information on its different RAID groups and the role of a drive within a RAID group (data, parity, or hot spare drive).

Finally, a separate data set contains an entry for each drive that was marked as *failed* during the course of our study. These drives are being *replaced* (typically by a hot spare) and sent for offline testing and diagnosis. In the remainder of the paper, we use the terms replacement and failure interchangeably. The data set also contains a reason *type* for the majority of SSD replacements, explaining why the drive was replaced.

3 Summary Statistics

In this section, we present baseline statistics on the characteristics of the drives in our population and summary statistics on various reliability metrics.

3.1 Drive characteristics and usage

The first six columns in Table 1 describe the key characteristics of the different drive families in the SSD population, including manufacturer and model (in anonymized form), capacity (ranging from 200GB to 15.3TB), interface (SAS versus SATA), flash technology (SLC, cMLC, eMLC, 3D-TLC), lithography and the model's program-erase (PE) cycle limit, i.e., the maximum number of PE cycles it is rated for (ranging from 10K to 100K). Each drive family contains a few thousand to hundred thousand SSDs. Finally, as shown in

Table 1, the population spans a large number of configurations that have been common in practice over the last years.

The next four columns in the table present some summary statistics on how the different drive models have been used, including the over-provisioning (OP) factor (i.e., what fraction of the drive is reserved as spare capacity mostly to enable drive-internal garbage collection), the date when the first drive of this model was deployed, the median number of years drives have been powered on, and the mean and median fraction of the drives' rated life that has been used (i.e., the number of PE cycles the drive has experienced as a percentage of its PE cycle limit, as reported by the drive).

3.2 Health metrics

The last three columns in Table 1 provide statistics on three different drive health and reliability metrics. Specifically:
• *Percentage of Spare Blocks Consumed:* Each drive reserves an area (which is equal to 2.5% of the total drive capacity for the SSDs in this study) for remapping the contents of blocks that the drive internally declares as *bad*, e.g., due to an excessive error count. The *Percentage of Spare Blocks Consumed* metric reports what percentage of this area has been consumed (population mean and median).
• *Number of Bad Sectors:* Data ONTAP keeps track of a drive's defect list, known as *g-list*. This list is populated with a new entry every time the operating system receives an *unrecoverable error* for a block. The mean and median length of this list are reported as the *Number of Bad Sectors*.
• *Annual Replacement Rate (**ARR**):* We make use of the common *Annual Replacement Rate* metric, defined as number of device failures divided by numbers of device years.

3.3 High-level observations

Below, we make a number of first observations based on Table 1, before we delve into a more detailed analysis of our data set in the remainder of this paper:
• The average ARR across the entire population is 0.22%, but rates vary widely depending on the drive model, from as little as 0.07% to nearly 1.2%. These numbers are significantly lower than numbers previously reported for data center drives. For example, even the worst model in our study (ARR of 1.2%) is at the low end of the range reported for SSDs in Google's data centers (range of 1-2.5% [29]). The rates are also significantly lower than common numbers reported for hard disk drives (i.e., 2-9% [26, 28]).
• Even for drive models that are very similar in their technical specifications (e.g., same manufacturer, flash technology, capacity, age), ARR can vary dramatically, e.g., 0.53% for II-G 15TB drives versus 1.13% for II-C 15.3TB drives.
• The spare area reserved for bad blocks is generously provisioned for the typical drive: even for drives that have been in the field for several years, the percentage of consumed spare blocks is on average less than 15%. Even the drives in the 99th and 99.9th percentile of consumed spare blocks have

Drive characteristics						Usage characteristics				Reliability metrics		
Manu-fact./Model	Cap. (GB)	Inter-face	Flash Tech.	Lith.	PE Cycl.	OP	First Deploy-ment	Drive Power Years	Rated Life Used (%)	% of Sp. Blocks Cons.	Number of Bad Sectors	ARR (%) (all)
I - A	200	SAS	eMLC	2xnm	10K	28%	Apr '14	3.95	1.26 / 0	1.36 / 1	0.58 / 0	0.19
	400						Apr '14	3.93	0.52 / 0	1.54 / 1	2.96 / 0	0.16
	800						Mar '14	3.19	0.07 / 0	1.22 / 1	3.39 / 0	0.15
	1600						Mar '14	3.74	0.01 / 0	1.46 / 1	6.2 / 0	0.21
I - B	400	SAS	eMLC	1xnm	10K	28%	Dec '15	2.69	3.99 / 3	2.49 / 2	0.9 / 0	0.14
	800						Jan '16	2.58	1.87 / 2	3.7 / 4	4.62 / 0	0.17
	1600						Jan '16	2.55	1.68 / 2	3.55 / 3	6.01 / 0	0.23
I - C	400	SAS	eMLC	1xnm	10K	28%	Jan '17	1.7	7.48 / 7	1.83 / 2	1.86 / 0	0.12
	800						Jan '17	1.45	6.01 / 6	4.21 / 4	5.58 / 0	0.27
	1600						Mar '17	1.53	5.62 / 5	4.19 / 4	6.41 / 0	0.09
	3800						Jan '17	1.12	5.15 / 4	9.86 / 10	14.97 / 0	0.59
I - D	3800	SAS	eMLC	1xnm	10K	28%	Jul '17	1.01	4.01 / 3	10.01 / 10	12.34 / 0	0.23
II - A	3840	SAS	3D-TLC	V2	10K	7%	Dec '15	2.57	0.09 / 0	11.88 / 12	0.03 / 0	0.31
II - B	3800	SAS	3D-TLC	V2	10K	7%	Oct '16	1.77	0.01 / 0	12.01 / 12	0.36 / 0	0.13
II - C	8000	SAS	3D-TLC	V3	10K	7%	Sep '17	1.06	0.01 / 0	12.38 / 12	0.03 / 0	0.69
	15300						Sep '16	1.21	0.06 / 0	13.13 / 13	1.35 / 0	1.13
II - D	960	SAS	3D-TLC	V3	10K	7%	Oct '16	1.54	0.06 / 0	16.05 / 15	0.03 / 0	0.12
	3800						Oct '16	1.8	0.01 / 0	12	0.34 / 0	0.11
II - E	400	SAS	3D-TLC	V3	10K	28%	Dec '16	2.12	0.59 / 0	19.19 / 15	0.01 / 0	0.09
	800					28%	Jan '17	1.7	0.08 / 0	15.33 / 15	0 / 0	0.10
	3800					7%	Dec '16	2.02	0.05 / 0	12 / 12	0.26 / 0	0.13
II - F	400	SAS	3D-TLC	V2	10K	28%	Jan '16	2.52	0.86 / 0	12.31 / 12	0.01 / 0	0.48
	800						Feb '16	2.55	0.19 / 0	12.19 / 12	0.01 / 0	0.36
	1600						Jan '16	2.87	0.09 / 0	11.66 / 12	0.15 / 0	0.52
II - G	800	SAS	3D-TLC	V2	10K	28%	Apr '18	0.38	0.03 / 0	0 / 0	0 / 0	0.18
	960						Jan '18	0.5	0.11 / 0	0 / 0	0.03 / 0	0.18
	3800						Jan '18	2.89	0.09 / 0	11.64 / 12	0.15 / 0	0.28
	8000						May '18	0.45	0 / 0	0 / 0	0.25 / 0	0.37
	15000						May '18	0.46	0 / 0	0 / 0	0 / 0	0.53
II - H	800	SAS	cMLC	1xnm	10K	28%	Nov '14	3.61	1.34 / 0	7.49 / 7	1.36 / 0	0.10
II - I	200	SAS	eMLC	2xnm	30K	28%	Aug '09	4.83	5.24 / 2	6.7 / 6	0.31 / 0	0.07
	400						Dec '10	3.86	2.2 / 0	8.09 / 8	0.29 / 0	0.07
	800						Dec '10	4.66	0.69 / 0	6.94 / 7	4.53 / 0	0.11
II - J	400	SAS	eMLC	1xnm	10K	28%	May '15	3.37	2.32 / 0	6.9 / 7	0.08 / 0	0.18
	800						Jul '15	3.21	0.41 / 0	6.77 / 7	0.36 / 0	0.21
	1600						Jun '15	3.36	0.13 / 0	8.59 / 9	0.49 / 0	0.38
II - K	100	SATA	SLC	4xnm	100K	28%	Apr '12	0.43	1.4 / 1	3.78 / 4	3.62 / 0	0.14
II - L	100	SATA	SLC	3xnm	100K	28%	May '10	5.97	2.05 / 1	3.73 / 4	0.51 / 0	0.06
II - M	100	SATA	SLC	5xnm	100K	28%	Jul '10	4.23	1.63 / 1	14.82 / 13	0.18 / 0	0.11
III - A	200	SAS	eMLC	2xnm	30K	28%	Dec '12	5.89	3.08 / 1	0.02 / 0	6.18 / 0	0.07

Table 1: *Summary statistics describing our population of drives. Whenever a column includes two values (separated by "/"), these correspond to the mean and median values of that population, respectively.*

consumed only 17% and 33% of their spare blocks.

• The typical drive remains far from ever reaching its PE cycle limit. Even for models where most drives have been in the field for 2-3 years, less than two percent of the rated life is consumed on average. Even the drives in the 99th and 99.9th percentile of rated life used have consumed only 15% and 33% of their rated life, respectively. Hence, for the vast majority of drives, early death due to wear-out after prematurely reaching the PE cycle limit is unlikely.

4 Reasons for replacements

There are different reasons that can trigger the replacement of an SSD and also different sub-systems in the storage hierarchy which can detect issues that trigger the replacement of drives. For example, issues might be reported by the drive itself, the storage layer, or the file system. Table 2 describes the different reason *types* that can trigger a drive replacement, along with their frequency, the recovery action taken by the system (i.e., copying out data from the drive to be replaced versus reconstructing the data using RAID parities), and the scope of the problem (i.e., risk of partial data loss, risk of complete drive loss, or no immediate problems). In our data set, the reason type is missing for 40% of all replacement events due to issues with the data collection pipeline. These issues are not related to the actual reason for the replacements. Hence, we can assume replacements with a missing reason type to be proportionately spread over the remaining categories. Therefore, the frequency of each replacement type is normalized to account for the missing data. We group the different reason types behind SSD replacements into four *categories*, labelled

Cate-gory	Type	Pct.	ARR (%)	Description	Recovery Action	Scope
A	**SCSI Error**	32.78%	0.055	The SCSI layer detects a hardware error reported by the SSD, that is severe enough that *immediate* replacement of the drive and reconstruction of the data is triggered. For example, these errors could be due to ECC errors originating from the drive's DRAM that prevent it from functioning properly.	RAID Reconstr.	Full
	Unresponsive Drive	0.60%	0.001	The drive has completely *failed* and become unresponsive.		
B	**Lost Writes**	13.54%	0.023	A lost write is detected when the contents of a 4K WAFL block (read from the SSD) are inconsistent based on its *signature*, which includes attributes and version number. Since there are many potential causes with the same symptom, a heuristic is used to decide whether to fail the disk or not. If multiple such errors occur within one SSD and no errors within any other SSD, then the former SSD is marked as *failed*.	RAID Reconstr.	Partial
C	**Aborted Commands**	13.56%	0.023	This error is generated due to an aborted command and is reported either by the SSD itself or the Storage Layer. For instance, this error can occur when the host sends some write commands to the device, but the actual data never reach the device due to an issue on the host or due to connection issues.	RAID Reconstr.	Partial
	Disk Ownership I/O Errors	3.27%	0.005	This error is related to the sub-system responsible that keeps track of which node owns a disk. In case an error occurs during the communication with this sub-system, then the SSD is immediately marked as *failed*.		
	Command Timeouts	1.81%	0.003	SSDs internally keep track of timers and also the Storage Layer maintains its own timers for every command sent to each SSD. This error indicates that the operation could not be completed within the allotted time even after retries.		
D	**Predictive Failures**	12.78%	0.021	The SSD reports this error based on a pattern of recovered errors that have occurred internally using its own thresholds and criteria, as specified by the corresponding manufacturer.	Disk Copy	Zero
	Threshold Exceeded	12.73%	0.020	The Storage Health Monitor sub-system keeps track of different parameters for each SSD and in case a threshold (e.g., on the number of media errors) is exceeded, the SSD is proactively replaced.		
	Recommended Failures	8.93%	0.015	This error is reported by the system and indicates that the drive should be replaced in the near future. This failure type is less strict and less urgent than Threshold Exceeded failures.		

Table 2: *Description of reason types that can trigger a drive replacement. Disk copy operations are performed only where possible, i.e., a spare disk must be available; otherwise, the data of the replaced drive is constructed via RAID reconstruction.*

from *A* to *D*, based on their severity.

The most benign[1] category is category D, which relates to replacements that were triggered by logic either inside the drive or at higher levels in the system, which predicts future drive failure, for example based on previous errors, timeouts, and a drive's SMART statistics [33].

The most severe category is category A, which comprises those situations where drives become completely unresponsive, or where the SCSI layer detects a drive problem severe enough to trigger immediate replacement of the drive and RAID reconstruction of the data stored in it.

Category B refers to drive replacements that are taking place when the system suspects the drive to have *lost a write*, e.g., because it did not perform the write at all, wrote it to a wrong location, or otherwise corrupted the write. The root cause could be a firmware bug in the drive, although other layers in the storage stack could be responsible as well. As there are many potential causes, a heuristic is used to decide whether to trigger a replacement or not; specifically, if multiple such errors occur within one SSD and no errors within any other SSD, then the former SSD will be replaced.

Finally, in category C most of its reasons for replacements are related to commands that were aborted or timed out.

---

[1] We call them "benign" as the drive was still operational before getting replaced. Also, recovery is minimal (disk copy versus RAID reconstruction).

When examining the frequency at which individual replacement reason types are reported, we observe that the single most common reason type are SCSI errors, which are responsible for ∼33% of all replacements and are unfortunately also one of the most severe reason types. The other severe reason for drive replacements, i.e., a drive becoming completely unresponsive, is reported for only 0.60% of all replacements.

Fortunately, one third of all drive replacements are merely preventative (category D) using predictions of future drive failures and are hence unlikely to have severe impact on system reliability. A detailed investigation of predictive replacements (not covered in the table due to space reasons) reveals that the most common trigger behind a preventative replacement is exceeding the threshold of consecutive timeouts.

The two remaining categories are roughly equally common and both have the potential of partial data loss if RAID reconstruction of the affected data should turn out unsuccessful. The first category (C) refers to aborted and timed out commands, and makes up ∼19% of all reason types. The other category (B) refers to lost writes. This is an interesting category, since it is somewhat less clear whether it is the drive or other layers in the stack that are to blame for the lost write.

We will come back to the different reason types for replacements at various places in the remainder of the paper, when we will, for example, consider how the frequency of different

reason types behind replacements varies depending on drive capacity, lithography, age, or firmware version.

Finding 1: One third of replacements are associated with one of the most severe reason types (i.e., SCSI errors), but on the other hand, one third of drive replacements are merely preventative based on predictions.

5 Factors impacting replacement rates

In this section, we evaluate how different factors impact the *annual replacement rate* of the SSDs in our data set. We conduct our analysis on eMLC and 3D-TLC SSDs, and exclude cMLC and SLC drives due to insufficient data.

5.1 Usage and Age

Usage, and the wear-out of flash cells that comes with it, is well known to affect the reliability of flash-based SSDs; drives are guaranteed to remain functional for only a certain number of PE cycles. In our data set, SLC drives have a PE cycles limit of 100K, whereas the limit of most cMLC, eMLC, and 3D-TLC drives is equal to 10K cycles, with the exception of a few eMLC drive families with a 30K PE cycles limit.

Each drive reports the number of PE cycles it has experienced as a percentage of its PE cycle limit (the *"rated life used"* metric, recall Section 3.1), allowing us to study how usage affects replacement rates. Unfortunately, the rated life used is only reported as a truncated integer and a significant fraction of drives report a zero for this metric, indicating less than 1% of their rated life has been used. Therefore, our first step is a comparison of the ARR of drives that report less than 1% versus more than 1% of their rated life used. The results for eMLC and 3D-TLC drives are shown in Figure 1, which includes both overall replacement rates ("All"), and rates broken down by their replacement category (A to D). Throughout our paper, error bars refer to 95th percentile confidence intervals and we exclude two outlier models, i.e., II-C and I-C, with unusually high replacement rates to not obscure trends (except for graphs involving individual drive families).

We also perform statistical tests and calculate *p-values* to confirm our hypotheses (where applicable). For each test case, we perform a two-sample *z-test* [1]. Since our analysis is based on replacement rates, we need to calculate and compare the replacement rates of the two groups in each test. For each group, we create 1,000 random samples of replacement rates; in each sample, the replacement rate is measured based on a randomly chosen set of 1,000 SSDs from the corresponding group. Finally, we perform a z-test on the two sets of samples and report the calculated *p-value* associated with the test.

Figure 1 provides evidence for effects of infant mortality. For example, for eMLC drives, the drives with less than 1% rated life used are more likely (1.25X) to be replaced than those with more than 1% of rated life used (the estimated mean replacement rates of the two populations are 0.168 and 0.126 respectively, whereas the corresponding p-value is equal to 6.3211e-45). When further breaking results down by
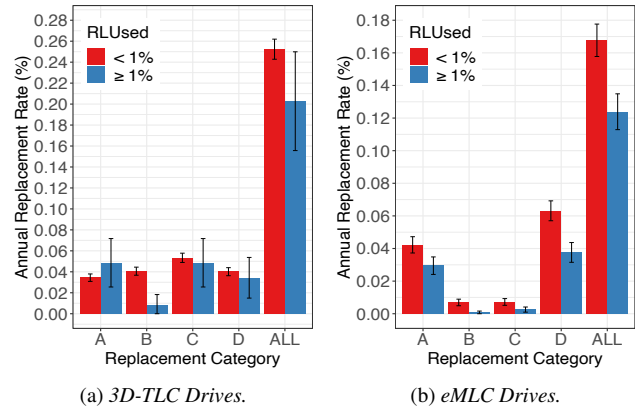

(a) *3D-TLC Drives.*   (b) *eMLC Drives.*

Figure 1: *Annual replacement rate per flash type based on the drives' "rated-life-used" percentage.*

reason category, we find that drives with less usage consistently experience higher replacement rates for all categories. Making conclusive claims for the 3D-TLC drives is harder due to limited data on drives above 1% of rated life used, resulting in wide confidence intervals. However, where we have enough data, observations are similar to those for eMLC drives, e.g., we see a significant drop in lost writes for drives above 1% of rated life used.

We also looked separately at drives that are extensively used (more than 50% of their PE cycles) and their typical reasons for replacement. We see the trend of decreasing rates of lost writes continues here, as we don't observe a single case related to *lost writes* among these drives. One possible explanation is that lost writes might be related to firmware bugs, and as firmware gets updated to improved versions over the course of a drive's life, rate of lost writes drops. We take a closer look at firmware versions in Section 5.5. It's also possible that issues leading to lost writes typically become evident early in a drive's life and the drive gets replaced before it makes it to more than 1% or 50% of its rated life.

Another interesting observation is that the heavily used drives are more likely to be replaced due to *predictive failures* compared to the overall population. This could mean that issues leading to predictive failures are only exposed after some significant usage (e.g., hardware problems that cause media errors and bad blocks, which then trigger failure prediction, require thoroughly exercising the NAND). Alternatively, it could mean that after more drive usage more data is available on the drive's health status, which improves predictions.

We also look at replacement rates as function of a drive's age measured by its total months in the field. Figure 2 (top) shows the conditional probability of a drive being replaced in a given month of its life. i.e., the probability that the drive will fail in month *x* given that it has survived month *x*-1.

We observe an unexpectedly long period of infant mortality with a shape that differs from the common "bathtub" model often used in reliability theory. The bathtub model assumes a short initial period of high failure rates, which then quickly

drops [14, 15, 28, 35]. Instead, we observe for both 3D-TLC and eMLC drives, a long period (12–15 months) of increasing failure rates, followed by a lengthy period (another 6–12 months) of slowly decreasing failure rates, before rates finally stabilize. That means that, given typical drive lifetimes of 5 years, drives spend 20-40% of their life in infant mortality.

We wondered whether these unexpected results might just be an artifact of our heterogeneous population, since each line in Figure 2 (top) is computed over a population comprising different drive families with different drive ages and characteristics. We therefore plotted in Figure 2 (bottom) the same probabilities, but this time only over a subset of drive families with similar characteristics (e.g., age and lithography). Again, we observe the same trends, and in fact in some aspects even slightly more pronounced: the duration of the two phases is similar in length and for 3D-TLC drives, the ratio of the peak failure rate to the lowest rate is even larger (a factor of 2.5X).

It might be surprising at first that we do not observe an increase in ARR for drives towards the end of their life. The reason is that the majority of drives, even those deployed for several years, do not experience a large number of PE cycles. Their fraction even in the population of older drives is too small to drive up the overall ARR.

Finding 2: We observe a very drawn-out period of infant mortality, which can last more than a year and see failure rates 2-3X larger than later in life.

5.2 Flash and drive type

The drive models in our study differ in the type of flash they are based on, i.e., in how many bits are encoded in a single flash cell. For instance, Single Level Cell (SLC) drives encode only one bit per cell, while Multi-Level Cell (MLC) drives encode two bits in one cell for higher data density and thus a lower total cost, but potentially higher propensity to errors. The most recent generation of flash is based on Triple Level Cell (3D-TLC) flash with three bits per cell.

The last column in Table 1 allows a comparison of ARRs across flash types. A cursory study of the numbers indicates generally higher replacement rates for 3D-TLC devices compared to the other flash types. Also, we observe that 3D-TLC drives have consumed 10-15X more of their spare blocks.

For a more nuanced comparison between 3D-TLC and eMLC we turn to Figures 1 and 4, which also take usage and lithography into account. Figure 1 indicates that ARRs for 3D-TLC drives are around 1.5X higher than for eMLC drives, when comparing similar levels of usage. Figure 4 paints a more complex picture. While V2 3D-TLC drives have a significantly higher replacement rate than any of the other groups, the V3 3D-TLC drives are actually comparable to 2xnm eMLC drives, and in fact have lower ARR than the 1xnm eMLC drives. So, lithography might play a larger role than flash type alone (we take a closer look at in Section 5.4).

We are also interested in differences between the enterprise-class eMLC drives and consumer-class cMLC drives. Unfor-
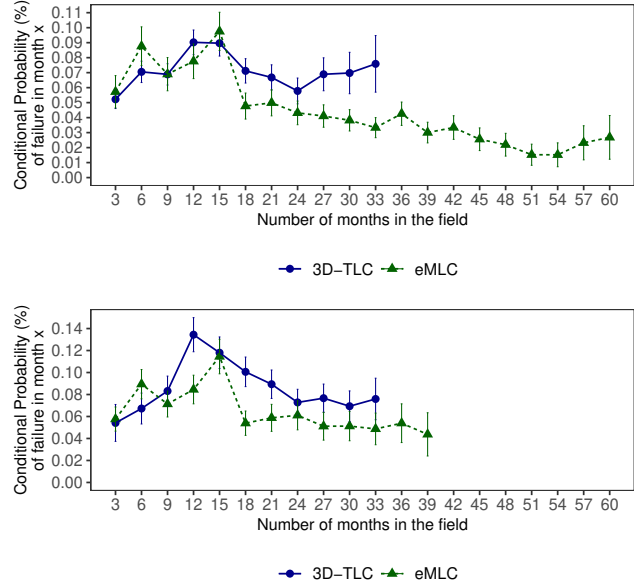

Figure 2: *Conditional probability of failure based on a drive's age (number of months in the field) for all drive families (top) and a subset of them (bottom), i.e., II-A, II-B, and II-F for 3D-TLC drives and I-B, I-C, I-D, and II-J for eMLC drives.*

tunately our data set contains only one family of cMLC drives (II-H). Interestingly, we find that this one family of cMLC drives reports much lower replacement rates than eMLC families of similar age and capacity (i.e., II-H drives vs II-J drives). Narayanan et al. [23] report replacement rates between 0.5-1% for their *consumer* class MLC drives, with the exception of a single *enterprise* class model, whose replacement rate is equal to 0.1%; however, the authors in [23] consider only fail-stop failures. In our study, we consider different types of failures and thus, the reported replacement rates would have been even smaller had we considered only fail-stop failures.

Finally, we observe that SLC models are not generally more reliable than eMLC models that are comparable in age and capacity. For example, when we look at the ARR column of Table 1, we observe that SLC models have similar replacement rates to two eMLC models with comparable capacities, i.e., II-I and III-A drives (their difference is small but still statistically significant, i.e., the estimated mean replacement rates of the two populations are 0.112 and 0.091 respectively, with a p-value equal to 5.0841e-22). This is consistent with the results in a field study based on drives in Google's data centers [29], which does not find SLC drives to have consistently lower replacement rates than MLC drives either. Considering that the lithography between SLC and MLC drives can be identical, their main difference is the way cells are programmed internally, suggesting that controller reliability can be a dominant factor.

Finding 3: Overall, the highest replacement rates in our study are associated with 3D-TLC SSDs. However, no single flash type has noticeably higher replacement rates than the
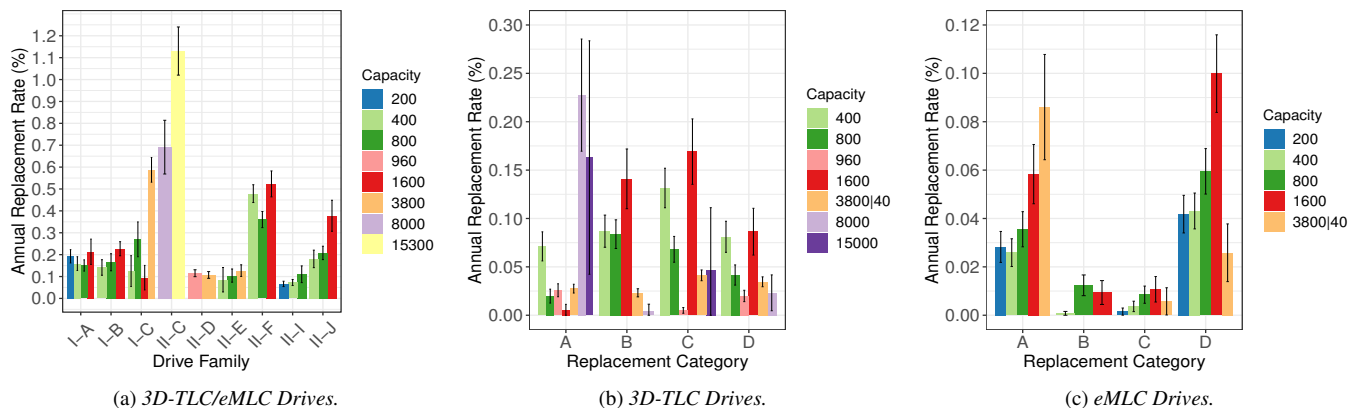
(a) *3D-TLC/eMLC Drives.* (b) *3D-TLC Drives.* (c) *eMLC Drives.*

Figure 3: *Figure 3a shows the annual replacement rates for the drive families shipped with multiple capacities. Figures 3b and 3c show replacement rates for different capacities broken down by their replacement category, for 3D-TLC drives and eMLC drives, respectively. In these figures, the 3800GB and 3840GB capacities have been consolidated.*

other flash types studied in this work, indicating that other factors, such as capacity or lithography, can have a bigger impact on reliability.

5.3 Capacity

The drives in our data set range in capacity from 100GB to 15.3TB and most drive families include drives of different capacities, which allows us to study the effect of drive capacity on replacement rates. One would expect the rate of failures that are due to underlying hardware issues (such as failure of NAND cells or DRAM) would grow with capacity. On the other hand, failures that are related to firmware bugs are not likely to be strongly correlated with capacity (all else being equal). In the remainder of this section, we test our hypothesis of NAND related problems increasing with capacity.

First, we turn to the reported numbers on bad sectors in Table 1. We observe that consistently within each drive family, the total number of bad sectors continuously increases with capacity. For example, for model I-C, the average number of bad sectors per drive is growing from 1.9 to 5.6 to 6.4 to 14.97 for capacities of 400, 800, 1600 and 3800 GB, respectively. Moreover, the percentage of drives with a non-zero count of bad blocks continuously increases with capacity.

Figure 3a explores how overall ARRs change with capacity by plotting the ARR of different drive families, broken down by capacity. We make a slightly more nuanced observation for ARR, compared to the bad sector count. For smaller capacities, in the range of 200 to 1600 GB, the ARR shows no clear relationship with capacity. It might be that for these smaller capacities replacements are dominated by reasons other than issues with the underlying NAND. The trend starts to change around 1600GB, as for four out of the five families that have 1600GB drives, those drives have the highest ARR. And for larger capacities, there is a clear trend for increasing ARRs. The 15TB drives always have higher ARR than the other drives in the same family. The 3800GB and 8000GB

drives always have higher ARR than the drives less than 3800GB within the same family.

We also looked for differences in the reasons for replacement between smaller and larger capacity drives and made an interesting observation: for the largest capacity drives, the rate of predictive failures is lower than for smaller capacity drives. In contrast, the most severe failure reason, i.e., an unresponsive drive, occurs at a much higher rate for the larger capacity drives than for the smaller capacity drives. Figures 3b and 3c illustrate this observation, as they break down the ARR by capacity and replacement category for different flash technologies. Among the eMLC drives, the 3800GB and 3840GB capacities and among the 3D-TLC drives, the 8TB and 15TB capacities have very high rates of replacement due to an unresponsive drive, compared to smaller capacities. They also have a lower rate of replacements due to predictive failures. This means that the replacement rate associated with high capacity drives is not only bigger, but also has potentially more severe consequences. Another potential implication is that failures of large capacity drives are either harder to predict or the prediction algorithms have not been optimized for them. It may be possible that the severe failures and unpredictability of such failures is an artifact of the larger DRAM footprint associated with large flash capacity, rather than the flash capacity itself. Potential for such impact could be mitigated by upcoming architectures such as Zoned Storage (ZNS) [4, 30] that obviate the need for large Flash Translation Layer (FTL) tables in DRAM and consequently reducing the DRAM footprint.

Finding 4: *Drives with very large capacities not only see a higher replacement rate overall, but also see more severe failures and fewer of the (more benign) predictive failures.*

5.4 Lithography

Lithography has been shown to be highly correlated with a drive's raw bit error rate (RBER); models with smaller lithog-
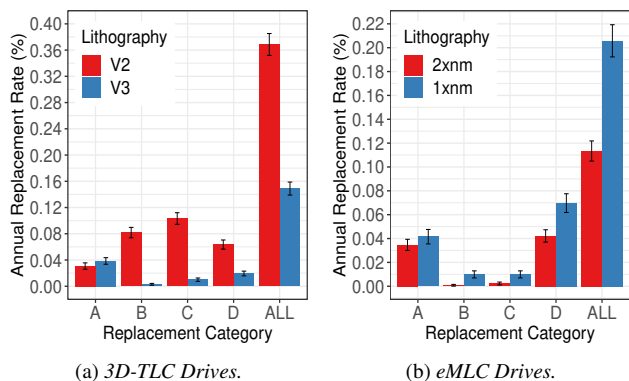
(a) *3D-TLC Drives.* (b) *eMLC Drives.*

Figure 4: *Annual replacement rate per flash type and lithography broken down by replacement category.*

Figure 5: *Annual replacement rates per firmware version.*

raphy report higher RBERs according to a study based on data center drives [29], but not necessarily higher replacement rates. We explore what these trends look like for the drives in enterprise storage systems. To separate the effect of lithography from flash type (i.e., SLC, cMLC, eMLC, 3D-TLC), we perform the analysis separately for each flash type.

The bar graph in Figure 4 (right) shows the ARR for eMLC drives separated into 2xnm and 1xnm lithographies broken down by failure category, also including one bar for replacements of all categories. We observe that the higher density 1xnm drives experience almost twice the replacement rate of 2xnm drives (the p-value is equal to 4.2365e-120). Also, replacement rates for each of the individual reason categories are higher for 1xnm drives than for 2xnm, with the only exception of reason category A, which corresponds to unresponsive drives. Finally, we also observe that the 1xnm drives also have, on average, consumed a larger percentage of spare blocks (an indicator of developing bad blocks) and developed a larger number of bad sectors, despite the fact that they are generally younger than the 2xnm drives.

In contrast to eMLC drives, the 3D-TLC drives see higher replacement rates for the lower density V2 drives, which internally have fewer layers than V3 (the corresponding z-test returns a p-value equal to 2.7624e-275). When breaking replacement rates down by reason category, we observe that consistently with the results for eMLC drives, the only category that is not affected by lithography is category A, which corresponds to unresponsive drives. Regarding the percentage of spare blocks consumed, we observe comparable values between V2 and V3 drives (if we exclude the II-G family, which is much younger than the others).

Finally, for SLC drives, we do not see a clear trend for replacement rates as a function of lithography; however, we also have limited data, with only one drive model in each lithography for SLC drives.

Finding 5: *In contrast to previous work, higher density drives do not always see higher replacement rates. In fact, we observe that, although higher density eMLC drives have higher replacement rates, this trend is reversed for 3D-TLC.*
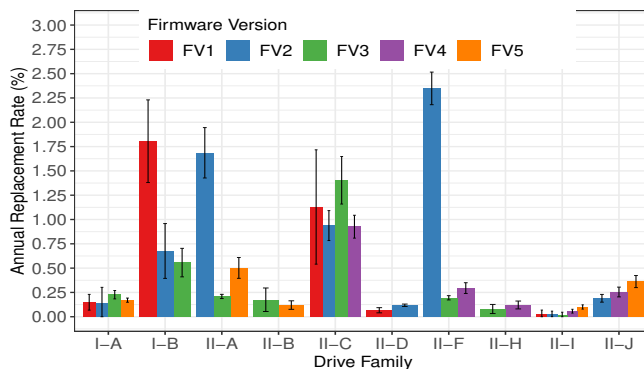
5.5 Firmware Version

Given that bugs in a drive's firmware can lead to drive errors or in the worst case to an unresponsive drive, we are interested to see whether different firmware versions are associated with a different ARR. Each drive model/family in our study experiences different firmware versions over time. We name the first firmware version of a model FV1, the next one FV2, and so on. An individual drive's firmware might be updated to a new version, but we observe that the majority of drives (70%) appear under the same firmware version in all data snapshots.

Figure 5 shows the ARR associated with different firmware versions for each drive model. Considering that firmware varies across drive families and manufacturers, it only makes sense to compare the ARR of different firmware versions within the same drive family. To avoid other confounding factors, in particular age and usage, the graph in the figure only includes drives with rated life used of less than 1% (the majority of drives). We have also analyzed the data in different ways, for example by including only drives that appear consistently under the same firmware version in all data snapshots, observing similar results.

We find that drive's firmware version can have a tremendous impact on reliability. In particular, the earliest versions can have an order of magnitude higher ARR than later versions. This effect is most notable for families I-B (more than factor 2X decrease in ARR from FV1 to FV2), II-A (factor 8X decrease from FV2 to FV3) and II-F (more than 10X decrease from FV2 to FV3). The corresponding z-tests return extremely small p-values and thus, confirm our results.

We note that the effect where earlier firmware versions have higher replacement rates persists even if we only include drives whose firmware has never changed in our data snapshots, e.g., we compare drives that spend their entire lives in FV1 and compare them to drives who only saw FV2. This provides confirmation that the effect is actually due to firmware versions, and not due to infant mortality, where the earlier version is used at an earlier time of a drive's life and the later version during a later point in life.

A likely explanation is that later firmware versions include bug fixes and improvements over earlier versions. This expla-
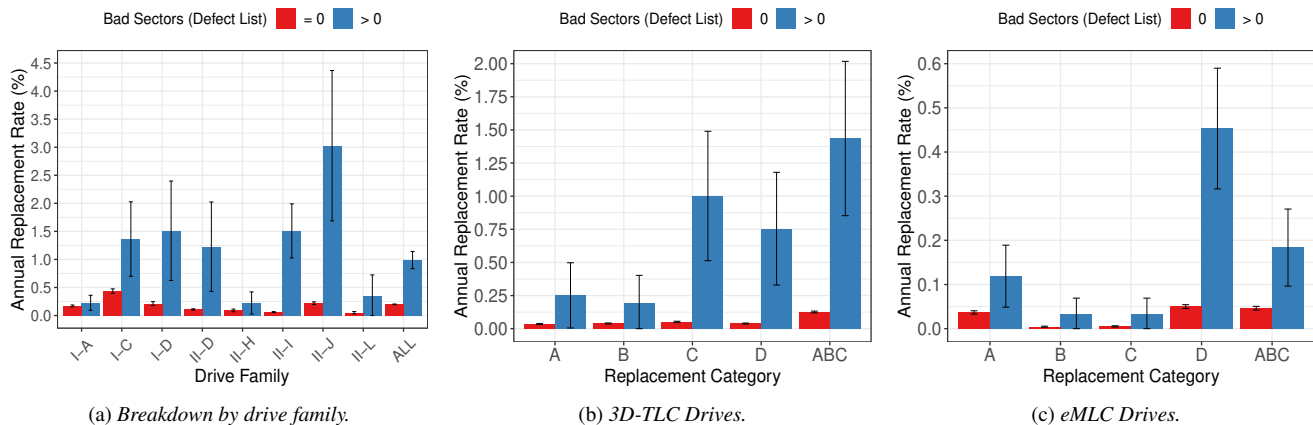
(a) *Breakdown by drive family.* (b) *3D-TLC Drives.* (c) *eMLC Drives.*

Figure 6: *Annual replacement rates per flash type based on the drives' bad sectors count. Figure 6a breaks results down by drive family and Figures 6b and 6c by replacement category.*

nation is further supported by our observation that the failures that decrease the most when moving from FV1 to later versions (e.g., for the two families with the highest decrease in ARR, II-A and II-F) are failures in categories B and C (lost writes and timeouts), both of which could be caused by firmware problems.

Interestingly, we also observe cases where the ARR increases with increasing version numbers, albeit not as frequently. One example is family II-J, where FV5 has a significantly higher ARR than FV2 or FV4. The difference between FV2 and FV5 is more than factor of 10X when considering only drives that do not change firmware version (graph omitted for lack of space). One possible explanation is that as the firmware code base evolves, it becomes more complex and the new code also introduces new bugs.

Finding 6: *Earlier firmware versions can be correlated with significantly higher replacement rates, emphasizing the importance of firmware updates.*

5.6   All Flash FAS (AFF) Systems

We also looked at whether a drive's type of usage, i.e., either as part of an AFF system or as part of a caching layer, affects its replacement rate. We find no indication that within a drive family, replacement rates vary as a function of type of usage.

5.7   Device Role

We also studied whether drives within a RAID group have different replacement rates, depending on their role in the RAID group (i.e., data and parity), but found no indication of statistically significant differences. This might indicate that WAFL is effective at balancing load across drives and minimizing the number of parity updates.

5.8   Over-provisioning

We also looked at the amount of over-provisioning (OP) as a factor, but find no clear correlation between the amount of over-provisioned space and ARR. One reason might be that the typical drive in our population is far from reaching its en-

durance limit. Therefore, the potential endurance-increasing effects of over-provisioning do not become relevant.

5.9   Number of bad blocks

In this section, we are exploring the relationship between a drive developing bad blocks and replacement rates. We consider two different metrics associated with bad blocks.

The first metric is the length of the *g-list*, also referred to as defect list, which is maintained by Data ONTAP and contains an entry for every block generated an *unrecoverable* error upon access. Since the g-list is empty for a large fraction of drives (99.04%), we distinguish between drives with an empty and a non-empty g-list, and plot their ARR separately. Figure 6a shows the results broken down by drive family.

We observe that drives that have experienced at least one unrecoverable error (i.e., they have a non-empty g-list) have significantly higher replacement rates. Part of this observation might just be an artifact of predictive drive replacements (category D), as predictions might be based on the length of the g-list. We therefore plot in Figures 6b and 6c the same rates, but broken down by replacement category.

Not surprisingly, we see that there is a strong correlation between a non-empty g-list and predictive failures (category D); however, the more interesting observation is that also for the other replacement reasons, there is a correlation between having a non-empty g-list and the drive being replaced. That means developing unrecoverable errors is indicative of a variety of future issues a drive might develop.

The second factor we consider is the number of consumed spare blocks inside each individual SSD. While we omit full results due to lack of space, we note that again we observe similar correlations.

Finding 7: *SSDs with a non-empty defect list have a higher chance of getting replaced, not only due to predictive failures, but also due to other replacement reasons as well.*

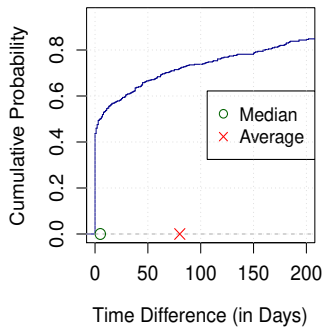**Finding 8:** *SSDs that make greater use of their over-provisioned space are quite likely to be replaced in the future.*
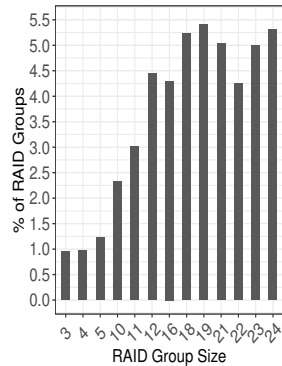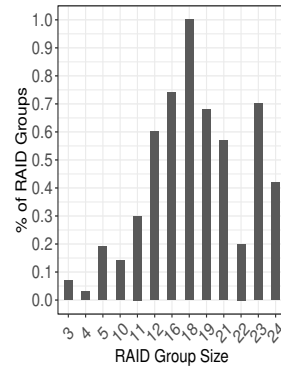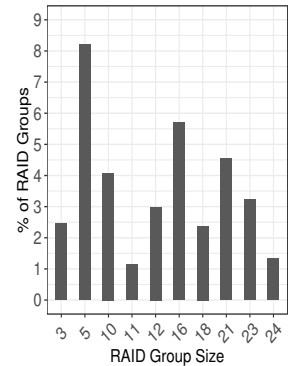
Figure 7: *Time difference between successive replacements within RAID groups.*

(a) *RAID groups that experience at least 1 replacement.*

(b) *RAID groups that experience multiple replacements.*

(c) *RAID groups with replacement that experience at least 1 follow-up replacement (within 1 week).*

Figure 8: *Statistics on replacements within RAID groups.*

6   Correlations between drive failures

A key question when deriving reliability estimates, e.g., for different RAID configurations, is how failures of drives within the same RAID group are correlated.

As a first measure of correlation, we explore the probability that a RAID group will experience a drive replacement following a prior drive replacement. More precisely, we start by computing the empirical probability that a RAID group will experience a drive replacement in a random week; this probability is equal to 0.0504%. Then, we compute the probability that a RAID group will experience another drive replacement within a week following a previous drive replacement. The probability is equal to 9.39%, that is, more than a factor of 180X increase compared to the probability that a drive replacement will occur within a random week. We speculate on a few possible reasons that could explain this. First, RAID reconstruction imposes an additional load to the other drives of the group and exposes latent errors, as these drives must be fully scanned to reconstruct the data of the failed drive. Second, shared environmental issues (e.g., overheating, power surge), could affect multiple drives from the same group simultaneously, as they are all placed within the same filer.

For a more detailed understanding of correlations, we consider all RAID groups that have experienced more than one drive replacement over the course of our observation period and plot in Figure 7, the time between consecutive drive replacements within the same RAID group. We observe that very commonly, the second drive replacement follows the preceding one within a short time interval. For example, 46% of consecutive replacements take place at most one day after the previous replacement, while 52% of all consecutive replacements take place within a week of the previous replacement.

Another important question in RAID reliability modelling is how the chance of multiple failures grows as the number of drives in the RAID group increases. Figure 8a presents, for the most common RAID group sizes, the percentage of

RAID groups of that size that experienced at least one drive replacement. As one would expect, larger RAID groups have a higher chance of experiencing a drive replacement; yet, the effect of a RAID group's size on the replacement rates saturates for RAID groups comprising more than 18 drives.

However, we make an interesting observation in Figure 8b, when we look at the percentage of RAID groups that have experienced at least two drive replacements (potential double failure): this percentage is not clearly correlated with RAID group size, except for maybe very small RAID groups of three or four drives. The largest RAID group sizes do not have a higher rate of double (or multiple) failures.

The reason becomes clear when we look at the conditional probability that a RAID group will experience a replacement, *given that it has already experienced another replacement*, in Figure 8c. More precisely, for each RAID group size, we consider the RAID groups that had at least one drive replacement and compute what percentage of them had at least one more replacement within a week. Interestingly, we observe there is no clear trend that larger RAID group sizes have a larger chance of one drive replacement being followed by more replacements. Note that, as already mentioned, the chance of experiencing a drive failure grows with the size of the RAID group (Figure 8b); however, the chance of *correlated failures* does not show a direct relationship with the group's size.

Finding 9: *While large RAID groups have a larger number of drive replacements, we find no evidence that the rate of multiple failures per group (which is what can create potential for data loss) is correlated with RAID group size. The reason seems to be that the likelihood of a follow-up failure after a first failure is not correlated with RAID group size.*

7   Related Work

Four recent field studies have looked at failure characteristics of SSDs in data centers at Facebook, Microsoft, Google, and Alibaba, respectively [22, 23, 29, 34]. Our work is different

in that we focus on enterprise storage systems, rather than distributed data center storage and is the first to report on TLC drives, large capacity drives (8 TB and 15 TB), and several models with 10xnm lithographies. Moreover, our study considers a large number of factors that were not studied in previous work, such as the effect of firmware versions and failure correlations within a RAID group.

Where we report statistics that were also considered in previous work, we have included a comparison in the relevant sections of our paper. In other cases, a direct comparison with failure rates reported in prior work is not meaningful. For example, the Facebook [22] and Microsoft [23] studies focus on unccorrectable errors and fail-stop events respectively, which are different from the drive replacements considered in our study. Furthermore, fail-stop events do not always lead to drive replacements, and other events that might lead to replacements are not included in the rates reported in [23]. Similarly, while the study of drives at Alibaba [34] includes a breakdown of reason for replacement, their taxonomy is different, with categories that do not map to ours. Moreover, their work does not report on rates of failures (only the relative frequency of reasons).

8   Lessons learned

• Our observations emphasize the importance of firmware updates, as earlier firmware versions can be correlated with significantly higher failure rates (§5.5). Yet, we observe that 70% of drives in our study remain at the same firmware version throughout the length of our study. Consequently, we encourage enterprise storage vendors to make firmware upgrades as easy and painless as possible, so that customers apply the upgrades without worries about stability issues.

• A question that often comes up when configuring RAID groups is how the size of a group, in terms of number of drives, will affect its reliability. After all, intuitively, more drives create more potential for failures. Our observations show that larger RAID groups might not be as bad as often thought. While large RAID groups have a higher number of drive replacements, we have no evidence that the rate of multiple failures per group (which is what creates potential for data loss) is correlated with RAID group size (§6).

• Our results highlight the occurrence of temporally correlated failures within the same RAID group (§6). This observation indicates that single parity RAID configurations (e.g., RAID-5), might be susceptible to data loss, and realistic data loss analysis certainly has to consider correlated failures.

• Drives with very large capacities experience higher failure rates overall and see more severe failures (§5.3). The higher failure rate could stem from the larger amount of NAND and dies on the drives, emphasizing the importance of a drive and its system being able to handle a partial drive failure, such as a die failure. NetApp is working toward this direction by carving out the lost capacity of a dead die from the OP area.

• Our observation regarding the smaller rate of predictive failures for larger capacities (§5.3) also brings up the question whether large capacity drives require different types of failure predictors and potentially more input from the drive on its internal issues (e.g., a bad die or issues with DRAM).

• There is renewed concern around NAND-SSDs reliability with the introduction of QLC NAND, whose PE cycle limit is significantly lower than current TLC NAND. Based on our data, we predict that for the vast majority of enterprise users, a move towards QLC's PE cycle limits poses no risks, as 99% of systems use at most 15% of the rated life of their drives.

• There has been a fear that the limited PE cycles of NAND SSDs can create a threat to data reliability in the later part of a RAID system's life due to correlated wear-out failures, as the drives in a RAID group age at the same rate. Instead, we observe that correlated failures due to infant mortality are likely to be a bigger threat. For example, for the 3D-TLC drives in our study, the failure rate at the peak of infant mortality is 2.5X larger than later in life (§5.1).

• We observe unexpected behavior for failure rates as a function of age (§5.1). In contrast to the "bathtub" shape assumed by classical reliability models, we observe no signs of failure rate increases at end of life and also a very drawn-out period of infant mortality, which can last more than a year and see failure rates 2-3X larger than later in life. This brings up the question what could be done to reduce these effects. One might consider, for example, an extended, more intense burn-in period before deployment, where drives are subjected to longer periods of high read and write loads. Given the low consumption of PE cycles that drives see in the field (99% of drives do not even use up 1% of their PE cycle limit), there seems to be room to sacrifice some PE cycles in the burn-in process. More detailed recommendations would require a more thorough understanding of the relationship between PE cycles and failure rates; we are currently working on collecting such data.

• When choosing among drive types/models, our results indicate that from a reliability point of view, flash type (i.e., eMLC versus 3D-TLC) seems to play a smaller role than lithography (i.e., 1xnm versus 2xnm eMLC) or capacity (§5.2–5.4).

9   Acknowledgements

References

[1] Alan T. Arnholt and Ben Evans. *BSDA: Basic Statistics and Data Analysis*, 2017. R package version 1.2.0.

[2] 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.

[3] Hanmant P Belgal, Nick Righos, Ivan Kalastirsky, Jeff J Peterson, Robert Shiner, and Neal Mielke. A new reliability model for post-cycling charge retention of flash memories. In *Proceedings of the 40th Annual International Reliability Physics Symposium*, pages 7–20. IEEE, 2002.

[4] Matias Bjørling. From Open-Channel SSDs to Zoned Namespaces. In *Linux Storage and Filesystems Conference (Vault 19)*, 2019.

[5] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, pages 115–128. USENIX Association, 2010.

[6] Adam Brand, Ken Wu, Sam Pan, and David Chin. Novel read disturb failure mechanism induced by FLASH cycling. In *Proceedings of the 31st Annual International Reliability Physics Symposium*, pages 127–132. IEEE, 1993.

[7] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error patterns in MLC NAND flash memory: Measurement, Characterization, and Analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 521–526. EDA Consortium, 2012.

[8] Yu Cai, Yixin Luo, Erich F Haratsch, Ken Mai, and Onur Mutlu. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 551–563. IEEE, 2015.

[9] Yu Cai, Onur Mutlu, Erich F Haratsch, and Ken Mai. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *31st International Conference on Computer Design (ICCD)*, pages 123–130. IEEE, 2013.

[10] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Cristal, Osman S Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *30th International Conference on Computer Design (ICCD)*, pages 94–101. IEEE, 2012.

[11] Paolo Cappelletti, Roberto Bez, Daniele Cantarelli, and Lorenzo Fratin. Failure mechanisms of Flash cell in program/erase cycling. In *Proceedings of the IEEE International Electron Devices Meeting*, pages 291–294. IEEE, 1994.

[12] 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.

[13] Robin Degraeve, F Schuler, Ben Kaczer, Martino Lorenzini, Dirk Wellekens, Paul Hendrickx, Michiel van Duuren, GJM Dormans, Jan Van Houdt, L Haspeslagh, et al. Analytical percolation model for predicting anomalous charge loss in flash memories. *IEEE Transactions on Electron Devices*, 51(9):1392–1400, 2004.

[14] Jon G Elerath. AFR: problems of definition, calculation and measurement in a commercial environment. In *Annual Reliability and Maintainability Symposium. 2000 Proceedings. International Symposium on Product Quality and Integrity (Cat. No. 00CH37055)*, pages 71–76. IEEE, 2000.

[15] Jon G Elerath. Specifying reliability in the disk drive industry: No more MTBF's. In *Annual Reliability and Maintainability Symposium. 2000 Proceedings. International Symposium on Product Quality and Integrity (Cat. No. 00CH37055)*, pages 194–199. IEEE, 2000.

[16] Atul Goel and Peter Corbett. RAID triple parity. *ACM SIGOPS Operating Systems Review*, 46(3):41–49, 2012.

[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] S Hur, J Lee, M Park, J Choi, K Park, K Kim, and K Kim. Effective program inhibition beyond 90nm NAND flash memories. *Proc. NVSM*, pages 44–45, 2004.

[19] Seok Jin Joo, Hea Jong Yang, Keum Hwan Noh, Hee Gee Lee, Won Sik Woo, Joo Yeop Lee, Min Kyu Lee, Won Yol Choi, Kyoung Pil Hwang, Hyoung Seok Kim, et al. Abnormal disturbance mechanism of sub-100 nm NAND flash memory. *Japanese journal of applied physics*, 45(8R):6210, 2006.

[20] Myoungsoo Jung and Mahmut Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-level Implications. In *Proceedings of the 2013 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '13)*, pages 203–216, 2013.

[21] Jae-Duk Lee, Chi-Kyung Lee, Myung-Won Lee, Han-Soo Kim, Kyu-Charn Park, and Won-Seong Lee. A new programming disturbance phenomenon in NAND flash memory by source/drain hot-electrons generated by GIDL current. In *Non-Volatile Semiconductor Memory Workshop, 2006. IEEE NVSMW 2006. 21st*, pages 31–33. IEEE, 2006.

[22] 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.

[23] 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.

[24] NetApp Inc. Data ONTAP 9. `http://www.netapp.com/us/products/platform-os/ontap/`.

[25] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, page 109–116, New York, NY, USA, 1988. Association for Computing Machinery (ACM).

[26] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, volume 7, pages 17–23, 2007.

[27] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. *ACM Transactions on storage (TOS)*, 6(3):9:1–9:23, September 2010.

[28] Bianca Schroeder and Garth A Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, volume 7, pages 1–16, San Jose, CA, 2007.

[29] 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.

[30] Zoned Storage. NVMe Zoned Namespaces. `https://zonedstorage.io/introduction/zns/`. Accessed: 2019-09-21.

[31] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, et al. A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme. *IEEE Journal of Solid-State Circuits*, 30(11):1149–1156, 1995.

[32] Hung-Wei Tseng, Laura Grupp, and Steven Swanson. Understanding the Impact of Power Loss on Flash Memory. In *Proceedings of the 48th Design Automation Conference (DAC '11)*, pages 35–40, San Diego, CA, 2011.

[33] Wikipedia. S.M.A.R.T. `https://en.wikipedia.org/wiki/S.M.A.R.T.` Accessed: 2019-09-12.

[34] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and actions: What we learned from 10k ssd-related storage system failures. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 961–976, Renton, WA, July 2019. USENIX Association.

[35] Jimmy Yang and Feng-Bin Sun. A comprehensive review of hard-disk drive reliability. In *Annual Reliability and Maintainability. Symposium. 1999 Proceedings (Cat. No. 99CH36283)*, pages 403–409. IEEE, 1999.

[36] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the Robustness of SSDs Under Power Fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, pages 271–284, San Jose, CA, 2013. USENIX Association.

Making Disk Failure Predictions SMARTer!

Sidi Lu
Wayne State University

Bing Luo
Wayne State University

Tirthak Patel
Northeastern University

Yongtao Yao
Wayne State University

Devesh Tiwari
Northeastern University

Weisong Shi
Wayne State University

Abstract

Disk drives are one of the most commonly replaced hardware components and continue to pose challenges for accurate failure prediction. In this work, we present analysis and findings from one of the largest disk failure prediction studies covering a total of 380,000 hard drives over a period of two months across 64 sites of a large leading data center operator. Our proposed machine learning based models predict disk failures with 0.95 F-measure and 0.95 Matthews correlation coefficient (MCC) for 10-days prediction horizon on average.

1 Introduction

Hard disk drives (HDDs) continue to be a key driving factor behind enabling modern enterprise computing and scientific discovery — residing in large-scale data centers. Unfortunately, HDDs are not only the most frequently replaced hardware components of a data center; they are also the main reason behind server failures [82]. The failure of HDDs can result in data loss, service unavailability, increases in operational cost and economic loss [42,76]. Consequently, the storage community has invested a significant amount of effort in making disks reliable and, in particular, predicting disk failures [4,9,19,23,24,36,41,51,54,58,59,85,89,92]. Although widely-investigated, effective hard disk failure prediction still remains challenging [83,88] and hence, the storage community benefits from the disk reliability field-studies [8,37,44,53,55,60,65,77,83,88]. Unfortunately, such field studies are not published often enough and are limited in sample size [8,9,28,30,37,60,83,88,89].

To bridge this gap, we perform large-scale disk failure analysis, covering 380,000 hard disks and five disk manufacturers distributed across 10,000 server racks and 64 data center sites over two months, hosted by an enterprise data center operator — one of the largest disk failure analysis studies reported in the literature [4,9,51,83].

For the first time, this paper demonstrates that disk failure predictions can be made highly accurate by combining disk performance and disk location data with disk monitoring data (Self-Monitoring, Analysis, and Reporting Technology — SMART data). Traditionally, disk failure prediction works have largely focused on using SMART data for predicting disk failures — this is based on in-the-field evidence that SMART attributes (e.g., correctable
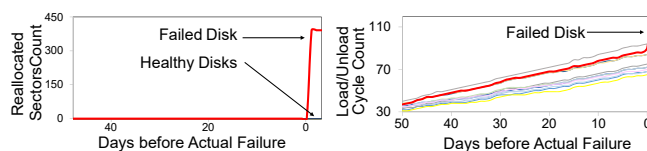
Figure 1: SMART attributes of healthy vs. failed disks prior to disk failures.
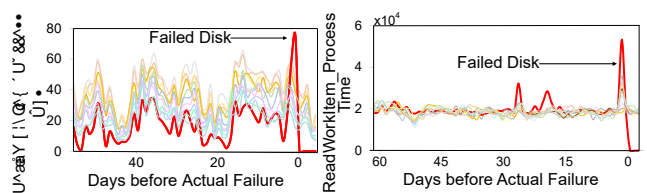
Figure 2: Performance metrics of healthy vs. failed disks prior to disk failures.

errors, temperature, disk spin-up time, etc.) are correlated with the disk health and indicative of eventual failure. While this conventional wisdom holds true as shown by previous works, we found that SMART attributes do not always have the strong predictive capability of making disk failure predictions at longer prediction horizon windows for all disks (i.e, predicting disk failures a few days before the actual failure instead of a few hours). This is primarily because the value of SMART attributes often does not change frequently enough during the period leading up to the failure, and the change is often noticeable only a few hours before the actual failure, especially in hard-to-predict cases.

On the other hand, the value of performance metrics may exhibit more variations much before the actual drive failure. A small example is shown in Figure 1 and Figure 2. We observe that the performance metrics of failed disk drives may indeed show distinguishable behavior from healthy disks (Figure 2) while SMART attributes do not (Figure 1). In Figure 1, the SMART attributes of healthy disks show the same value or similar pattern as failed disks located on the same server until the time of disk failure. For the performance metrics shown in Figure 2, although the trends of failed disks are close to healthy disks, failed disks may report multiple sharp impulses before they actually fail. Only a subset of SMART attributes are shown in the plot, but others also show similar behavior (our methodology is
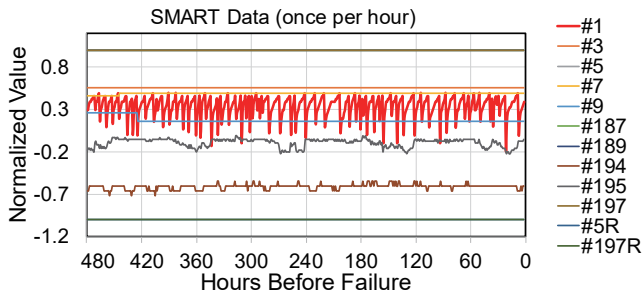
Figure 3: Values of SMART attributes before a hard disk failure, collected on an hourly basis, extracted from the open-source Baidu dataset [40]. The legend on the right shows the IDs of disk SMART attributes as defined by the industry standard [3], and "R" represents the raw value of an attribute.

covered in Section 2). We note that this example evidence does not suggest that *all* failed disk drives show variation in performance metrics leading up to the failure, or that SMART attributes do not change for any failed disks. Instead, it shows that performance metrics, when combined with a traditional approach of using SMART attributes, may be more powerful than using SMART attributes alone, especially for hard-to-predict failures.

One could argue that SMART attributes not exhibiting distinct patterns between healthy and failed disks is specific to this data center under study. To test this hypothesis, we plotted the normalized value of SMART attributes of failed and healthy disks from a publicly available disk failure dataset released by Baidu in 2016 [40]. Figure 3 shows that the normalized values of 12 SMART attributes of a randomly selected failed disk do not vary noticeably leading up to the failure — 477 hours before its actual failure. This observation is particularly notable, especially, given that the SMART attributes for this dataset are collected at much finer-granularity (one hour) as opposed to traditional per-day granularity (e.g., Backblaze public dataset [46]). Thus, SMART attributes alone may not be able to predict all disk failures.

Intuitively, the addition of performance metrics toward disk failure prediction increases the predictive power because it increases our coverage in capturing the workload characteristics accessing the storage system, beyond what SMART attributes cover. The nature of workloads running on a system often affects the failure rates of different system components, not only disks. But, it's much more challenging to obtain and incorporate workload related information due to the business-sensitive nature of data center workloads. As shown in Section 5, performance metrics can act as a good proxy for workload characteristics for disk failure prediction.

Finally, this paper shows that disk failure prediction can be further improved by incorporating the location information of disk drives in the data center — an aspect that has not been explored in the previous disk failure prediction works because typically data center logs do not include location and organization of disks by default. Intuitively, the addition of location information toward disk failure prediction increases the predictive power because it increases our coverage of the operating conditions of data center disks.

Disks in close spatial neighborhoods are more likely to be affected by the same environmental factors, such as relative humidity and temperature, which are responsible for accelerating disk component failures [55,73]. Notably, disks with physical proximity are likely to experience similar vibration levels. Although vibration is not a part of the SMART attributes or performance metrics, it is known to affect the reliability of disk drives [56,65]. Therefore, adding location information can capture disks operating under similar environmental or operating conditions which can experience similar failure characteristics. Our evaluation (Section 5) shows that adding location information to SMART attribute information indeed improves the failure prediction quality, although as expected, the effects are not as large as adding performance metrics to SMART.

While using the combination of SMART attributes, performance metrics, and location information is likely to improve disk failure prediction quality, the types of attributes, and the raw amount of combined information is almost unmanageable. It is unclear what attributes should be selected and how they should be used. Traditional rule-based or analytical models are not likely to exploit the hidden interactions among different attributes of the same type (e.g., SMART) and different types (e.g., performance vs. SMART). Therefore, to increase the effectiveness of our approach, we take advantage of machine learning (ML) models for leveraging such hidden interactions, as done in several previous disk failure prediction works [9,51,54,65,89].

Our core contributions are not in the development of machine learning based models, built on top of well-understood and mature models such as naive Bayes classifier (Bayes) [36], random forest (RF) [52], gradient boosted decision tree (GBDT) [29,91], and long short-term memory networks (LSTM) [23,38]. Instead, the core usefulness of our study is in providing actionable insights, trade-off lessons learned in applying these models, and assessment of model robustness. Additionally, we develop and evaluate a new hybrid deep neural networks model, convolutional neural network long short-term memory (CNN-LSTM) [2] for disk failure prediction that achieves close to the best prediction quality in most of the test cases.

Summary of Our Contributions:

⋆ *This paper presents findings from one of the largest disk failure prediction studies covering 380,000 hard drives over a period of two months across 64 sites of a leading data center operator.* Our disk failure prediction framework and the dataset used in this study including performance, SMART, and location attributes is hosted at `http://codegreen.cs.wayne.edu/wizard`.

⋆ *This paper provides experimental evidence to establish that performance and location attributes are effective in improving the disk failure prediction quality.* We show that, as expected, machine learning based models can be useful in predicting disk failures. But, as we discover and discuss in this paper, there are several trade-offs in the model selection. We also understand, discuss, and explain the limitations of these models. This paper provides details of an experimental and evaluation methodology for effective disk failure prediction.

⋆ *Overall, our evaluation shows that no single machine learning model is a winner across all scenarios, although CNN-LSTM is fairly effective across different situations. We achieve up to 0.95 F-measure [66] and 0.95 MCC (Matthews correlation coefficient) [10, 35, 43, 71] score for a 10-day lead-time prediction horizon (Refer to Section 4 for the definitions of F-measure and MCC).* We show that combining SMART attributes, performance metrics, and location records enables us to do disk failure prediction with long lead-times, although the prediction quality changes with the lead time window size.

2 Background and Methodology

This study covers the disk and server data measured and collected at a large data center. Over, the dataset spans over 64 data center sites, 10,000 server racks and 380,000 hard disks for roughly 70 days. This corresponds to roughly 2.6 million device hours [4, 9, 51, 83]. We note that during this period, the data center housed more than two million hard disks, but not all of them are included in our study because we only focus on those disks that have logged data in all three aspects: SMART, performance, and location. Collection and storage of both performance and SMART data are not enabled for all disks due to performance overhead and business-sensitivity concerns.

Next, we assess the types of disk events recorded at the data center sites and describe our definition of disk failure. Then, we discuss all three types of data collected and analyzed for this study: (1) disk SMART attributes (most commonly used for disk failure prediction by other studies [4, 19, 79, 87], (2) performance data, and (3) spatial location data of disks.

Table 1: SMART attributes for disk failure analysis.

ID	Attribute Names	ID	Attribute Names
1	Read Error Rate	7	Seek Error Rate
9	Power-On Hours	192	Power-off Retract Count
10	Spin Retry Count	193	Load/Unload Cycle Count
3	Spin-Up Time	194	Temperature
12	Power Cycle Count	197	Current Pending Sector Count
4	Start/Stop Count	198	Uncorrectable Sector Count
5	Reallocated Sectors Count	199	UltraDMA CRC Error Count

2.1 Definition of Disk Failure

Given the complexity of disk failures, there is no common, agreed-upon universal definition of a disk failure [53]. Latent sector errors (LSEs) are typically considered to be one of the most common disk errors which cause disk failures. However, a large-scale study of disk failures [75] shows that a small number of LSEs alone do not necessarily indicate that a disk failure has occurred or is imminent, but LSEs may cause performance degradation that could eventually lead to a "failure" — where error messages such as "the system cannot connect to the disk" or "disk operation exceeded the prescribed time limit" are treated as disk failures and warrant disk replacement. In this paper, we consider a disk to be failed when the production data center operator deems a disk necessary to be replaced. The IT operators of the production data center we study deem it appropriate for a disk to be replaced or repaired when there is a failed read/write operation and the disk cannot function properly upon restart. All other disks are considered healthy.

2.2 Disk SMART Data

SMART attributes values are produced and logged under the Self-Monitoring, Analysis and Reporting Technology (SMART) monitoring system for hard disks, which detects and reports various indicators of drive reliability [3]. The number of available SMART attributes is more than 50, but not all disks log all of the attributes at all times. For our study, we select 14 SMART attributes (Table 1) as features for our training models using the method described in Section 3. More than 97% of our disks reported these attributes, and these attributes also overlap with the widely used attributes for disk failure prediction by other studies [9, 51, 54, 65, 89]. In our study, these SMART attributes are collected continuously and reported at per-day granularity during the whole duration of the data collection period, similar to previous works [37, 54, 54]. As discussed earlier, more frequent SMART reporting did not necessarily improve the prediction quality at the start of this study and hence, once-a-day reporting was employed.

In our study, we consider two values corresponding to each SMART attribute in Table 1: (1) raw value of the attribute, and (2) normalized value of the attribute. Raw values are collected directly by the sensors or in-

Table 2: Selected disk-level performance metrics.

ID	Metrics	ID	Metrics
1	DiskStatus	7	Background_Checksum_ReadFailQps
2	IOQueueSize	8	TempFile_WriteWorkItem_SuccessQps
3	ReadSuccess_Throughput	9	TempFile_WriteSuccess_Throughput
4	ReadWorkItem_QueueTime	10	NormalFile_WriteWorkItem_SuccessQps
5	ReadWorkItem_SuccessQps	11	NormalFile_WriteWorkItem_QueueTime
6	ReadWorkItem_ProcessTime	12	NormalFile_WriteSuccess_Throughput

Table 3: Selected server-level performance metrics and the corresponding categories.

Categories	Metrics	Categories	Metrics
disk_util	max	udp_stat	udp_outdatagrams
tcp_segs_stat	tcp_outsegs	disk_sys_read_write	read
page_activity	page_in	net_pps_summary	net_pps_receive
disk_summary	total_disk_read	net_summary	receive_speed
disk_throughput	read	page_activity	page_out
disk_util	avg	udp_stat	udp_indatagrams
memory_summary	mem_res	disk_summary	total_disk_write
tcp_currestab	NONE	net_pps_summary	net_pps_transmit
cpu_summary	cpu_kernel	tcp_segs_stat	tcp_insegs

ternal software in disks, and their interpretation can be specific to the disk manufacturer. Normalized values are obtained by mapping the related raw value to one byte using vendor-specific methods. Higher normalized value usually indicates a healthier status, except in the case of head load and unload cycles and temperature. We note that whether a higher (or lower) raw value is better often depends on the attribute itself. For example, a higher value of "Reallocated Sectors Count" represents that more failed sectors have been found and reallocated (worse case), while a lower value of "Throughput Performance" indicates a possibility of a disk failure.

2.3 Performance Data

In our study, we measure and collect two types of performance metrics maintained by the OS kernel, i.e., disk-level performance metrics and server-level performance metrics. Disk-level performance metrics include IOQueue size, throughput, latency, the average waiting time for I/O operations, etc. Server-level performance metrics include CPU activity, page in and out activities, etc. Performance metrics are reported at per-hour granularity because we found that hourly granularity was effective in improving the prediction quality. However, the storage overhead of all performance metrics can become significant at scale and over time, and it can incur significant operational costs. Therefore, as described in Section 3, we use a simple method to down-select the number of metrics used by our ML models to manage prediction quality with low storage overhead.

2.3.1 Disk-level Performance Metrics

In our study, we measure and collect 12 disk-level performance metrics in total; all of these metrics are used in this paper. Table 2 shows the 12 metrics related to individual disks.

The distinct value of "DiskStatus" represents different disk working statuses. For example, 0, 1, 2, 4, 8, 16, 32, 64 and 128 indicate healthy, initial, busy, error, hang, only read, shutdown, repair, and complete repair states, respectively. "IOQueueSize" shows the number of items in the IO worker queue. "Normal-File/TempFile_WriteSuccess_Throughput" represents the throughput of normal/temp files successfully written to disks. "NormalFile/TempFile_WriteWorkItem_SuccessQps" and "ReadWorkItem_SuccessQps" stand for the number of normal/temp files successfully

written/read by the disk per second. Similarly, "NormalFile_WriteWorkItem_QueueTime" indicates the average waiting time for disks to write. "ReadSuccess_Throughput," "ReadWorkItem_ProcessTime," and "ReadWorkItem_QueueTime" indicate the throughput, process time, and the average waiting time through the reading process of disks.

2.3.2 Server-level Performance Metrics

As to the server-level metrics, we have 154 metrics categorized into 54 categories; each category has a different number of metrics. We first extract the most common pairs of category-metrics and make sure that more than 97% of servers have these server-level metrics. We down-select the number of metrics to 18 that we feed to our machine learning model to manage prediction quality vs. storage overhead via a simple method described in Section 3. Table 3 lists the 18 server-level performance metrics and their corresponding categories.

"Tcp_outsegs" displays the total number of the disk storage segments that have been sent, including those on current connections but excluding those containing only retransmitted octets. Similarly, "tcp_insegs" shows the total number of disk storage segments received, and "tcp_currestab" represents the number of TCP connections for which the current state is either established or close-wait. "Udp_outdatagrams" displays the total number of the disk storage UDP datagrams that have been sent. "Page_in" represents the number of transferring data from a disk to the memory per second. Similarly, "page_out" occurs when the data is transferred from the memory to a disk. Packets per second (PPS) is a measure of throughput for network devices. Hence, "net_pps_receive" and "net_pps_transmit" indicate the rate of successfully receiving and transmitting messages over a communication channel, respectively. Note that the performance data also includes network-related (TCP/UDP) metrics some of which appear in the selected server-level performance metrics; this suggests that network- and disk- activity might be correlated and may be predictive of disk failures when combined.

2.4 Disk Spatial Location Data

As noted in Table 4, our disks are spread over more than 50 sites and 10,000 racks. All disks are directly

Table 4: Numbers of sites, rooms, racks, and servers.

	# of Sites	# of Rooms	# of Racks	# of Servers
Total	64	199	10,440	120,000

attached to a server. Each disk has four levels of location markers associated with it: site, room, rack, and server. One server may host multiple disks. Multiple servers could be on the same rack. A room has multiple racks, and a site may host several rooms. Location markers are used for both healthy and failed disks. Note that these location markers do not explicitly indicate the actual physical proximity between two disks, since the physical distance between two sites or rooms is not captured by our location coordinates, and they do not indicate the physical proximity within a room.

2.5 Other Methodological Considerations

Our disk failure prediction study is carefully designed to ensure that it is not prone to experimental pitfalls. For example, we verified that the disk failure rate is roughly similar over time across all 64 sites because if most disk failures happen during the same week it can skew the prediction quality. Similarly, we ensured that the concentration of disk failures in space is not skewed. Although failures in space have non-uniform distribution, we have verified that the density of failures in space changes over time. Our annual disk failure rate of ≈1.36% is consistent with failure rates observed at other data centers [47–49].

We note that missing SMART or performance data is a possibility and can itself be indicative of the system's health. For example, if failed disks observe a higher degree of missing data than healthy disks and the failed disks have been missing data continuously for a long period (e.g., more than the prediction horizon), then this feature alone could predict disk failure with high success rate. However, in our case, we observed that healthy and failed disks do not have an imbalance in terms of missing data. Furthermore, the length of continuous missing data is less than one day in most cases because we have multiple types of data: performance and SMART. The likelihood of missing all samples from both groups simultaneously is low — and if data is missing, it often points to an abrupt disk/server failure or other infrastructure-related issues.

We also ensure that disk failures are not concentrated on a particular manufacturer only, or limited to only old-aged drives. Although our datset has multiple vendors and drives of different ages, we verified that failure prediction does not reduce to trivially knowing the vendor name or age of the disk — although these features are used by our machine learning models to improve the quality of prediction. We explored training and building vendor-specific ML models, but we found that this

leads to multiple problems: (1) overfitting to a particular vendor, (2) lack of portability across sites and vendors, (3) managing multiple models, and (4) lower prediction quality than the approach taken in this paper (normalizing the attributes across vendors and disks as discussed in Section 3).

3 Selection of SMART and Performance Attributes

In this section, we present a simple method to down-select SMART and performance metrics. These down-selected metrics are then fed to our machine learning models as input features. Unless otherwise noted, we use this method for selecting important features and use the resulting features to present evaluation results. However, one could argue that machine learning models can automatically infer important features out of all the input features. The reason for performing this step is to demonstrate that down-selecting features using a simple method does not compromise the prediction quality, as we evaluate in Section 5. The benefit of this step is the saving in storage overhead. Although our study needed to store all the features (over 100) to demonstrate the effectiveness of down-selection, in the future, data center operators can use the method to save storage space and reduce processing overhead. Since the range of values for different attributes across different disks and vendors varies widely, it is hard to perform meaningful comparisons. Thus, we pre-process the SMART and performance metrics using a min-max normalization to facilitate a fair comparison between them as per equation: $x_{norm} = (x - x_{min})/(x_{max} - x_{min})$. Here, x is the original value of a feature, x_{min} is the minimum and x_{max} is the maximum value of the feature (over all observations). We use 0 to represent the NULL value, and we label constant features as 0. Next, we leverage Youden's J index (also named as J-Index) [27,74] for the down-selection of features.

3.1 How does J-Index (JIC) work?

After features are normalized to the scale of 0-1, we set a series of threshold candidates for each feature with a step of 0.01, starting from 0 until 1. For each threshold candidate t, we calculate the value of the corresponding J-Index [6]. We define J-Index classification (JIC) as:

$$\text{J-Index} = \text{True Positive Rate} + \text{True Negative Rate} - 1$$

$$= \frac{\text{TP}}{\text{TP+FN}} + \frac{\text{TN}}{\text{TN+FP}} - 1$$

Here T and F indicate whether the prediction result is correct; P and N denote the disk is classified as failed (positive) or healthy (negative). TP denotes the number of actually failed disks that are correctly predicted as
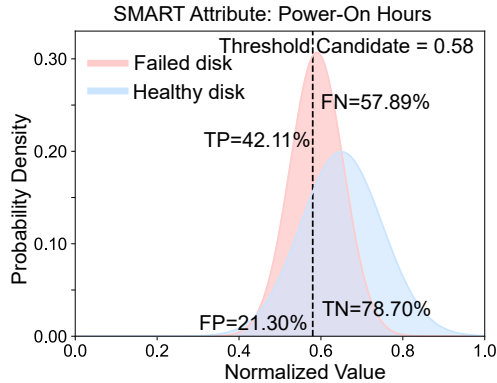
Figure 4: An example of J-Index classification (JIC): Distinguishing failed disks from healthy disks. The upper curve represents the failed disk, and the lower curve indicates the healthy disk.

failed, and TN denotes the number of healthy disks that are correctly predicted as healthy. Similarly, FP denotes the number of healthy disks that are falsely predicted as failed, and FN denotes the number of failed disks that are falsely predicted as healthy.

More specifically, suppose the input feature is Power-On Hours, and the distribution looks like Figure 4 for the current threshold candidate t (t = 0.58 as an example here). We calculate the percentage of failed disks that are distributed on the left-hand part of t, which is 42.11%, i.e., TP = 42.11%. Similarly, we have FN = 57.89%, FP = 21.30%, and TN = 78.70%. It is intuitive that we predict a disk is healthy if its value is greater than 0.58 or it is otherwise failed. We also calculate the corresponding J-Index based on the above definition. Following this method, for a specific feature, we have a series of threshold candidates and their corresponding J-Indexes. The range of J-Indexes is 0 to 1. A higher J-Index means the corresponding threshold candidate is more distinguishable to identify failed disks from healthy disks. Therefore, the threshold candidate with the highest J-Index is selected as the best (final) threshold for a feature.

Intuitively, J-Index classification is a low-overhead and practical method for IT operators to adopt and perform feature selection on their datsets.

Table 5 shows the J-Indexes (greater than 0.1) for SMART attributes. The fourth and sixth columns (yellow color) represent the percentages of disks that are smaller than the threshold, while the fifth and last columns (blue color) show the percentages of disks that are greater than the threshold. For each attribute, the first bold font indicates the true positive rate, and the second bold font denotes the true negative rate. Since failures are not always supposed to be values that are less than the threshold, i.e., there are upper-bound thresholds and lower-bound thresholds for failed disks, the

Table 5: Highest J-Indexes for SMART attributes (R represents raw value, N denotes normalized value).

ID	Threshold	J-Index	% of failed disks		% of healthy disks	
9R	0.58	0.21	**42%**	58%	21%	**79%**
9N	0.54	0.19	52%	**48%**	**72%**	28%
3R	0.72	0.18	80%	**20%**	**98%**	2%
5R	0.49	0.18	**18%**	82%	0%	**100%**
194N	0.50	0.18	**45%**	55%	27%	**73%**
194R	0.50	0.15	**96%**	4%	81%	**19%**
1R	0.38	0.14	**28%**	72%	14%	**86%**
12N	0.04	0.14	65%	**35%**	**79%**	21%
4N	0.01	0.13	64%	**36%**	**77%**	23%
5N	0.01	0.13	87%	**13%**	**100%**	0%
3N	0.59	0.13	77%	**23%**	**90%**	10%

Table 6: Highest J-Indexes for performance metrics.

ID	Threshold	J-Index	% of failed disk		% of healthy disk	
2	0.13	0.45	**100%**	0%	55%	**45%**
3	0.11	0.44	2%	**98%**	**46%**	54%
7	0.10	0.40	8%	**92%**	**48%**	52%
11	0.16	0.40	8%	**92%**	**49%**	51%
6	0.15	0.38	12%	**88%**	**50%**	50%
9	0.12	0.31	25%	**75%**	**56%**	44%
8	0.10	0.30	27%	**73%**	**56%**	44%

bold values for the true positive rate and true negative rate span multiple columns.

Similar to SMART attribute analysis, we would like to see if performance metrics could be the indicators of disk failures. Table 6 shows a part of the highest J-Indexes for performance metrics following the same formatting guide as Table 5. By employing the JIC method, we figure out a set of most informative disk-level and server-level performance metrics that are indicative of impending disk failures, i.e., we select the metrics that have the highest J-Indexes (greater than 0.1). We also present the best (final) thresholds of some of the selected metrics in Table 5 and Table 6.

Contrary to SMART attributes, performance metrics tend to have a higher true positive rate and a lower true negative rate. We observe that although a single performance metric is not perfect to distinguish failed disks from healthy disks, it has an overall higher J-Index than most of the SMART attributes based on our dataset. This indicates that performance metrics are likely to be predictive for disk failures.

Next, we show that performance metrics of failed disks may show different distinguishing patterns before failure compared to the healthy disks. Recall that there are 12 disk-level performance metrics in total. For each server that contains one or more failed disks (failed server), we extract these 12 metrics of each disk within 240 hours before disks are reported to be failed. If there is only one failed disk on a specific failed server, we keep the raw value of the failed disk (RFD) and calculate the average value of all healthy disks (AHD) for every time point. Then, we get the difference between RFD and AHD, which indicates the real-time difference between the signatures of failed disks and healthy disks on the
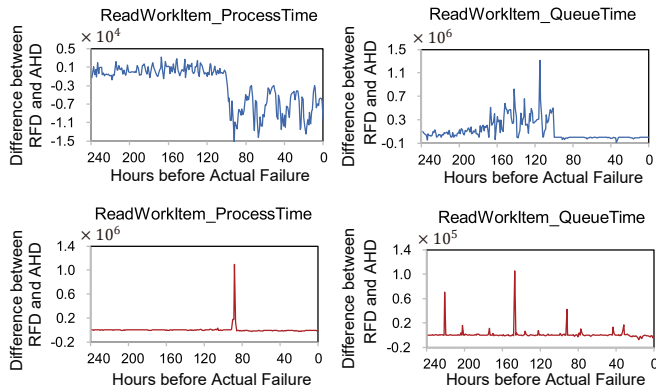
Figure 5: Different types of patterns of performance metrics observed 240 hours before disks failure.

same server. If there are N ($N \geq 2$) failed disks, then for each failed disk, we calculate the difference between RFD and AHD for every time point.

Figure 5 shows representative samples of the difference between RFD and AHD curves for different performance attributes on different servers. To reveal the patterns more intuitively, we use the raw values of metrics to calculate the difference between RFD and AHD rather than the normalized values in Figure 5. All disks on the same server have the same value of server-level performance metrics, and hence, 18 selected server-level performance metrics are not shown in the plot. The top two graphs of Figure 5 illustrate that some failed disks have a similar value to healthy disks at first, but then their behavior becomes unstable as the disk nears the impending failure. The bottom two graphs of Figure 5 show that some failed disks report a sharp impulse before they fail, as opposed to a longer erratic behavior. These sharp impulses may even repeat multiple times. We did not find such patterns for SMART attributes so far before the failure of this selected example. The diversity of patterns demonstrates that disk failure prediction using performance metrics is non-trivial.

4 ML Problem Formulation and Solution

Problem Definition. We formulate the problem of predicting disk failures as a classification problem. Specifically, we use $T = \{(\text{input}_i, \text{label}_i)\}_{i=1}^{n}$ to represent our training dataset, in which $\text{input}_i \in I$ denotes all input features. Here, $\text{label}_i \in \{0,1\}$ is a binary response variable for each disk i: 0 indicates healthy state and 1 indicates failed state. Our goal is to employ the best method to learn the function $f : I \rightarrow \{0,1\}$, which minimizes the loss function $\ell(h(\text{input}); \text{label})$, a measurement of the difference between the desired output and the actual output of the current model, such that the trained model is able to predict disk failures ($\text{label}_i = 1$) over a specific

prediction horizon with high accuracy.

More specifically, during the training process, assume we only use one attribute a as an input feature. For each disk, we have multiple readings of the attribute: $a_1, a_2,..., a_n$ (j is the time in a_j), and we treat $\{a_1,...,a_n\}$ as a sample. Since the input of a machine learning algorithm should be a fixed length of the observation period for each sample, n should be a fixed number. Our goal is to predict disk failure in advance, so a_j in $\{a_1,...,a_n\}$ should be the value of healthy states (of the healthy disks or healthy states prior to failures), i.e., a_j in $\{a_1,...,a_n\}$ does not contain failed state data. Note that we aim to predict *if* the disk will fail and not the exactly *when* the disk will fail in the next ten days.

Effective Measurements. To evaluate the effectiveness of our prediction approaches, we use Precision, Recall, F-measure, and Matthews correlation coefficient (MCC) to measure the wellness of our prediction approaches. Precision [22] indicates the proportion of TP among all predicted failures. Recall that the true positive rate (TPR) [81] represents the proportion of TP within all actually failed disks. Since our binary classification is largely imbalanced — there are many more healthy disks than failed disks — we also use F-measure [39,69] and MCC [10] as our evaluation metrics. F-measure is the harmonic average of precision and recall and ranges between 0 and 1 (higher is better). *We use MCC because it is a more balanced measure than F-measure, especially suitable for imbalanced data. It ranges from 1 (perfect prediction) to -1 (inverse prediction)*. These metrics are defined as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP+FP}}$$

$$\text{Recall (TPR)} = \frac{\text{TP}}{\text{TP+FN}}$$

$$\text{F-measure} = \frac{2*\text{Precision}*\text{Recall}}{\text{Precision}+\text{Recall}}$$

$$\text{MCC} = \frac{\text{TP}\times\text{TN}-\text{FP}\times\text{FN}}{\sqrt{(\text{TP+FN})(\text{TP+FP})(\text{TN+FP})(\text{TN+FN})}}$$

Prior ML Models and Our Models. Previous works have focused on leveraging fundamental classification and regression techniques for disk failure prediction [4,9,51]. These methods include naive Bayes classifier (Bayes) [69], random forests (RF) [52], gradient boosted decision trees (GBDT) [29,91] and long short-term memory networks (LSTM) [23,38]. Bayes is a family of probabilistic classifiers based on applying Bayes' theorem. RF and GBDT are types of traditional machine learning (ML) ensemble methods, while LSTM is a class of deep neural networks (DNNs). Since previous works have not considered performance and location features for disk failure prediction, we implement and

tune Bayes, RF, GBDT, and LSTM models to use them as a proxy for prior learning based disk failure prediction models. In addition, we consider a convolutional neural network with long short-term memory (CNN-LSTM) based model [72]. We implement our models in Python, using TensorFlow 1.5.0 [1], Keras 2.1.5 [34], and Scikit-learn libraries [64] for model building.

Brief Model Background and Intuitions. Bayes [69] is a probabilistic machine learning model used for classification tasks. RF [52] and GBDT [29, 91] are both ensemble methods that are constructed by a multitude of individual trees (called base learners or weak learners) and consider the conclusions of all trees to make accurate predictions through averaging or max voting.

The difference between RF and GBDT is that RF generates trees in a parallel manner (bagging algorithm) [52], while GBDT grows tress sequentially (boosting algorithms) [29,91]. More specifically, the bagging algorithm randomly takes data samples with replacement from the original dataset to train every weak learner, which means that the training stage of generating multiple learners is parallel (i.e., each learner is built independently). Boosting algorithm, however, uses all data to train each learner and builds the new learner in a sequential manner, and it assigns more weight to the misclassified samples to pay more attention to improving their predictability them during the training phase.

On the other hand, LSTM [23,38] is capable of addressing the long-term back-propagation problem (iteratively adjusting the weights of network connections to reduce the value of the loss function). LSTM includes a memory cell which tends to preserve information for a relatively long time. Hence, LSTM is effective for sequential data modeling, and employing LSTM to predict disk failure has been explored previously [23]. To further improve the performance of LSTM in the disk failure prediction, we integrate CNN and LSTM as a unified CNN-LSTM model (a CNN at the front and an LSTM network at the rear), since CNN and LSTM are complementary in the modeling capabilities — CNN offers advantages in selecting better features, while LSTM is effective at learning sequential data [2]. The choice of combining CNN and LSTM is inspired by the analysis presented by Pascanu *et al.* [63] — suggesting that the performance of LSTM could be further improved by taking better features as the input, which could be provided by CNN through dimensionality reduction [68]. Therefore, we include the CNN-LSTM approach to explore its effectiveness in the field of disk failure prediction.

Model Training and Testing Methodology. We use 5-fold cross-validation [50], which is a validation technique to assess the predictive performance of machine learning models, judge how models perform to an unseen dataset (testing dataset) [70] and avoid the overfitting issue. More specifically, our dataset is randomly partitioned into five equal-sized sub-samples. We take one sub-sample as the testing dataset at a time and take the remaining four sub-samples as the training dataset. We fit a model on the training dataset, evaluate it on the testing dataset, and calculate the evaluation scores. After that, we retain the evaluation scores and discard the current model. The process is then repeated five times with different combinations of sub-samples, and we use the average of the five evaluation scores as the final result for each method.

Tuning Hyperparameters of Models. We search for the best values of hyperparameters for all models using the hold-out method [45], which splits our original training phase data further into the hyperparameter training dataset (80% of the original training phase data) and the validation dataset (20% of the original training phase data). The biggest difference between the hold-out method and *k*-fold cross-validation approach (*k* refers to the number of sub-samples) is that the training and validation process of the hold-out approach only needs to be run once, while *k*-fold cross-validation needs to be run *k* times. In the hyperparameter tuning phase, we conduct a grid search to build and evaluate models for each combination of hyperparameters, and the goal is to find the best combination with the highest performance. For example, for RF and GBDT, we run experiments with different numbers of trees (estimators), and we settle on using 2000 trees in the RF model, and 1000 trees in the GBDT model, since using more than 2000 and 1000 trees, respectively, does not have significant improvements in practice. Using a similar method, the additive Lidstone smoothing parameter (α) of Bayes [20] was set to 2.

For LSTM-based models, after conducting a grid search on the values of hyperparameters to find the best combinations, we build an LSTM model with four layers and 128 nodes. For CNN-LSTM, in the CNN sub-module, we employ 1 one-dimensional convolutional layer at the front followed by one max-pooling layer and one flatten layer (shown in Figure 6). The 1D convolutional layer contains 128 filters which interpret snapshots based on the input. The max-pooling layer is responsible for consolidating and abstracting the interpretation to get a two-dimensional matrix of features. The flatten layer transforms the matrix into a vector, which is fed into the next classifier. The LSTM module consists of two LSTM layers and one dense layer (fully connected layer). We empirically set the same learning rate of 0.001 for the LSTM and CNN-LSTM models, and we set the drop-out rate to 0.25.
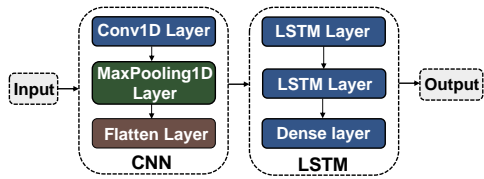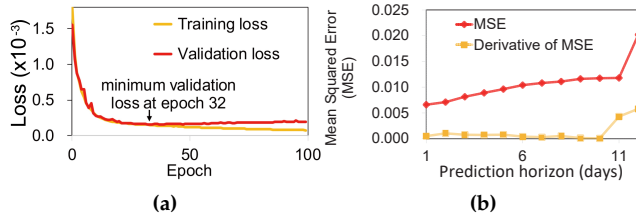
Figure 6: Structure of CNN-LSTM.

Figure 7: (a) The validation loss reaches its minimum value at 32 epochs for LSTM; thereafter it increases. (b) The mean squared error (MSE) and its derivative increases at a prediction horizon beyond 10 days.

Avoiding Overfitting of the Models. As far as LSTM and CNN-LSTM are concerned, one of the most important factors is the epoch [32], which indicates the number of iterations of processing the input dataset during the training process. A higher epoch value will reduce the error on training data; however, at a crucial tipping point, the network begins to over-fit the training data. Hence, finding the best value of the epoch is essential to avoid overfitting. Figure 7(a) shows the change in the value of the training and validation loss functions (the smaller, the better) as the epoch increases. Initially, the values of the two loss functions are decreasing with increasing epoch values; but after 32 epochs, the value of the validation loss function slowly increases (higher than the training loss), which indicates the over-fitting issue. Therefore, we choose 32 epochs for LSTM. Similarly, we choose 200 epochs for CNN-LSTM.

Feature Group Sets. We consider different input datasets to evaluate the effectiveness of different features: SMART attributes (S), performance metrics (P), and location markers (L). We construct six groups using different feature combinations: SPL, SL, SP, PL, S, and P. Table 7 shows the input features for these groups.

Prediction Horizon Selection. The first step in evaluating the ML model is to determine how long the prediction horizon should be. We choose 10 days as our prediction horizon, i.e., we aim to detect if a given disk will fail within the next 10 days, similar to previous studies [4, 9]. The 10-day horizon is long enough for IT operators to conduct early countermeasures. We also conduct a sensitivity study showing the change in the value of mean squared error (MSE) of different metrics

Table 7: Input features for six experimental groups. For performance metrics, the first column (red color) represents disk-level metrics, and the last two columns (yellow cells) represent server-level metrics.

	SMART	Performance			Location
SPL Group	28 attributes	12 metrics	18 metric categories	18 metrics	1 marker
SL Group	28 attributes	NONE			1 marker
SP Group	28 attributes	12 metrics	18 metric categories	18 metrics	NONE
PL Group	NONE	12 metrics	18 metric categories	18 metrics	1 marker
S Group	28 attributes	NONE			NONE
P Group	NONE	12 metrics	18 metric categories	18 metrics	NONE

for different lengths of prediction horizon, as shown in Figure 7(b) (using "ReadSuccessThroughput" as a representative example), where MSE indicates the average squared difference between the predicted values and the actual values [86]. We note that the derivative of MSE remains low for up to ten days, but it increases after ten days. This behavior can have slight variations across different features. Our prediction horizon is 10 days unless otherwise stated in our evaluation. We also evaluate the models' sensitivity with regard to prediction horizon (Section 5).

5 Results and Analysis

In this section, we present and analyze the results of various ML models, their sensitivity toward different feature groups, their limitations, robustness, and portability. Our discussion includes supporting evidence and reasons to explain observed trends, and implications of observed trends for data centers. First, we present the key prediction quality measures for all models and feature sets (Figure 8). We make several interesting observations as following:

1. We observe that the SPL feature group performs the best across all ML models, confirming our hypothesis that performance and location features are critical for improving the effectiveness of disk failure prediction, beyond traditional SMART attribute based approaches.

2. Adding location information improves the prediction quality across models, but the improvement is limited in absolute degree (e.g., less than 10% for CNN-LSTM in terms of MCC score). Interestingly, the effect of location information is pronounced only in the presence of performance features. The disk performance metrics are potentially correlated with disks' location information, Therefore, adding location markers may help ML mod-
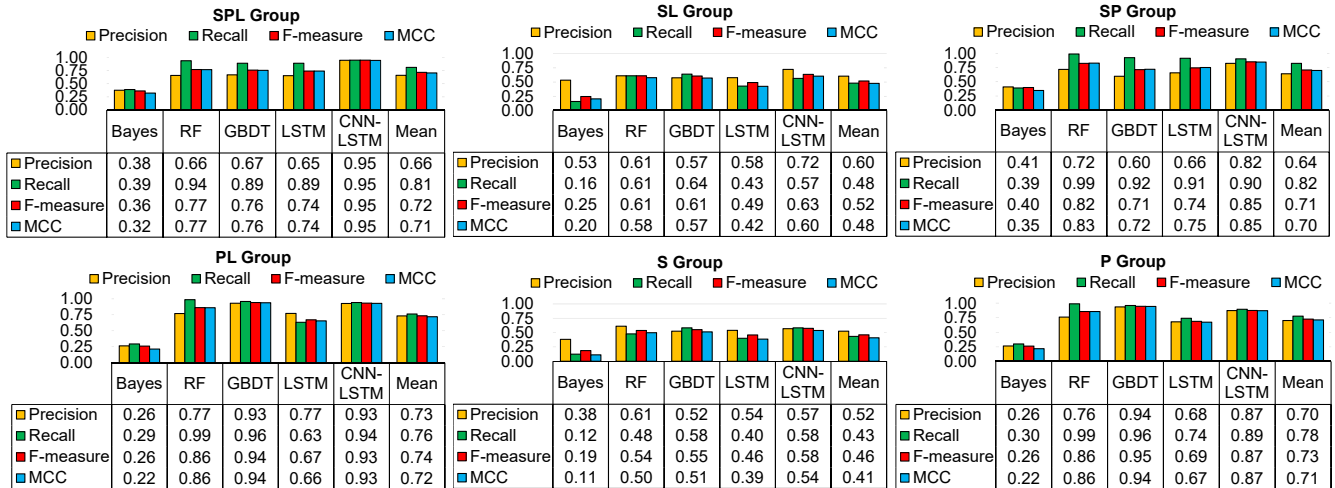
SPL Group

	Bayes	RF	GBDT	LSTM	CNN-LSTM	Mean
Precision	0.38	0.66	0.67	0.65	0.95	0.66
Recall	0.39	0.94	0.89	0.89	0.95	0.81
F-measure	0.36	0.77	0.76	0.74	0.95	0.72
MCC	0.32	0.77	0.76	0.74	0.95	0.71

SL Group

	Bayes	RF	GBDT	LSTM	CNN-LSTM	Mean
Precision	0.53	0.61	0.57	0.58	0.72	0.60
Recall	0.16	0.61	0.64	0.43	0.57	0.48
F-measure	0.25	0.61	0.61	0.49	0.63	0.52
MCC	0.20	0.58	0.57	0.42	0.60	0.48

SP Group

	Bayes	RF	GBDT	LSTM	CNN-LSTM	Mean
Precision	0.41	0.72	0.60	0.66	0.82	0.64
Recall	0.39	0.99	0.92	0.91	0.90	0.82
F-measure	0.40	0.82	0.71	0.74	0.85	0.71
MCC	0.35	0.83	0.72	0.75	0.85	0.70

PL Group

	Bayes	RF	GBDT	LSTM	CNN-LSTM	Mean
Precision	0.26	0.77	0.93	0.77	0.93	0.73
Recall	0.29	0.99	0.96	0.63	0.94	0.76
F-measure	0.26	0.86	0.94	0.67	0.93	0.74
MCC	0.22	0.86	0.94	0.66	0.93	0.72

S Group

	Bayes	RF	GBDT	LSTM	CNN-LSTM	Mean
Precision	0.38	0.61	0.52	0.54	0.57	0.52
Recall	0.12	0.48	0.58	0.40	0.58	0.43
F-measure	0.19	0.54	0.55	0.46	0.58	0.46
MCC	0.11	0.50	0.51	0.39	0.54	0.41

P Group

	Bayes	RF	GBDT	LSTM	CNN-LSTM	Mean
Precision	0.26	0.76	0.94	0.68	0.87	0.70
Recall	0.30	0.99	0.96	0.74	0.89	0.78
F-measure	0.26	0.86	0.95	0.69	0.87	0.73
MCC	0.22	0.86	0.94	0.67	0.87	0.71

Figure 8: Model prediction quality with different groups of SMART (S), performance (P), and location (L) features.

S Group — CNN-LSTM: FPR 3.9%, FNR 41.7%; LSTM: FPR 6.0%, FNR 60.0%; GBDT: FPR 4.6%, FNR 41.7%; RF: FPR 3.3%, FNR 52.2%; Bayes: FPR 5.6%, FNR 87.6%.

SL Group — CNN-LSTM: FPR 2.4%, FNR 43.5%; LSTM: FPR 5.5%, FNR 57.1%; GBDT: FPR 4.1%, FNR 36.1%; RF: FPR 3.4%, FNR 38.9%; Bayes: FPR 3.8%, FNR 84.0%.

SPL Group — CNN-LSTM: FPR 0.5%, FNR 5.1%; LSTM: FPR 2.9%, FNR 11.9%; GBDT: FPR 2.7%, FNR 10.0%; RF: FPR 2.9%, FNR 5.3%; Bayes: FPR 5.5%, FNR 62.1%.
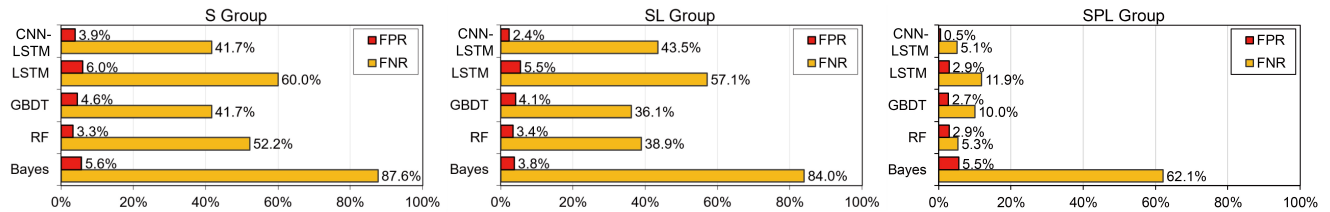
Figure 9: Model false positive rate ($FPR = FP/(FP + TN)$) and false negative rate ($FNR = FN/(TP + FN)$).

els amplify the hidden patterns in performance metrics.

3. While there is no single model winner across different feature groups, CNN-LSTM performs close to the best in all the situations, achieving an MCC score of 0.95 for the SPL group, compared to 0.77 MCC score for RF (next best method) for the SPL group. Further, we plot the false positive and false negative rates for different ML models for different feature groups (Figure 9). Figure 9 reveals interesting trends. First, SMART-attribute-based models have a very high false negative rate or FNR (failed disks predicted healthy) across all models. Adding performance and location features decreases the FNR significantly and hence, the prediction quality improves. It also decreases the false positive rate, but the scope for reduction is already limited.

Second, there is a trade-off between FPR and FNR in terms of cost (cost of disk failure vs. replacing healthy disks conservatively). Depending on the estimated costs of these factors, data center operators could choose between different models. For example, for the SPL group, GBDT provides lower FPR but higher FNR. Similarly, Figure 9 also shows that in the S group, such trade-offs exist between the RF and LSTM models.

4. Finally, we observe a trade-off between models with respect to the different availability of feature sets. Figure 8 shows that when a data center operator does not collect or have access to the performance features, traditional tree-based ML models (RF and GBDT) can perform roughly as well as complex neural network based models such as CNN-LSTM or LSTM. In fact, RF and GBDT models may even beat the LSTM model in absence of P and L features — this is similar to what a recent work has also shown which does not consider performance metrics [4].

Our work shows that adding performance and location features leads to a different and new outcome. Also, we note that the CNN-LSTM model takes much longer to train compared to simple tree-based models (up to four hours in our case for one training progress); therefore, in absence of performance and location features, RF and GBDT models can provide equally accurate predictions, and they might be preferred for building models based on the SMART data only due to the relatively lesser training time.

Next, we investigate when and how ML models fail to achieve high prediction accuracy over space and time.
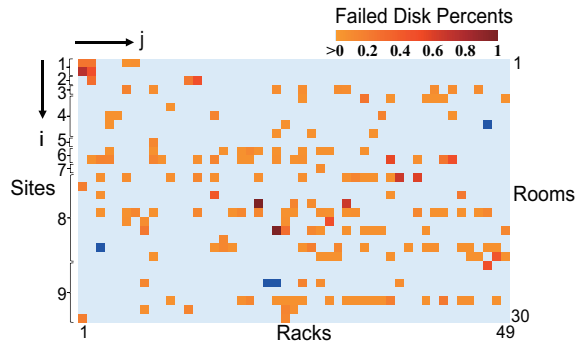
Figure 10: Mispredicted failures (blue) tend to occur in the locations where there is a low failure rate for all models. Each row stands for a room, and each column refers to a rack. i.e., the pixel of the j-th column and the i-th row represents the j-th rack of the i-th room. The color of each pixel indicates the failed disk percentage on the rack (pixel).

Where do ML models perform relatively poorly and why? Figure 10 shows that ML models are somewhat less effective at predicting with high accuracy and recall in areas where the concentration of failures is relatively lower. This is reasonable since ML models are not able to collect enough failed disk samples. ML models are by definition less effective for cases they have not been trained or situations they have not encountered before. This observation is important for data center operators as it emphasizes the need for adding location markers in disk failure prediction models.

When do ML models fail to predict and why? To understand the limitations of ML models better, we investigate the false positive (healthy disks predicted as failed) and false negative (failed disks predicted as healthy) predictions. Figure 11(a) shows the false positives categorized in 20-day windows for the CNN-LSTM model (other models produce similar trends). The number of false positives is very low initially as it predicts many disks as healthy though they eventually fail in that window — and, this is why the false negatives are high (Figure 11(b)). This can be explained by the lack of sufficient training data — the ML model does not have enough data and (conservatively) predicts that disks are healthy. This trend indeed reverses over time. Although the fraction of false positives appears to be very high toward the last window, we note that the actual number of false positives is quite low (Figure 9). This observation indicates the need for sufficiently long testing periods before concluding the prediction quality of ML models.

Is the prediction model portable across data center sites? Data centers operators often increase their number of sites over time, and it takes time to build models at
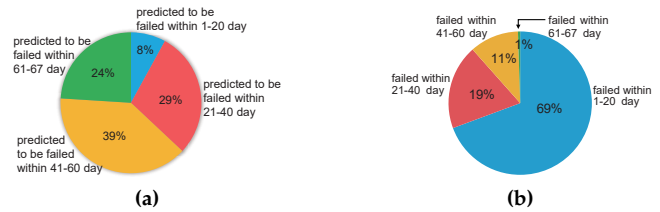

Figure 11: (a) Temporal distribution of CNN-LSTM model's false positives. (b) Temporal distribution of CNN-LSTM model's false negatives.

Table 8: Prediction quality on unseen Site A.

	Precision	Recall	F-measure	MCC
Bayes	0.35	0.37	0.36	0.31
RF	0.66	0.94	0.78	0.78
GBDT	0.65	0.89	0.75	0.74
LSTM	0.66	0.88	0.75	0.74
CNN-LSTM	0.93	0.94	0.94	0.93

new sites and in some cases, model training at new sites may not be possible due to strict business-sensitivity reasons. Therefore, we want to test if machine learning based disk failure models are unsuitable to a large degree for porting across data center sites? One can expect the operating conditions and workload characteristics to change across data center sites and hence, the disk failure prediction model may not work at all.

As expected, this is true if we simply try to train on one data center site and port it to another data center site (i.e., test on another unseen site) — the MCC score can drop significantly. However, we found that training on multiple data sites before testing on a new unseen data site provides reasonable accuracy. We tested on two unseen data center sites A and B, while training our model on the rest of the 62 sites (Table 8 shows results for site A; site B has similar results). Our results show that the prediction quality still remains reasonably high (e.g., >0.90 MCC score for a 10-day prediction horizon using CNN-LSTM model and SPL group features). We did not find a significant drop in prediction quality for any ML model; however, with some traditional ML models (RF and GBDT) the prediction quality does not remain high (more than 15% drop in some cases). Data center operators should be careful in porting ML-based prediction models as-is across sites without sufficiently training on multiple sites and should prefer CNN-LSTM models if portability is a requirement.

Is the prediction model effective at different prediction horizon (lead time)? To test this, we plotted MCC values for different ML models at different lead times (2-15 days). Figure 12 presents MCC scores of Bayes, RF, GBDT, LSTM, and CNN-LSTM for the SPL group, when the prediction horizon is 2 days, 5 days, 10 days and 15 days. As expected, the prediction quality indeed goes
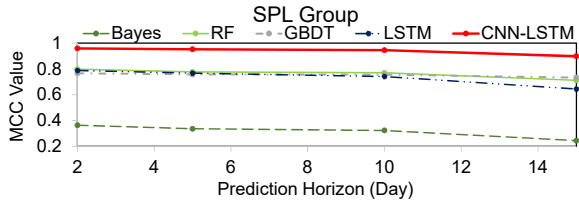
Figure 12: MCC scores of all ML models with SPL group features for different lengths of prediction horizon.
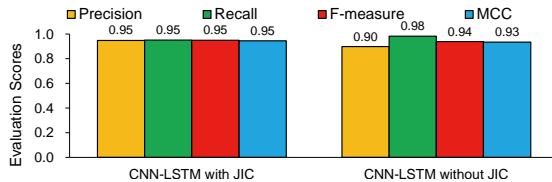

Figure 13: Prediction quality comparison among all features with and without J-Index classification (CNN-LSTM model on SPL group features).

down with increasing prediction horizon window (the MCC score for a 15-day window is 0.89), but the rate of decrease is not steep for any model — SPL group feature based ML models are effective even at sufficiently large prediction horizons.

Does J-Index classification for feature selection degrade the overall prediction accuracy compared to models trained with all features? Recall that we employed J-Index classification choosing the features (different performance and location metrics) for training our models. We compared the prediction quality for models using all the features (Figure 13). Our results show that manually selecting a subset of features using J-Index provides similar quality results, although it does affect the precision and recall trade-offs slightly. This notable observation suggests that data center operators can use J-Index to manage the storage overhead of storing attributes from thousands of disks without risking the prediction quality significantly.

6 Related Work

To the best of our knowledge, prior works do not consider all three types of data: SMART, performance, and location data for failure prediction. Instead, previous works rely only on SMART attributes [4, 19, 36, 41, 59, 79, 87]. We analyze large-scale field data collected from one of the biggest e-commerce sites, while most of the previous works propose prediction methods based on the publicly available Backblaze data [4, 5, 7, 9, 62, 80]. Also, the datasets analyzed were of limited in size, types of vendors, and were often closed-source [8, 9, 24, 28, 30, 31, 36, 37, 41, 51, 53, 58–60, 77, 83, 85, 88, 89, 92].

Much of previous work with disk failure prediction is limited to the detection of incipient failures

[9, 41, 59, 67, 84, 85]. Although Lima *et al.* [23] proposed an approach to predict disk failures in long- and short-term, they are also limited to SMART attributes. Studies by Sandeep *et al.* [25, 26, 78] enable a qualitative understanding of factors that affect disk drive reliability. Yang *et al.* [90] and Gerry Cole [21] both achieve reliability predictions based on accelerated life tests. In addition, non-parametric statistical tests [58], Markov Models [24, 92], and Mahalanobis distance [85] have been proposed to predict disk failures. Hughes *et al.* [41] applied the multivariate rank-sum test and achieved a 60% failure detection rate (FDR).

In our study, we focus on HDDs, and some previous works have focused on solid-state drives (SSDs). Three typical studies of SSDs are based on data collected by Facebook [57], Google [77], and Alibaba Cloud [88]. Furthermore, Grupp *et al.* [33] examined the reliability of flash memory. Ouyang *et al.* [61] studied programmable SSD controllers at a web services company. A number of studies by Cai *et al.* [11–18] explored different patterns of Multi-Level Cell (MLC) flash chip failure. Ma *et al.* [53] found the accumulation of reallocated sectors would deteriorate disk reliability. Narayanan *et al.* [60] proposed machine learning based approaches to answer what, when and why of SSD failures.

Overall, few studies have separately employed ML [4, 36, 54, 79] and DNN techniques [4, 23] to predict disk failures. Our work explores and compares three classic ML methods with two DNNs using six feature groups to predict disk failures. This kind of extensive analysis helps us derive insights such as there is no need to employ complex DNNs when only SMART data are available. In fact, we are also the first to demonstrate the cross-site portability of different models.

7 Conclusion

We conducted a field study of HDDs based on a large-scale dataset collected from a leading e-commerce production data center, including SMART attributes, performance metrics, and location markers. We discover that performance metrics are good indicators of disk failures. We also found that location markers can improve the accuracy of disk failure prediction. Lastly, we trained machine learning models including neural network models to predict disk failures with 0.95 F-measure and 0.95 MCC for 10 days prediction horizon.

Acknowledgement

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, volume 16, pages 265–283, 2016.

[2] Abdulaziz M Alayba, Vasile Palade, Matthew England, and Rahat Iqbal. A combined CNN and LSTM model for arabic sentiment analysis. In *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*, pages 179–191. Springer, 2018.

[3] Bruce Allen. Monitoring hard disks with SMART. *Linux Journal*, (117):74–77, 2004.

[4] Preethi Anantharaman, Mu Qiao, and Divyesh Jadav. Large scale predictive analytics for hard disk remaining useful life estimation. In *Proceedings of the 2018 IEEE International Congress on Big Data (BigData Congress)*, pages 251–254. IEEE, 2018.

[5] Nicolas Aussel, Samuel Jaulin, Guillaume Gandon, Yohan Petetin, Eriza Fazli, and Sophie Chabridon. Predictive models of hard drive failures based on operational data. In *Proceedings of the 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 619–625. IEEE, 2017.

[6] Viv Bewick, Liz Cheek, and Jonathan Ball. Statistics review 13: receiver operating characteristic curves. *Critical care*, 8(6):508, 2004.

[7] Shivam Bhardwaj, Akshay Saxena, and Achal Nayyar. Exploratory data analysis on hard drive failure statistics and prediction. *International Journal*, 6(6), 2018.

[8] Richard Black, Austin Donnelly, Dave Harper, Aaron Ogus, and Anthony Rowstron. Feeding the pelican: Using archival hard drives for cold storage racks. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[9] Mirela Madalina Botezatu, Ioana Giurgiu, Jasmina Bogojeska, and Dorothea Wiesmann. Predicting disk replacement towards reliable data centers. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 39–48, 2016.

[10] Sabri Boughorbel, Fethi Jarray, and Mohammed El-Anbari. Optimal classifier for imbalanced data using matthews correlation coefficient metric. *PloS one*, 12(6):e0177678, 2017.

[11] 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.

[12] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 521–526. EDA Consortium, 2012.

[13] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Threshold voltage distribution in MLC NAND flash memory: Characterization, analysis, and modeling. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1285–1290. IEEE, 2013.

[14] Yu Cai, Yixin Luo, Erich F Haratsch, Ken Mai, and Onur Mutlu. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 551–563. IEEE, 2015.

[15] Yu Cai, Onur Mutlu, Erich F Haratsch, and Ken Mai. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *Proceedings of the 31st International Conference on Computer Design (ICCD)*, pages 123–130. IEEE, 2013.

[16] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Crista, Osman S Unsal, and Ken Mai. Error analysis and retention-aware error management for NAND flash memory. *Intel Technology Journal*, 17(1), 2013.

[17] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Cristal, Osman S Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *Proceedings of the 30th International Conference on Computer Design (ICCD)*, pages 94–101. IEEE, 2012.

[18] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Osman Unsal, Adrian Cristal, and Ken Mai. Neighbor-cell assisted error correction for MLC NAND flash memories. In *ACM SIGMETRICS Performance Evaluation Review (SIGMETRICS)*, volume 42, pages 491–504. ACM, 2014.

[19] Iago C Chaves, Manoel Rui P de Paula, Lucas GM Leite, Joao Paulo P Gomes, and Javam C Machado.

Hard disk drive failure prediction method based on a Bayesian network. In *Proceedings of the 2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2018.

[20] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 13(4):359–394, 1999.

[21] Gerry Cole. Estimating drive reliability in desktop computers and consumer electronics systems. *Seagate Technology Paper TP*, 338, 2000.

[22] Jesse Davis and Mark Goadrich. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[23] Fernando Dione dos Santos Lima, Gabriel Maia Rocha Amaral, Lucas Goncalves de Moura Leite, João Paulo Pordeus Gomes, and Javam de Castro Machado. Predicting failures in hard drives with LSTM networks. In *Proceedings of the 2017 Brazilian Conference on Intelligent Systems (BRACIS)*, pages 222–227. IEEE, 2017.

[24] Ben Eckart, Xin Chen, Xubin He, and Stephen L Scott. Failure prediction models for proactive fault tolerance within storage systems. In *Proceedings of the 2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2008.

[25] Jon G Elerath and Sandeep Shah. Disk drive reliability case study: dependence upon head flyheight and quantity of heads. In *Proceedings of the 2003 Annual Reliability and Maintainability Symposium (RAMS)*, pages 608–612. IEEE, 2003.

[26] Jon G Elerath and Sandeep Shah. Server class disk drives: how reliable are they? In *Proceedings of the 2004 Annual Reliability and Maintainability Symposium (RAMS)*, pages 151–156. IEEE, 2004.

[27] Ronen Fluss, David Faraggi, and Benjamin Reiser. Estimation of the Youden Index and its associated cutoff point. *Biometrical Journal: Journal of Mathematical Methods in Biosciences*, 47(4):458–472, 2005.

[28] Daniel Ford, François Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. 2010.

[29] Jerome H Friedman. Stochastic gradient boosting. *Computational Statistics and Data Analysis*, 38(4):367–378, 2002.

[30] Peter Garraghan, Paul Townend, and Jie Xu. An empirical failure-analysis of a large-scale cloud computing environment. In *IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 113–120. IEEE, 2014.

[31] Moises Goldszmidt. Finding soon-to-fail disks in a haystack. In *HotStorage*, 2012.

[32] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5-6):602–610, 2005.

[33] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, pages 2–2. USENIX Association, 2012.

[34] Antonio Gulli and Sujit Pal. *Deep Learning with Keras*. Packt Publishing Ltd, 2017.

[35] Chongomweru Halimu, Asem Kasem, and SH Newaz. Empirical comparison of area under roc curve (AUC) and matthews correlation coefficient (MCC) for evaluating machine learning algorithms on imbalanced datasets for binary classification. In *Proceedings of the 3rd International Conference on Machine Learning and Soft Computing*, pages 1–6. ACM, 2019.

[36] Greg Hamerly, Charles Elkan, et al. Bayesian approaches to failure prediction for disk drives. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML)*, volume 1, pages 202–209, 2001.

[37] 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)*, pages 263–276, 2016.

[38] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[39] George Hripcsak and Adam S Rothschild. Agreement, the F-measure, and reliability in information retrieval. *Journal of the American Medical Informatics Association*, 12(3):296–298, 2005.

[40] Song Huang, Song Fu, Quan Zhang, and Weisong Shi. Characterizing disk failures with quantified disk degradation signatures: An early experience.

In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC)*, pages 150–159. IEEE, 2015.

[41] Gordon F Hughes, Joseph F Murray, Kenneth Kreutz-Delgado, and Charles Elkan. Improved disk-drive failure warnings. *IEEE transactions on reliability*, 51(3):350–357, 2002.

[42] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *ACM Transactions on Storage (TOS)*, 4(3):7, 2008.

[43] Giuseppe Jurman, Samantha Riccadonna, and Cesare Furlanello. A comparison of MCC and cen error measures in multi-class prediction. *PloS one*, 7(8):e41882, 2012.

[44] Saurabh Kadekodi, KV Rashmi, and Gregory R Ganger. Cluster storage systems gotta have heart: improving storage efficiency by exploiting disk-reliability heterogeneity. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, pages 345–358, 2019.

[45] Ji-Hyun Kim. Estimating classification error rate: Repeated cross-validation, repeated hold-out and bootstrap. *Computational statistics and data analysis*, 53(11):3735–3745, 2009.

[46] Andy Klein. What SMART stats tell us about hard drives. https://www.backblaze.com/blog/what-smart-stats-indicate-hard-drive-failures/, October 2016.

[47] Andy Klein. Backblaze hard drive stats for 2017. https://www.backblaze.com/blog/hard-drive-stats-for-2017/, February 2018.

[48] Andy Klein. Backblaze hard drive stats for 2018. https://www.backblaze.com/blog/hard-drive-stats-for-2018/, January 2019.

[49] Andy Klein. Backblaze hard drive stats Q3 2019. https://www.backblaze.com/blog/backblaze-hard-drive-stats-q3-2019/, November 2019.

[50] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 14, pages 1137–1145, 1995.

[51] Jing Li, Xinpu Ji, Yuhan Jia, Bingpeng Zhu, Gang Wang, Zhongwei Li, and Xiaoguang Liu. Hard

drive failure prediction using classification and regression trees. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 383–394. IEEE, 2014.

[52] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomForest. *R news*, 2(3):18–22, 2002.

[53] Ao Ma, Rachel Traylor, Fred Douglis, Mark Chamness, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. RAIDShield: characterizing, monitoring, and proactively protecting against disk failures. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, volume 11, page 17, 2015.

[54] Farzaneh Mahdisoltani, Ioan Stefanovici, and Bianca Schroeder. Proactive error prediction to improve storage system reliability. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC). Santa Clara, CA*, pages 391–402, 2017.

[55] Ioannis Manousakis, Sriram Sankar, Gregg McKnight, Thu D Nguyen, and Ricardo Bianchini. Environmental conditions and disk reliability in free-cooled datacenters. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 53–65, 2016.

[56] Brian S Merrow. Vibration isolation within disk drive testing systems, November 6 2012. US Patent 8,305,751.

[57] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In *ACM SIGMETRICS Performance Evaluation Review*, volume 43, pages 177–190. ACM, 2015.

[58] Joseph F Murray, Gordon F Hughes, and Kenneth Kreutz-Delgado. Hard drive failure prediction using non-parametric statistical methods. In *Proceedings of the International Conference on Artificial Neural Networks (ICANN)*, 2003.

[59] Joseph F Murray, Gordon F Hughes, and Kenneth Kreutz-Delgado. Machine learning methods for predicting failures in hard drives: A multiple-instance application. *Journal of Machine Learning Research*, 6(May):783–816, 2005.

[60] 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)*, page 7. ACM, 2016.

[61] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: software-defined flash for web-scale internet storage systems. In *ACM SIGARCH Computer Architecture News (ASPLOS)*, volume 42, pages 471–484. ACM, 2014.

[62] Jehan-François Pâris, SJ Thomas Schwarz, SJ Ahmed Amer, and Darrell DE Long. Protecting RAID arrays against unexpectedly high disk failure rates. In *Proceedings of the IEEE 20th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 68–75. IEEE, 2014.

[63] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*, 2013.

[64] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[65] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, volume 7, pages 17–23, 2007.

[66] David Martin Powers. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. Bioinfo Publications. 2011.

[67] Lucas P Queiroz, Francisco Caio M Rodrigues, Joao Paulo P Gomes, Felipe T Brito, Iago C Chaves, Manoel Rui P Paula, Marcos R Salvador, and Javam C Machado. A fault detection method for hard disk drives based on mixture of Gaussians and nonparametric statistics. *IEEE Transactions on Industrial Informatics*, 13(2):542–550, 2017.

[68] Oren Rippel, Jasper Snoek, and Ryan P Adams. Spectral representations for convolutional neural networks. In *Advances in neural information processing systems*, pages 2449–2457, 2015.

[69] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

[70] Juan D Rodriguez, Aritz Perez, and Jose A Lozano. Sensitivity analysis of K-fold cross validation in prediction error estimation. *IEEE transactions on pattern analysis and machine intelligence (TPAMI)*, 32(3):569–575, 2010.

[71] Kunal Roy, Supratik Kar, and Rudra Narayan Das. *Understanding the basics of QSAR for applications in pharmaceutical sciences and risk assessment*. Academic press, 2015.

[72] Tara N Sainath, Oriol Vinyals, Andrew Senior, and Haşim Sak. Convolutional, long short-term memory, fully connected deep neural networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4580–4584. IEEE, 2015.

[73] Sriram Sankar, Mark Shaw, and Kushagra Vaid. Impact of temperature on hard disk drive reliability in large datacenters. In *Proceedings of IEEE/IFIP the 41st International Conference on Dependable Systems and Networks (DSN)*, pages 530–537. IEEE, 2011.

[74] Enrique F Schisterman, Neil J Perkins, Aiyi Liu, and Howard Bondell. Optimal cut-point and its corresponding Youden Index to discriminate individuals using pooled blood samples. *Epidemiology*, pages 73–81, 2005.

[75] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding latent sector errors and how to protect against them. *ACM Transactions on storage (TOS)*, 6(3):9, 2010.

[76] Bianca Schroeder and Garth A Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, volume 7, pages 1–16, 2007.

[77] 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)*, pages 67–80, 2016.

[78] Sandeep Shah and Jon G Elerath. Reliability analysis of disk drive failure mechanisms. In *Proceedings of the 2005 Annual Reliability and Maintainability Symposium (RAMS)*, pages 226–231. IEEE, 2005.

[79] Jing Shen, Jian Wan, Se-Jung Lim, and Lifeng Yu. Random-forest-based failure prediction for hard disk drives. *International Journal of Distributed Sensor Networks*, 14(11), 2018.

[80] Chuan-Jun Su and Shi-Feng Huang. Real-time big data analytics for hard disk drive predictive maintenance. *Computers and Electrical Engineering*, 71:93–101, 2018.

[81] Yoshimasa Tsuruoka and Jun'ichi Tsujii. Boosting precision and recall of dictionary-based protein name recognition. In *Proceedings of the ACL 2003 workshop on Natural language processing in biomedicine-Volume 13*, pages 41–48. Association for Computational Linguistics, 2003.

[82] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)*, pages 193–204. ACM, 2010.

[83] Guosai Wang, Lifei Zhang, and Wei Xu. What can we learn from four years of data center hardware failures? In *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36. IEEE, 2017.

[84] Yu Wang, Eden WM Ma, Tommy WS Chow, and Kwok-Leung Tsui. A two-step parametric method for failure prediction in hard disk drives. *IEEE Transactions on industrial informatics*, 10(1):419–430, 2014.

[85] Yu Wang, Qiang Miao, Eden WM Ma, Kwok-Leung Tsui, and Michael G Pecht. Online anomaly detection for hard disk drives based on Mahalanobis distance. *IEEE Transactions on Reliability*, 62(1):136–145, 2013.

[86] Zhou Wang and Alan C Bovik. Mean squared error: Love it or leave it? a new look at signal fidelity measures. *IEEE signal processing magazine*, 26(1):98–117, 2009.

[87] Jiang Xiao, Zhuang Xiong, Song Wu, Yusheng Yi, Hai Jin, and Kan Hu. Disk failure prediction in data centers via online learning. In *Proceedings of the 47th International Conference on Parallel Processing*, page 35. ACM, 2018.

[88] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and actions: What we learned from 10K SSD-related storage system failures. In *Proceedings of 2019 USENIX Annual Technical Conference (ATC)*, pages 961–976, 2019.

[89] Yong Xu, Kaixin Sui, Randolph Yao, Hongyu Zhang, Qingwei Lin, Yingnong Dang, Peng Li, Keceng Jiang, Wenchi Zhang, Jian-Guang Lou, et al. Improving service availability of cloud systems by predicting disk error. In *Proceedings of 2018 USENIX Annual Technical Conference (ATC)*, pages 481–494, 2018.

[90] Jimmy Yang and Feng-Bin Sun. A comprehensive review of hard-disk drive reliability. In *Proceedings of Annual Reliability and Maintainability Symposium (RAMS)*, pages 403–409. IEEE, 1999.

[91] Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. Stochastic gradient boosted distributed decision trees. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 2061–2064. ACM, 2009.

[92] Ying Zhao, Xiang Liu, Siqing Gan, and Weimin Zheng. Predicting disk failures with HMM-and HSMM-based approaches. In *Proceedings of the 2010 Industrial Conference on Data Mining (ICDM)*, pages 390–404, 2010.

An Empirical Guide to the Behavior and Use of Scalable Persistent Memory

Jian Yang[*][†], Juno Kim[†], Morteza Hoseinzadeh[†], Joseph Izraelevitz[§], and Steven Swanson[†]

{jianyang, juno, mhoseinzadeh, swanson}@eng.ucsd.edu[†]   joseph.izraelevitz@colorado.edu[§]
[†]UC San Diego   [§]University of Colorado, Boulder

Abstract

After nearly a decade of anticipation, scalable nonvolatile memory DIMMs are finally commercially available with the release of Intel's Optane DIMM. This new nonvolatile DIMM supports byte-granularity accesses with access times on the order of DRAM, while also providing data storage that survives power outages.

Researchers have not idly waited for real nonvolatile DIMMs (NVDIMMs) to arrive. Over the past decade, they have written a slew of papers proposing new programming models, file systems, libraries, and applications built to exploit the performance and flexibility that NVDIMMs promised to deliver. Those papers drew conclusions and made design decisions without detailed knowledge of how real NVDIMMs would behave or how industry would integrate them into computer architectures. Now that Optane NVDIMMs are actually here, we can provide detailed performance numbers, concrete guidance for programmers on these systems, reevaluate prior art for performance, and reoptimize persistent memory software for the real Optane DIMM.

In this paper, we explore the performance properties and characteristics of Intel's new Optane DIMM at the micro and macro level. First, we investigate the basic characteristics of the device, taking special note of the particular ways in which its performance is peculiar relative to traditional DRAM or other past methods used to emulate NVM. From these observations, we recommend a set of best practices to maximize the performance of the device. With our improved understanding, we then explore and reoptimize the performance of prior art in application-level software for persistent memory.

1 Introduction

Over the past ten years, researchers have been anticipating the arrival of commercially available, scalable non-volatile main memory (NVMM) technologies that provide "byte-addressable" storage that survives power outages. With the arrival of Intel's Optane DC Persistent Memory Module (which we refer to as Optane DIMMs), we can start to understand the real capabilities, limitations, and characteristics of these memories and start designing systems to fully leverage them.

We have characterized the performance and behavior of Optane DIMMs using a wide range of microbenchmarks, benchmarks, and applications. The data we have collected demonstrate that many of the assumptions that researchers

have made about how NVDIMMs would behave and perform are incorrect. The widely expressed expectation was that NVDIMMs would have behavior that was broadly similar to DRAM-based DIMMs but with lower performance (i.e., higher latency and lower bandwidth). These assumptions are reflected in the methodology that research studies used to emulate NVDIMMs, which include specialized hardware platforms [21], software emulation mechanisms [12,32,36,43,47], exploiting NUMA effects [19,20,29], and simply pretending DRAM is persistent [8,9,38].

We have found the actual behavior of Optane DIMMs to be more complicated and nuanced than the "slower, persistent DRAM" label would suggest. Optane DIMM performance is much more strongly dependent on access size, access type (read vs. write), pattern, and degree of concurrency than DRAM performance. Furthermore, Optane DIMM's persistence, combined with the architectural support that Intel's latest processors provide, leads to a wider range of design choices for software designers.

This paper presents a detailed evaluation of the behavior and performance of Optane DIMMs on microbenchmarks and applications and provides concrete, actionable guidelines for how programmers should tune their programs to make the best use of these new memories. We describe these guidelines, explore their consequences, and demonstrate their utility by using them to guide the optimization of several NVMM-aware software packages, noting that prior methods of emulation have been unreliable.

The paper proceeds as follows. Section 2 provides architectural details on our test machine and the Optane DIMM. Section 3 presents experiments on basic microarchitectural parameters, and Section 4 focuses on how Optane DIMM is different from DRAM and other emulation techniques. Section 5 uses these results to posit best practices for programmers on the Optane DIMM. In this section, we first justify each guideline with a microbenchmark demonstrating the root cause. We then present one or more case studies where guideline influences a previously proposed Optane-aware software system. Section 6 provides discussion as to how our guidelines extend to future generations of NVM. Section 7 describes related work in this space, and Section 8 concludes.

2 Background and Methodology

In this section, we provide background on Intel's Optane DIMM, describe the test system, and then describe the configurations we use throughout the rest of the paper.
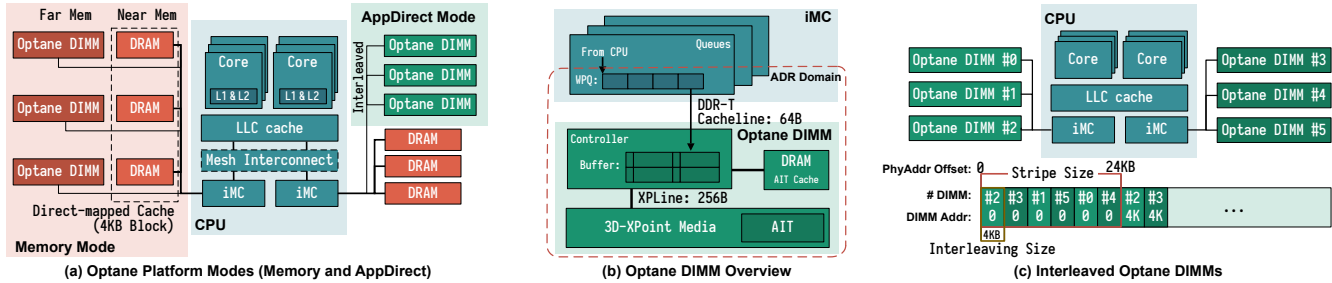
---

[*]Now at Google

Figure 1: Overview of (a) Optane platform, (b) Optane DIMM and (c) how Optane memories interleave across channels
Optane DIMM can work as either volatile far memory with DRAM as cache (memory mode), or persistent memory with DRAM as main memory (AppDirect mode).

2.1 Optane Memory

The Optane DIMM is the first scalable, commercially available NVDIMM. Compared to existing storage devices (including the Optane SSDs) that connect to an external interface such as PCIe, the Optane DIMM has lower latency, higher read bandwidth, and presents a memory address-based interface instead of a block-based NVMe interface. Compared to DRAM, it has higher density and persistence. At its debut, the Optane DIMM is available in three different capacities: 128, 256, and 512 GB.

2.1.1 Intel's Optane DIMM

Like traditional DRAM DIMMs, the Optane DIMM sits on the memory bus, and connects to the processor's integrated memory controller (iMC) (Figure 1(a)). Intel's Cascade Lake processors are the first (and only) CPUs to support Optane DIMM. On this platform, each processor contains one or two processor dies which comprise separate NUMA nodes. Each processor die has two iMCs, and each iMC supports three channels. Therefore, in total, a processor die can support a total of six Optane DIMMs across its two iMCs.

To ensure persistence, the iMC sits within the *asynchronous DRAM refresh (ADR)* domain — Intel's ADR feature ensures that CPU stores that reach the ADR domain will survive a power failure (i.e., will be flushed to the NVDIMM within the hold-up time, $< 100~\mu s$) [42]. The iMC maintains read and write pending queues (RPQs and WPQs) for each of the Optane DIMMs (Figure 1(b)), and the ADR domain includes WPQs. Once data reaches the WPQs, the ADR ensures that the iMC will flush the updates to 3D-XPoint media on power failure. The ADR domain does not include the processor caches, so stores are only persistent once they reach the WPQs.

The iMC communicates with the Optane DIMM using the DDR-T interface in cache-line (64-byte) granularity, which shares a mechanical and electrical interface with DDR4 but uses a different protocol that allows for asynchronous command and data timing.

Memory accesses to the NVDIMM (Figure 1(b)) arrive first at the on-DIMM controller (referred as *XPController* in this paper), which coordinates access to the Optane media. Similar to SSDs, the Optane DIMM performs an internal address translation for wear-leveling and bad-block management, and maintains an *address indirection table* (AIT) for this translation [7].

After address translation, the actual access to storage media occurs. As the 3D-XPoint physical media access granularity is 256 bytes (referred as *XPLine* in this paper), the XPController translates smaller requests into larger 256-byte accesses, causing write amplification as small stores become read-modify-write operations. The XPController has a small write-combining buffer (referred as *XPBuffer* in this paper), to merge adjacent writes.

It is important to note that all updates that reach the XP-Buffer are already persistent since XPBuffer resides within the ADR. Consequently, the NVDIMM can buffer and merge updates regardless of ordering requirements that the program specifies with memory fences.

2.1.2 Operation Modes

Optane DIMMs can operate in two modes (Figure 1(a)): Memory and App Direct.

Memory mode uses Optane to expand main memory capacity without persistence. It combines an Optane DIMM with a conventional DRAM DIMM on the same memory channel that serves as a direct-mapped cache for the NVDIMM. The cache block size is 64 B, and the CPU's memory controller manages the cache transparently. The CPU and operating system simply see the Optane DIMM as a larger (volatile) portion of main memory.

App Direct mode provides persistence and does not use a DRAM cache. The Optane DIMM appears as a separate, persistent memory device. A file system or other management layer provides allocation, naming, and access to persistent data.

In both modes, Optane memory can be (optionally) interleaved across channels and DIMMs (Figure 1(c)). On our

platform, the only supported interleaving size is 4 kB, which ensures that accesses to a single page fall into a single DIMM. With six DIMMs, an access larger than 24 kB will access all the DIMMs.

2.1.3 ISA Support

In App Direct mode, applications and file systems can access the Optane DIMMs with CPU instructions. The extended Instruction Set Architecture (ISA) offers programmers a number of options to control store ordering.

Applications access the Optane DIMM's content using store instructions, and those stores will, eventually, become persistent. The cache hierarchy, however, can reorder stores, making recovery after a crash challenging [12, 28, 33, 40, 49]. Current Intel ISA provides `clflush` and `clflushopt` instructions to flush cache lines back to memory with `clflushopt` having weaker ordering constraints, and `clwb` can write back (but not evict) cache lines. Alternatively, software can use non-temporal stores (e.g., `ntstore`) to bypass the cache hierarchy and write directly to memory. All these instructions are non-blocking, so the program must issue an `sfence` to ensure that a previous cache flush, cache write back, or non-temporal store is complete and persistent.

2.2 System Description

We performed our experiments on a dual-socket evaluation platform provided by Intel Corporation. The CPUs are 24-core Cascade Lake engineering samples with the similar spec as the previous-generation Xeon Platinum 8160. Each CPU has two iMCs and six memory channels (three channels per iMC). A 32 GB Micron DDR4 DIMM and a 256 GB Intel Optane DIMM are attached to each of the memory channels. Thus the system has 384 GB (2 socket × 6 channel × 32 GB/DIMM) of DRAM, and 3 TB (2 socket × 6 channel × 256 GB/DIMM) of NVMM. Our machine runs Fedora 27 with Linux kernel version 4.13.0 built from source.

2.3 Experimental Configurations

As the Optane DIMM is both persistent and byte-addressable, it can fill the role of either a main memory device (i.e., replacing DRAM) or a persistent device (i.e., replacing disk). In this paper, we focus on the persistent usage, and defer discussion on how our results apply to using Optane DIMM as volatile memory to Section 6.

Linux manages persistent memory by creating `pmem` namespaces over a contiguous span of physical memory. A namepace can be backed by interleaved or non-interleaved Optane memory, or emulated persistent memory backed by DRAM. In this study, we configure Optane memory in App Direct mode and create a namespace for each type of memory.

Our baseline (referred as *Optane*) exposes six Optane DIMMs from the same socket as a single interleaved names-

pace. In our experiments, we used local accesses (i.e., from the same NUMA node) as the baseline to compare with one or more other configurations, such as access to Optane memory on the remote socket (*Optane-Remote*) or DRAM on the local or remote socket (*DRAM* and *DRAM-Remote*). To better understand the raw performance of Optane memory without interleaving, we also create a namespace consisting of a single Optane DIMM and denote it as *Optane-NI*.

3 Performance Characterization

In this section, we measure Optane's performance along multiple axes to provide the intuition and data that programmers and system designers will need to effectively utilize Optane. We find that Optane's performance characteristics are surprising in many ways, and more complex than the common assumption that Optane behaves like slightly-slower DRAM.

3.1 LATTester

Characterizing Optane memory is challenging for two reasons. First, the underlying technology has major differences from DRAM but publicly-available documentation is scarce. Secondly, existing tools measure memory performance primarily as a function of locality and access size, but we have found that Optane performance depends strongly on memory interleaving and concurrency as well. Persistence adds an additional layer of complexity for performance measurements.

To fully understand the behavior of the Optane memory, we built a microbenchmark toolkit called LATTester. To accurately measure the CPU cycle count and minimize the impact from the virtual memory system, LATTester runs as a dummy file system in the kernel and accesses pre-populated (i.e., no page-faults) kernel virtual addresses of Optane DIMMs. LATTester also pins the kernel threads to fixed CPU cores and disables IRQ and cache prefetcher. In addition to simple latency and bandwidth measurements, LATTester collects a large set of hardware counters at both the CPU and NVDIMM.

Our investigation of Optane memory behavior proceeds in two phases. First, we performed a broad, systematic "sweep" over Optane configuration parameters including access patterns (random vs. sequential), operations (loads, stores, fences, etc.), access size, stride size, power budget, NUMA configuration, and address space interleaving. Using this data, we designed targeted experiments to investigate anomalies and verify or disprove our hypotheses about the underlying causes. Between our initial sweep and the follow-up tests, we collected over ten thousand data points. The program and dataset are available at https://github.com/NVSL/OptaneStudy.

3.2 Typical Latency

Read and write latencies are key memory technology parameters. We measured read latency by timing the average latency
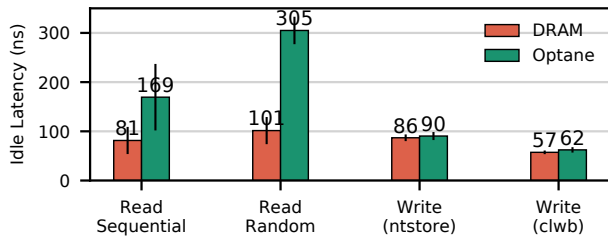
Figure 2: **Best-case latency** An experiment showing random and sequential read latency, as well as write latency using cached write with `clwb` and `ntstore` instructions. Error bars show one standard deviation.

for individual 8-byte load instructions to sequential and random memory addresses. To eliminate caching and queuing effects, we empty the CPU pipeline and issue a memory fence (`mfence`) between measurements (`mfence` serves the purpose of serialization for reading timestamps). For writes, we load the cache line into the cache and then measure the latency of one of two instruction sequences: a 64-bit store, a `clwb`, and an `mfence`; or a non-temporal store followed by an `mfence`.

These measurements reflect the load and store latency as seen by software rather than those of these underlying memory devices. For loads, the latency includes the delay from the on-chip interconnect, iMC, XPController and the actual 3D-XPoint media. Our results (Figure 2) show the read latency for Optane is 2×–3× higher than DRAM. We believe most of this difference is due to Optane's longer media latency. Optane memory is also more pattern-dependent than DRAM. The random-vs-sequential gap is 20% for DRAM but 80% for Optane memory, and this gap is a consequence of the XPBuffer. For stores, the memory store and fence instructions commit once the data reaches the ADR at the iMC, so both DRAM and Optane show a similar latency. Non-temporal stores are more expensive than writes with cache flushes (`clwb`).

In general, the latency variance for Optane is extremely small, save for an extremely small number of "outliers", which we investigate in the next section. The sequential read latencies for Optane DIMMs have higher variances, as the first cache line access loads the entire XPLine into XPBuffer, and the following three accesses read data in the buffer.

3.3 Tail Latency

Memory and storage system tail latency critically affects response times and worst-case behavior in many systems. In our tests, we observed a very consistent latency for loads and stores except a few "outliers", which increase in number for stores when accesses are concentrated in a "hot spot".

Figure 3 measures the relationship between tail latency and access locality. The graph shows the 99.9th, 99.99th, and maximal latencies as a function of hot spot size. We allocate a circular buffer with each hot spot size, and use a single thread to issue 20 million 64-byte writes. The number of outliers
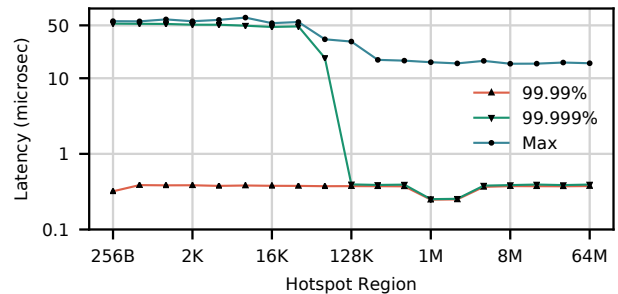

Figure 3: **Tail latency** An experiment showing the tail latency of writing to a small area of memory (hotspot) sequentially. Optane memory has rare "outliers" where a small number of writes take up to 50 μs to complete (an increase of 100× over the usual latency).

(especially for the ones over 50μs) reduces as the hotspot size increases and do not exist for DRAM. These spikes are rare (0.006% of the accesses), but their latency are 2 orders of magnitude higher than a common case Optane access. We suspect this effect is due to remapping for wear-leveling or thermal concerns, but we cannot be sure.

3.4 Bandwidth

Detailed bandwidth measurements are useful to application designers as they provide insight into how a memory technology will impact overall system throughput. First, we measured Optane and DRAM bandwidth for random and sequential reads and writes under different levels of concurrency.

Figure 4 shows the bandwidth achieved at different thread counts for sequential accesses with 256 B access granularity. We show loads and stores (`Write(ntstore)`), as well as cached writes with flushes (`Write(clwb)`). All experiments use AVX-512 instructions. The left-most graph plots performance for interleaved DRAM, while the center and right-most graphs plot performance for non-interleaved and interleaved Optane. In the non-interleaved measurements all accesses hit a single DIMM.

Figure 5 shows how performance varies with access size. The graphs plot aggregate bandwidth for random accesses of a given size. We use the best-performing thread count for each curve (given as "<load thread count>/<`ntstore` thread count>/<`store+clwb` thread count>" in the figure). Note that the best performing thread count for Optane(Read) varies with different access sizes for random accesses, where 16 threads show good performance consistently.

The data shows that DRAM bandwidth is both higher than Optane and scales predictably (and monotonically) with thread count until it saturates the DRAM's bandwidth, which is mostly independent of access size.

The results for Optane are wildly different. First, for a single DIMM, the maximal read bandwidth is 2.9× of the maxi-
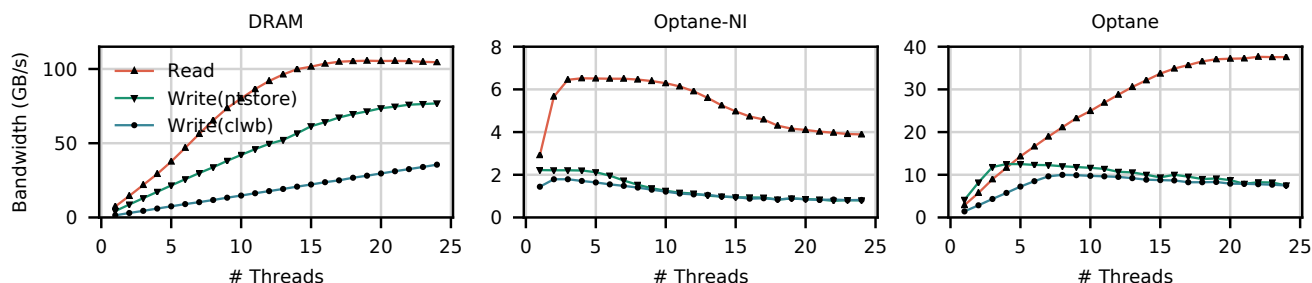
Figure 4: **Bandwidth vs. thread count** An experiment showing the maximal bandwidth as thread count increases (from left to right) on local DRAM, non-interleaved and interleaved Optane memory. All threads use a 256 B access size. (Note the difference in vertical scales).

mal write bandwidth (6.6 GB/s and 2.3 GB/s, respectively), where DRAM has a smaller gap (1.3×) between read and write bandwidth. Second, with the exception of interleaved reads, Optane performance is non-monotonic with increasing thread count. For the non-interleaved (i.e., single-DIMM) cases, performance peaks at between one and four threads and then tails off. Interleaving pushes the peak to twelve threads for store+clwb. We will return to the negative impact of rising thread count on performance in Section 5.1. Third, Optane bandwidth for random accesses under 256 B is poor. This "knee" corresponds to XPLine size. DRAM bandwidth does not exhibit a similar "knee" at 8 kB (the typical DRAM page size), because the cost of opening a page of DRAM is much lower than accessing a new page of Optane.

Interleaving (which spreads accesses across all six DIMMs) adds further complexity: Figure 4 (right) and Figure 5 (right) measure bandwidth across six interleaved NVDIMMs. Interleaving improves peak read and write bandwidth by 5.8× and 5.6×, respectively. These speedups match the number of DIMMs in the system and highlight the per-DIMM bandwidth limitations of Optane. The most striking feature of the graph is a dip in performance at 4 kB — this dip is an emergent effect caused by contention at the iMC, and it is maximized when threads perform random accesses close to the interleaving size. We further discuss this issue in Section 5.3.

4 Comparison to Emulation

Non-volatile memory research has been popular in recent years (e.g. [9, 18, 28, 31, 32, 36, 38, 40, 43, 49, 51, 54, 56, 60]). However, since scalable NVDIMMs have not been available, most of the NVM based systems have been evaluated on emulated NVM. Common ways to emulate NVM include adding delays to memory accesses in software [32, 36, 49], using software emulators [43, 47], using software simulation [12, 31, 40], using hardware emulators such as Intel's Persistent Memory Emulator Platform (PMEP) [21] to limit latency and bandwidth [54, 55, 60], using DRAM on a remote socket (DRAM-Remote) [19, 20, 29], underclocking DRAM [28] or just using plain DRAM [9, 34, 38, 56] or battery-backed DRAM [18].

Below, we compare these emulation techniques to real Optane using microbenchmarks and then provide a case study in how those differences can affect research results.

4.1 Microbenchmarks in Emulation

Figure 6 (left) shows the write latency/bandwidth curves for NVM emulation mechanisms (e.g. PMEP, DRAM-Remote, DRAM) in comparison to real Optane memory. Figure 6 (right) shows the bandwidth with respect to the number of reader/writer threads (all experiments use a fixed number of threads that give maximum bandwidth). Our PMEP configuration adds a 300 ns latency on load instructions and throttles write bandwidth at 1/8 of DRAM bandwidth as this configuration is the standard used in previous works [54, 55, 59, 60]. Note that PMEP is a specialized hardware platform, so its performance numbers are not directly comparable to the system we used in our other experiments.

The data in these figures shows that none of the emulation mechanisms captures the details of Optane's behavior — all methods deviate drastically from real Optane memory. They fail to capture Optane memory's preference for sequential accesses and read/write asymmetry, and give wildly inaccurate guesses for device latency and bandwidth.

4.2 Case Study: Optimizing RocksDB

The lack of emulation fidelity can have a dramatic effect on the performance of software. Results may be misleading, especially those based on simple DRAM. In this section, we revisit prior art in NVM programming in order to demonstrate that emulation is generally insufficient to capture the performance of Optane DIMMs and that future work should be validated on real hardware.

RocksDB [22] is a high-performance embedded key-value store, designed by Facebook and inspired by Google's LevelDB [16]. RocksDB's design is centered around the log-structured merge tree (LSM-tree), designed for block-based storage devices, which absorbs random writes and converts them to sequential writes to maximize disk bandwidth.
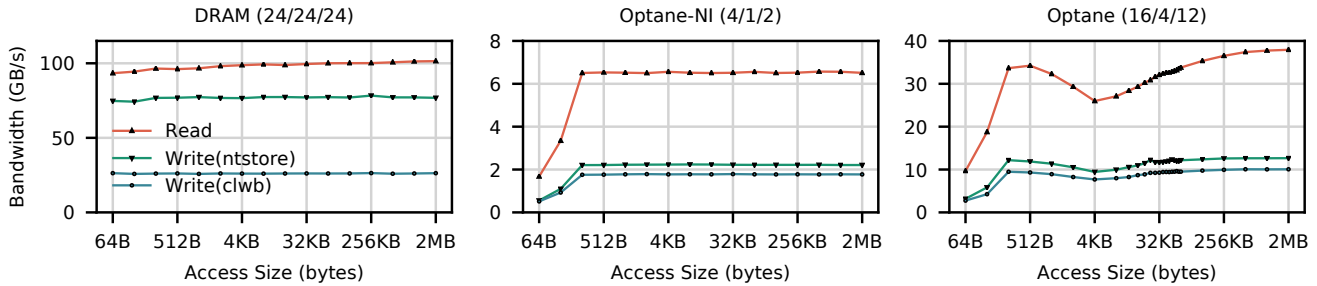
Figure 5: **Bandwidth over access size** An experiment showing maximal bandwidth over different access sizes on (from left to right) local DRAM, interleaved and non-interleaved Optane memory. Graph titles include the number of threads used in each experiment (`Read/Write(ntstore)/Write(clwb)`).
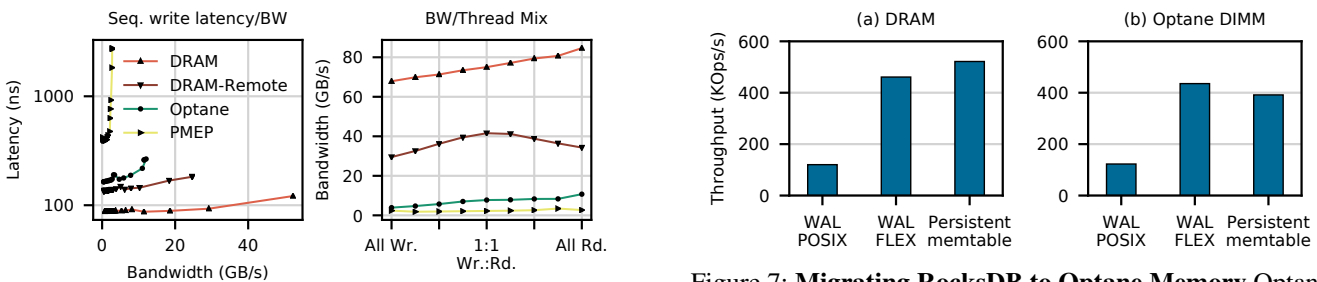

Figure 6: **Microbenchmarks under emulation** The emulation mechanisms used to evaluate many projects do not accurately capture the complexities of Optane performance.

A recent study [53] compared two strategies for adapting RocksDB to use persistent memory. The first used a fine-grained persistence approach to migrate RockDB's "memtable" to persistent memory, eliminating the need for a file-based write-ahead log. The second approach moved the write-ahead log to persistent memory and used a simpler acceleration technique called FLEX to improve logging performance. The study used DRAM as a stand-in for Optane, and found that fine-grained persistence offered 19% better performance.

We replicated these experiments on real Optane DIMMs. We used the RocksDB test `db_bench` on SET throughput with 20-byte key size and 100-byte value size, and sync'ed the database after each SET operation; the results are shown in Figure 7. With real Optane, the result is the opposite: FLEX performs better than fine-grained persistence by 10%. These results are not surprising given Optane memory's preference for sequential accesses and its problem with small random writes.

5   Best Practices for Optane DIMMs

Section 3 highlights the many differences between Optane and conventional storage and memory technologies, and Section 4 shows how these differences can manifest to invalidate macro-level results. These differences mean that existing intuitions

Figure 7: **Migrating RocksDB to Optane Memory** Optane memory is sufficiently different from DRAM to invert prior conclusions. Using a persistent memtable works best for DRAM emulating persistent memory, but on real Optane memory the conclusion is reversed.

about how to optimize software for disks and memory do not apply directly to Optane. This section distills the results of our characterization experiments into a set of four principles for how to build and tune Optane-based systems.

1. **Avoid random accesses smaller than 256 B.**

2. **Use non-temporal stores when possible for large transfers, and control cache evictions.**

3. **Limit the number of concurrent threads accessing an Optane DIMM.**

4. **Avoid NUMA accesses (especially read-modify-write sequences).**

Below, we describe the guidelines in detail, give examples on how to implement them, and provide case studies in their application.

5.1   Avoid small random accesses

Internally, Optane DIMMs update Optane contents at a 256 B granularity. This granularity, combined with a large internal store latency, means that smaller updates are inefficient since
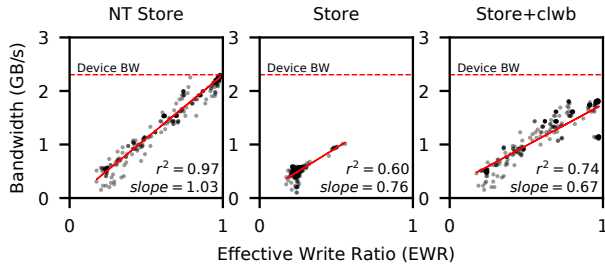
Figure 8: **Relationship between EWR and throughput on a single DIMM** Each dot represents an experiment with different access size, thread count and power budget configurations. Note the correlation between the metrics.

they require the DIMM to perform an internal read-modify-write operation causing write amplification. The less locality the accesses exhibit, the more severe the performance impact.

To characterize the impact of small stores, we performed two experiments. First, we quantify the inefficiency of small stores using a metric we have found useful in our study of Optane DIMMs. The *Effective Write Ratio(EWR)* is the ratio of bytes issued by the iMC divided by the number of bytes actually written to the 3D-XPoint media (as measured by the DIMM's hardware counters). EWR is the inverse of write amplification. EWR values below one indicate the Optane DIMM is operating inefficiently since it is writing more data internally than the application requested. The EWR can also be greater than one, due to write-combining at the XP-Buffer. In Memory Mode, DRAM caching may also introduce a higher EWR.

Figure 8 plots the strong correlation between EWR and effective device bandwidth for a single DIMM for all the measurements in our systematic sweep of Optane performance. Based on this relationship, we conclude that working to maximize EWR is a good way to maximize bandwidth.

In general, small stores exhibit EWR's less than one. For example, when using a single thread to perform non-temporal stores to random accesses, it achieves an EWR of 0.25 for 64-byte accesses and 0.98 for 256-byte accesses.

Notably, 256-byte updates are efficient, even though the iMC only issues 64 B to accesses the DIMM — the XPBuffer is responsible for buffering and combining 64 B accesses into 256 B internal writes. As a consequence, Optane DIMMs can efficiently handle small stores, if they exhibit sufficient locality. To understand how much locality is sufficient, we crafted an experiment to measure the size of the XPBuffer. First, we allocate a contiguous region of N XPLines. During each "round" of the experiment, we first update the first half (128 B) of each XPLine in turn. Then, we update the second half of each XPLine. We measured the EWR for each round. Figure 9 shows the results. Below $N = 64$ (that is, a region size of 16 kB), the EWR is near unity, suggesting that the accesses to the second halves are hitting in the XPBuffer. Above $N = 64$, write amplification jumps, indicating a sharp rise in the
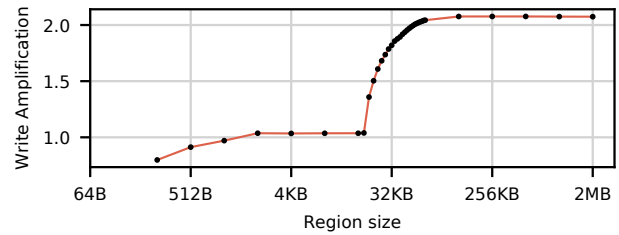

Figure 9: **Inferring XPBuffer capacity** The data shows that the Optane DIMM can use the XPBuffer to coalesce writes spread across up to 64 XPLines.
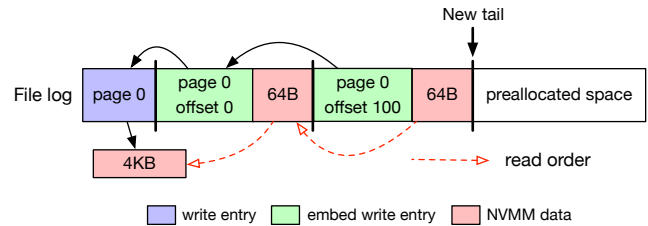

Figure 10: **NOVA-datalog mechanism** Sequentially embedding data along with metadata turns random writes into sequential writes. This figure illustrates how NOVA-datalog appends two 64 B random writes (at 0 and 100, respectively) into the log of a 4 kB file.

miss rate. This result implies the XPBuffer is approximately 16 kB in size. Further experiments demonstrate that reads also compete for space in the XPBuffer.

Together these results provide a specific guidance for maximizing Optane store efficiency: Avoid small stores, but if that is not possible, limit the working set to 16 kB per Optane DIMM.

5.1.1 Case Study: RocksDB

The correlation between EWR bandwidth explains the results for RocksDB seen in Section 4 and Figure 7. The persistent memtable resulted in many small stores with poor locality, leading to a low EWR of 0.434. In contrast, the FLEX-based optimization of WAL uses sequential (and larger) stores, resulting in an EWR of 0.999.

5.1.2 Case Study: The NOVA filesystem

NOVA [54, 55] is a log-structured, NVMM file system that maintains a separate log for each file and directory, and uses copy-on-write for file data updates to ensure data consistency. The original NOVA studies used emulated NVMM for their evaluations, so NOVA has not been tuned for Optane.

The original NOVA design has two characteristics that degrade performance on Optane. First, the log entries NOVA appends for each metadata update are small – 40-64 B, and since NOVA uses many logs, log updates exhibit little locality,
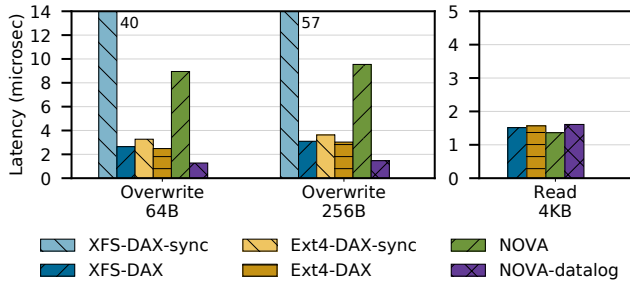
Figure 11: **File IO latency** NOVA-datalog significantly speeds up small random writes, but adds a slight overhead in the read path. Like NOVA and unlike `Ext4` or `XFS`, NOVA-datalog still provides data consistency.

especially when the file system is under load. Second, NOVA uses copy-on-write to 4 kB pages for file data updates, resulting in useless stores. This inefficiency occurs regardless of the underlying memory technology, but Optane's poor store performance exacerbates its effect.

We address both problems by increasing the size of log entries and avoiding some copy-on-write operations. Our modified version of NOVA – *NOVA-datalog* – embeds write data for sub-page writes into the log (Figure 10). Unlike the log's normal *write entry*, which contains a pointer to a new copy-on-write 4 kB page and its offset within the file, an *embed write entry* contains a page offset, an address within the page, and is followed by the actual contents of the write. This optimization requires several subsidiary changes to the original NOVA design. In particular, NOVA must merge sub-page updates into the target page before memory-mapping or reading the file.

Figure 11 shows the latencies of random overwrites and reads for three file systems with different modes. For `XFS-DAX` and `Ext4-DAX` (the two NVM-based file systems included in Linux), we measured both normal `write` and `write` followed by `fsync` (labeled with "-sync"). NOVA-datalog improves write performance significantly compared to the original design (by 7×, 6.5× for 64 byte, 256 byte writes, respectively) and meets (for 256 B) or exceeds (for 64 B) performance for the other file systems (which do not provide data consistency). Read latency increases slightly compared to the original NOVA. EWR measurements generally mirror the performance gains.

5.2 Use non-temporal stores for large writes

The choice of how programs perform and order updates to Optane has a large impact on performance. When writing to persistent memory, programmers have several options. After a regular `store`, programmers can either evict (`clflush`, `clflushopt`) or write back (`clwb`) the cache line to move the data into the ADR and eventually the Optane DIMM. Alternatively, the `ntstore` instruction writes directly to persistent memory, bypassing the cache hierarchy. For all these instruc-
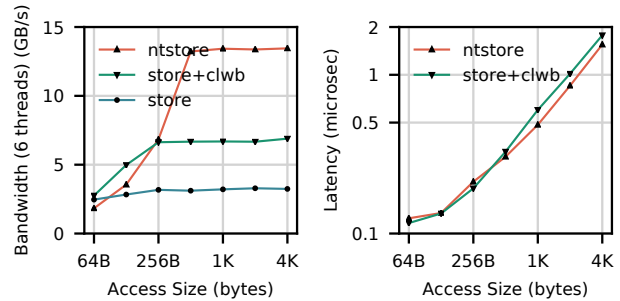

Figure 12: **Performance achievable with persistence instructions** Flush instructions have lower latency for small accesses, but `ntstore` has better latency for larger accesses. Using `ntstore` also avoids an additional read of the cache line from Optane memory, resulting in higher bandwidth.

tions, a subsequent `sfence` ensures that the effects of prior evictions, write backs, and non-temporal stores are persistent.

In Figure 12, we compared achieved bandwidth (left) and latency (right) for sequential accesses using AVX-512 stores with three different instruction sequences: `ntstore`, `store + clwb`, and `store`, followed by a `sfence`. Our bandwidth test used six threads as it gives good results for all instructions. The data show that flushing after each 64 B store improves the bandwidth for accesses larger than 64 B. We believe this occurs because letting the cache naturally evict cache lines adds nondeterminism to the access stream that reaches the Optane DIMM. Proactively cleaning the cache ensures that accesses remain sequential. The EWR correlates this hypothesis: Adding flushes increases EWR from 0.26 to 0.98.

The data also shows that non-temporal stores have lower latency than `store + clwb` for accesses over 512 B. Non-temporal stores also have highest bandwidth for accesses over 256 B. Here, the performance boost is due to the fact that a `store + clwb` must load the cache line into the CPU's local cache before executing store, thereby using up some of the Optane DIMMs bandwidth. As `ntstores` bypass the cache, they will avoid this extraneous read and can achieve higher bandwidth.

In Figure 13, we show how `sfences` affect performance. We used a single thread to issue sequential writes of different sizes on Optane-NI. We issued `clwb` during the write of each cache line (every 64B), or after the entire write (write size). At the end of the write we issued a single `sfence` to ensure the entire write is persistent (we call this entire operaton an "`sfence` interval"). The result shows the bandwidth peaks when the write size is 256 B. This peak is a consequence of the semantics of `clflushopt` which is tuned for moderately sized writes [1]. Flushing during or after a medium sized write (beyond 1 kB) does not affect the bandwidth, but when the write size is over 8 MB, flushing after the write causes performance degradation as we incurred cache capacity invalidations and a higher EWR.
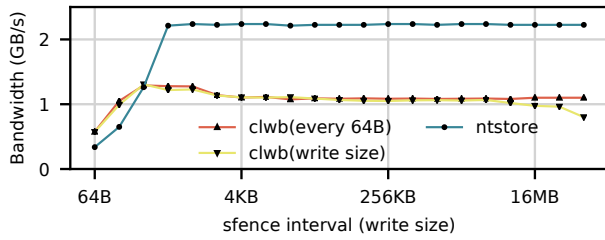
Figure 13: **Bandwidth over `sfence` intervals.** The bandwidth of Optane memory decreases when `sfence` interval increases, causing implicit cache evictions.
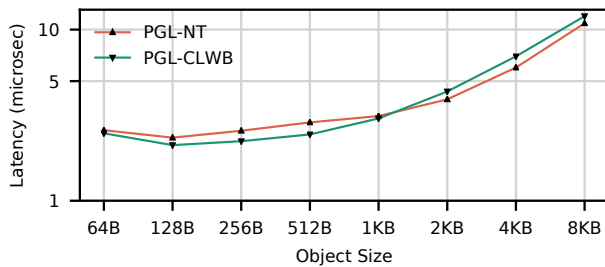

Figure 14: **Persistence instructions for micro-buffering** Using `ntstores` for large writes, and using `clwb` for small ones can improve performance even at macro-level. (Y-axis is in log scale)

5.2.1 Case Study: Micro-buffering for PMDK

Our analysis of the relative merits of non-temporal versus normal stores provides an opportunity to optimize existing work. For example, recent work proposed the "micro-buffering" [58] technique for transactionally updating persistent objects. That work modified the Intel's PMDK [14] transactional persistent object library to copy objects from Optane to DRAM at the start of a transaction rather than issuing loads directly to Optane. On transaction commit, it writes back the entire object at once using non-temporal stores.

The original paper only used non-temporal stores, but our analysis suggests micro-buffering would perform better if it used normal stores for small objects as long as it flushed the affected cache lines immediately after updating them. Figure 14 compares the latency of a no-op transaction for objects of various sizes for unmodified PMDK and micro-buffering with non-temporal and normal store-based write back. The crossover between normal stores and non-temporal stores for micro-buffering occurs at 1 kB.

5.3 Limit the number of concurrent threads accessing a Optane DIMM

Systems should minimize the number of concurrent threads targeting a single DIMM simultaneously. An Optane DIMM's limited store performance and limited buffering at the iMC
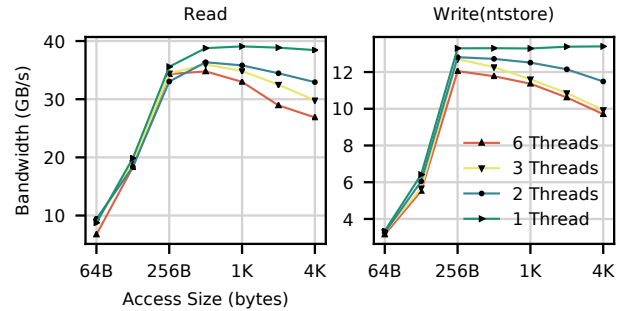

Figure 15: **Plotting iMC contention.** With a fixed number of 6 threads, as the number of DIMMs accessed by each thread grows, the bandwidth drops. For maximal bandwidth, threads should be pinned to DIMMs.

and on the DIMMs combine to limit its ability to handle accesses from multiple threads simultaneously. We have identified two distinct mechanisms that contribute to this effect.

Contention in the XPBuffer Contention for space in the XPBuffer will lead to increased evictions and write backs to 3D-XPoint media, which will drive down EWR. Figure 4 (center) shows this effect in action: the performance does not scale with higher thread counts. For example, having 8 threads issuing sequential non-temporal stores achieves an EWR of 0.62 and 69% bandwidth compared to a single thread, which has an EWR of 0.98.

Contention in the iMC Figure 15 illustrates how limited queue capacity in the iMC also hurts performance when multiple cores target a single DIMM. The figure shows an experiment that uses a fixed number of threads (24 for read and 6 for ntstore) to read/write data to 6 interleaved Optane DIMMs. We let each thread access *N* DIMMs (with even distribution across threads) randomly. As *N* rises, the number of writers targeting each DIMM grows, but the per-DIMM bandwidth drops. A possible culprit is the limited capacity of the XP-Buffer, but EWR remains very close to 1, so the performance problem must be in the iMC.

On our platform, the WPQ buffer queues up to 256 B data issued from a single thread. Our hypothesis is that, since Optane DIMMs are slow, they drain the WPQ slowly, which leads to head-of-line blocking effects. Increasing *N* increases contention for the DIMMs and the likelihood that any given processor will block waiting for stores ahead of it to complete.

Figure 5 (right) shows another example of this phenomenon — Optane bandwidth falls drastically when doing random 4 kB accesses across interleaved Optane DIMMs. Optane memory interleaving is similar to RAID-0 in disk arrays: The chunk size is 4 kB and the stripe size is 24 kB (Across the 6 DIMMs on the socket, each gets a 4 kB contiguous block). The workload in Figure 5 (right) spreads accesses across these interleaved DIMMs, and will lead to spikes in contention for particular DIMMs.

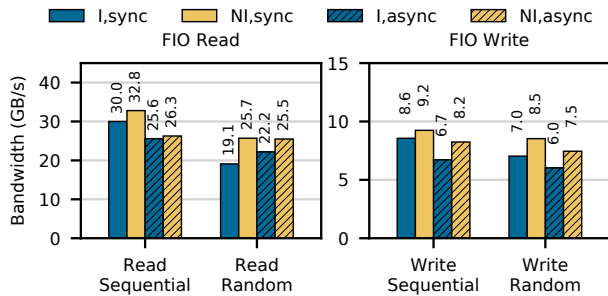Thread starvation occurs more often as the access size

Figure 16: **Multi-DIMM NOVA** We make NOVA multi-DIMM aware by evenly loading the NVDIMMs, and get improve performance by an average of 17% on the FIO benchmark.
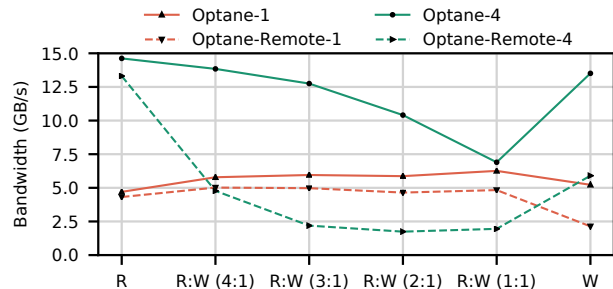

Figure 17: **Memory bandwidth on Optane and Optane-Remote** This chart shows bandwidth as we varied the mix of accesses for one and four threads. Pure reads or pure writes perform better on NUMA than mixed workloads, and increased thread count generally hurts NUMA performance.

grows, reaching maximum degradation at the interleaving size (4 kB). For accesses larger than the interleaving size, each core starts spreading their accesses across multiple DIMMs, evening out the load. The write data also show small peaks at 24 kB and 48 kB where accesses are perfectly distributed across the six DIMMs.

This degradation effect will occur whenever 4 kB accesses are distributed non-uniformly across the DIMMs. Unfortunately, this is probably a common case in practice. For instance, a page buffer with 4 kB pages would probably perform poorly in this configuration.

5.3.1 Case Study: Multi-NVDIMM NOVA

The original NOVA design did not attempt to limit the number of writers per DIMM. In fact, it tends to allocate pages for a file from contiguous regions which, via interleaving, tends to spread those pages across the DIMMs. To fix this issue, we configured our machine to pin writer threads to non-interleaved Optane DIMMs. This configuration ensures an even matching between threads and NVDIMMs, thereby leveling the load and maximizing bandwidth at each NVDIMM.

Figure 16 shows the result. Our experiment uses the FIO benchmark [6] to test the optimization and uses 24 threads. We plot the bandwidth of each file operation with two different IO engines: `sync` and `libaio` (async). By being Multi-NVDIMM aware, our optimization improves NOVA's bandwidth by between 3% and 34%.

5.4 Avoid mixed or multi-threaded accesses to remote NUMA nodes

NUMA effects for Optane are much larger than they are for DRAM, so designers should work even harder to avoid cross-socket memory traffic. The cost is especially steep for accesses that mix load and stores and include multiple threads. Between local and remote Optane memory, the typical read latency difference is 1.79× (sequential) and 1.20× (random), respectively. For writes, remote Optane's latency is 2.53×

(ntstore) and 1.68× higher compared to local. For bandwidth, remote Optane can achieve 59.2% and 61.7% of local read and write bandwidth at optimal thread count (16 for local read, 10 for remote read, and 4 for local and remote write).

The performance degradation ratio above is similar to remote DRAM to local DRAM. However, the bandwidth of Optane memory is drastically degraded when either the thread count increases or the workload is read/write mixed. Based on the results from our systematic sweep, the bandwidth gap between local and remote Optane memory for the same workload can be over 30×, while the gap between local and remote DRAM is, at max, only 3.3×.

In Figure 17, we show how the bandwidth changes for Optane on both local and remote CPUs by adjusting the read and write ratio. We show the performance of a single thread and four threads, as local Optane memory performance increases with thread count up to four threads for all the access patterns tested. Single-threaded bandwidth is similar for local and remote accesses. For multi-threaded accesses, remote performance drops off more quickly as store intensity rises, leading to lower performance relative to the local case.

5.4.1 Case Study: PMemKV

Intel's Persistent Memory Key-Value Store (PMemKV [15]) is an NVMM-optimized key-value data-store. It implements various index data structures and uses PMDK [14] to manage its persistent data. We used the concurrent hash map (`cmap`) in our tests as it is the only structure that supports concurrency.

To test the effect of Optane's NUMA imbalance on PMemKV, we varied the location of the server relative to the `pmem` pool; Figure 18 shows the result on an included benchmark with mixed workload (`overwrite`) that repeatedly performs read-modify-write operations. In this test, using a remote Optane DIMM drops the application's performance beyond two threads. Optane performance is far more impacted by the migration (loss of 75%) than DRAM (loss of 8%).
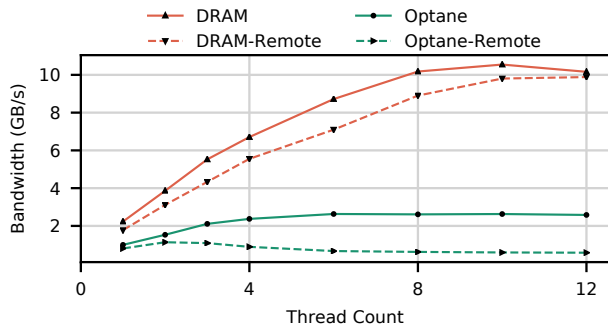
Figure 18: **NUMA degradation for PmemKV** Optane memory experiences greater NUMA-based degradation than DRAM. Migrating to a remote Optane node reduces PmemKV performance by up to 4.5× (18× vs. DRAM).

6 Discussion

The guidelines in Section 5 provide a starting point for building and tuning Optane-based systems. By necessity, they reflect the idiosyncrasies of a particular implementation of a particular persistent memory technology, and it is natural to question how applicable the guidelines will be to both other memory technologies and future versions of Intel's Optane memory. It is also important to note that we have only studied the guidelines in the context of App Direct mode, since the large DRAM cache that Memory Mode provides mitigates most or all of the effects they account for. We believe that our guidelines will remain valuable both as Optane evolves and as other persistent memories come to market.

The broadest contribution of our analysis and the resulting guidelines is that they provide a road map to potential performance problems that might arise in future persistent memories and the systems that use them. Our analysis shows how and why issues like interleaving, buffering on and off the DIMM, instruction choice, concurrency, and cross-core interference can affect performance. If future technologies are not subject the precisely the same performance pathologies as Optane, they may be subject to similar ones.

Ultimately it is unclear how scalable persistent memories will evolve. Several of our guidelines are the direct product of (micro)architectural characteristics of the current Optane incarnation. The size of the XPBuffer and iMC's WPQ might change in future implementations which would limit the importance of minimizing concurrent threads and reduce the importance of the 256 B write granularity. However, expanding these structures would increase the energy reserves required to drain the ADR during a power failure. Despite this, there are proposals to extend the ADR down to the last-level cache [37, 61] which would eliminate the problem. An even more energy-intensive change would be to make the DRAM cache that Optane uses in Memory mode persistent.

Increasing or decreasing the 256 B internal write size is likely to be expensive. It is widely believed that Optane is

phase-change memory and the small internal page size has long been a hallmark of the phase change memory [2] due to power limitations. Smaller internal page sizes are unlikely because the resulting memories are less dense.

A different underlying memory cell technology (e.g., spin-torque MRAM) would change things more drastically. Indeed, battery-backed DRAM is a well-known and widely deployed (although not very scalable or cost-effective) persistent memory technology. For it, most of our guidelines are unnecessary, though non-temporal stores are still more efficient for large transfers due to restrictions in the cache coherency model.

7 Related Work

With the release of the Optane DIMM in April 2019, early results on the devices have begun to be published. For instance, Van Renan et al. [44] have explored logging mechanisms for the devices. We expect additional results to be published in the near future as the devices become more widely available.

Prior art in persistent memory programming has spanned the system stack, though until very recently these results were untested on real Optane memory. A large body of work has explored transactional memory-type abstractions for enforcing a consistent persistent state [5, 9, 12, 25, 26, 28, 33, 49]. Various authors have built intricate NVM data structures for logging, data storage, and transaction processing [3, 4, 10, 11, 17, 23, 24, 27, 30, 35, 38, 39, 41, 45, 46, 50, 57]. Custom NVM file systems have also been explored [13, 21, 34, 48, 52, 54–56, 62].

8 Conclusion

This paper has described the performance of Intel's new Optane DIMMs across micro- and macro-level benchmarks. In doing so, we have extracted actionable guidelines for programmers to fully utilize these devices' strengths. The devices have performance characteristics that lie in-between traditional storage and memory devices, yet they also present interesting performance pathologies. We believe that the devices will be useful in extending the quantity of memory available and in providing low-latency storage.

Acknowledgments

References

[1] Intel® 64 and IA-32 Architectures Optimization Reference Manual. 2019.

[2] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Onyx: A protoype phase change memory storage array. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, Berkeley, CA, USA, 2011.

[3] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. BzTree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.*, 11(5):553–565, January 2018.

[4] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage: Recovery methods for non-volatile memory database systems. In *SIGMOD*, Melbourne, Australia, 2015.

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

[6] Jens Axboe. Flexible I/O Tester, 2017. https://github.com/axboe/fio.

[7] Brian Beeler. Intel Optane DC persistent memory module (PMM). https://www.storagereview.com/intel_optane_dc_persistent_memory_module_pmm. Accessed 1/1/2020.

[8] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 677–694, 2016.

[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 & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[10] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. VLDB Endow.*, 8(5):497–508, January 2015.

[11] Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[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 '11, pages 105–118, New York, NY, USA, 2011. ACM.

[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*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[14] Intel Corporation. Persistent Memory Development Kit. http://pmem.io/pmdk/.

[15] Intel Corporation. pmemkv: key/value datastore for persistent memory. https://github.com/pmem/pmemkv.

[16] Jeffrey Dean and Sanjay Ghemawat. LevelDB. https://github.com/google/leveldb.

[17] Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stan Zdonik, and Subramanya R. Dulloor. A prolegomenon on OLTP database systems for non-volatile memory. *Proc. VLDB Endow.*, 7(14), 2014.

[18] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, 2017. USENIX Association.

[19] Mingkai Dong, Qianqian Yu, Xiaozhou Zhou, Yang Hong, Haibo Chen, and Binyu Zang. Rethinking benchmarking for NVM-based file systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 20:1–20:7, New York, NY, USA, 2016. ACM.

[20] Z. Duan, H. Liu, X. Liao, and H. Jin. HME: A lightweight emulator for hybrid memory. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1375–1380, March 2018.

[21] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[22] Facebook. RocksDB, 2017. http://rocksdb.org.

[23] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. High performance database logging using storage class memory. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1221–1231, April 2011.

[24] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 28–40, New York, NY, USA, 2018. ACM.

[25] Eric 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, May 2015.

[26] Terry Ching-Hsiang Hsu, Helge Bruegner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Prac-

tical persistence for multi-threaded applications. In *Proceedings of the 12th ACM European Systems Conference*, EuroSys 2017, Belgrade, Republic of Serbia, 2017.

[27] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware logging in transaction systems. In *Proceedings of the VLDB Endowment*, 2014.

[28] 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. ACM.

[29] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. pVM: Persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 13:1–13:16, New York, NY, USA, 2016. ACM.

[30] Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 691–706, New York, NY, USA, 2015. ACM.

[31] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated persist ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[32] 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, pages 460–477, New York, NY, USA, 2017. ACM.

[33] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *51st IEEE/ACM International Symposium on Microarchitecture*, MICRO '18, October 2018.

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

[35] 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)*, Santa Clara, CA, 2017. USENIX Association.

[36] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, pages 1:1–1:17, New York, NY, USA, 2013. ACM.

[37] Dushyanth Narayanan and Orion Hodson. Whole-system persistence with non-volatile memories. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. ACM, March 2012.

[38] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey, Dhruva Chakrabarti, and Michael L. Scott. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing*, DISC '17, October 2017.

[39] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory.

[40] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[41] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage management in the NVRAM era. In *Proc. VLDB Endow.*, October 2014.

[42] Andy Rudoff. Deprecating the PCOMMIT instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, 2016. Accessed 1/1/2020.

[43] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. Failure-atomic slotted paging for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 91–104, New York, NY, USA, 2017. ACM.

[44] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory I/O primitives. *arXiv:1904.01614*, 2019.

[45] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST '11, San Jose, CA, USA, February 2011. USENIX Association.

[46] Stratis D Viglas. Write-limited sorts and joins for persistent memory. *Proc. VLDB Endow.*, 7(5):413–424, 2014.

[47] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 37–49, New York, NY, USA, 2015. ACM.

[48] 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.

[49] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2011. ACM.

[50] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[51] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 70–83, New York, NY, USA, 2018. ACM.

[52] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[53] 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.

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

[55] 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.

[56] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, 2019.

[57] 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 *13th USENIX Conference on File and Storage Technologies*, FAST '15, pages 167–181, Santa Clara, CA, February 2015. USENIX Association.

[58] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 897–912, Renton, WA, July 2019.

[59] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *Proceedings of the 2015 IEEE Symposium on Mass Storage Systems and Technologies (MSST'15)*, 2015.

[60] 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, pages 3–18, New York, NY, USA, 2015. ACM.

[61] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.

[62] 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, Boston, MA, 2019. USENIX Association.

DC-Store: Eliminating Noisy Neighbor Containers using Deterministic I/O Performance and Resource Isolation

Miryeong Kwon[1], *Donghyun Gouk*[1], *Changrim Lee*[1], *Byounggeun Kim*[2],
Jooyoung Hwang[2], *Myoungsoo Jung*[1]
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)[1], *Samsung*[2]
http://camelab.org

Abstract

We propose DC-store, a storage framework that offers deterministic I/O performance for a multi-container execution environment. DC-store's hardware-level design implements multiple NVM sets on a shared storage pool, each providing a deterministic SSD access time by removing internal resource conflicts. In parallel, software support of DC-Store is aware of the NVM sets and enlightens Linux kernel to isolate noisy neighbor containers, performing page frame reclaiming, from peers. We prototype both hardware and software counterparts of DC-Store and evaluate them in a real system. The evaluation results demonstrate that containerized data-intensive applications on DC-Store exhibit 31% shorter average execution time, on average, compared to those on a baseline system.

1 Introduction

Docker container technology is applied to diverse computing domains such as datacenter, serverless, cloud, and high-performance computing thanks to its predictable development and efficient resource isolation [1–7]. Docker can isolate the execution of specific applications from running other applications as Sandbox [8]. Specifically, users can explicitly set a resource limit for each service by using *control groups* (cgroups [9]) and namespace [10]. Therefore, different services can be free from conflicting dependencies and resource contention. Since this consistent environment supports completely isolated tenants, it is used to improve computing utilization and portability [11, 12]. For example, LEMP (Linux, Nginx, MySQL, & PHP) in the Google cloud platform can create multiple tenants per node, each running a separate web server (Nginx), fast CGI (PHP-FPM), and database (MySQL) [13].

While Docker is one of the best options to increase the computing utilization by sharing and/or isolating its resources, it does not yet well manage underlying storage [14–16]. Docker's virtualized environment enables multiple container executions atop the host-side kernel directly, which is different from hardware stack virtualization of the conventional virtual machines (VMs) [17]. Because of this OS-level virtualization, launching a container is much faster and lighter than executing a VM [14, 18, 19]. However, the multiple containers and host kernel often share the persistent states on a solid-state drive (SSD), which can unfortunately interfere with their executions and I/O services of any peers at a device-level. As modern SSDs in practice exploit internal parallelism to enhance performance [20–26], such interference introduces many storage resource conflicts, thereby degrading the performance. To manage the storage resource, ones may utilize proportional bandwidth of the blkio cgroup adopted by Docker [27] and assign different proportions to each of multiple containers, individually. However, this approach cannot address the interference issue for multi-container execution due to two root causes. First, even though the blkio cgroup throttles the target I/O queue with different proportions [28], the flash-level resource conflicts cannot be resolved as the host is unaware of the physical layout of the underlying SSD [29, 30]. Second, metadata I/O services, including page frame reclaiming, are not taken by I/O throttling of the blkio cgroup.

In this paper, we introduce *DC-Store*, a storage framework that supports deterministic I/O performance and resource isolation for the multi-container execution environment. DC-Store consists of two major components, called *Divided SSD* and *I/O Tacker*. Generally speaking, Divided SSD addresses I/O interference at the hardware-level, thereby offering deterministic performance per volume. At the same time, I/O Tacker isolates I/O requests of noisy neighbors from the execution of other peers by considering each container's volume ownership at the software-level. We design and implement Divided SSD, which separates physical flash channels and internal resources to support multiple NVM sets, which were very recently included by NVMe workgroup (NVMe 1.4 [31]). In specific, we statically partition the computing and hardware resources per NVM set such that different I/O requests

heading across different NVM sets can have no or minor flash-level resource conflicts. The proposed Divided SSD can provide an independent environment, which allows multiple containers to operate without the device-level I/O interference. It also enables us to use a large size of shared storage with better utilization per node, which is in typical required by modern data centers [32, 33].

Nevertheless, when multiple containers mount each of different NVM sets (provided by our Divided SSD), we still observe that the user experience of some containerized applications degrades from an execution time angle. Even though Linux cgroups and namespaces can limit/split the CPU, memory, and blkio subsystems for different containers, if there is a noisy container that unintentionally issues page-out I/O requests imposed by a low on virtual memory, the performance of other peers is severely interfered and degraded. Since Docker is not involved in metadata I/O management (and Linux treats containers as just like other processes), it cannot fully support the consistent environment for completely isolated tenants.

To address this shortcoming, the proposed I/O Tacker modifies Linux kernel in order to enable per-container page frame reclaiming by considering the container ownership and exposed NVM set organization. The current version of Linux applies the same page-in/out procedures to all processes and containers, which can introduce container-level performance interference. In contrast, our I/O Tacker assigns a per-container swap area and informs its the swap location to the kernel by modifying the kernel memory controller. Therefore, the noisy neighbor performs page in/out only from/to its own NVM sets. To this end, we also revise the Docker stack for users to pass through the swap-pinning information from the Docker client to the container engine. The I/O Tacker's swap pinning mechanism being aware of the container ownership can isolate noisy neighbors from other containers, thereby addressing the performance degradation of all containers running on the shared storage.

To the best of our knowledge, this paper is the first work that implements multiple NVM sets in real hardware as well as isolates the low on virtual memory impact from the execution environment of containers. Our results show that, even though the average latency of our Divided SSD prototype is worse than a baseline NVMe SSD for a single volume accesses (with a specific pattern), it provides better and deterministic I/O performance under concurrent storage accesses. When we co-run containerized data-intensive applications with memory-hungry containers (e.g., LEMP), the proposed DC-Store improves user-level experiences of the applications by 31%, on average, without degrading the performance of such memory hungry noisy neighbors.
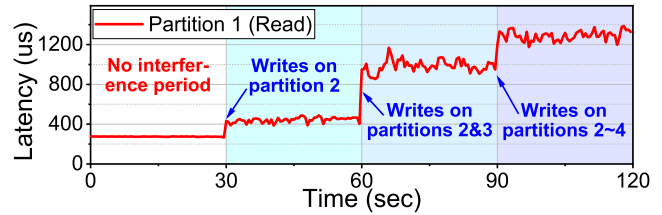

Figure 1: Interference in storage sharing.

2 Divided SSD: Hardware Level Design

Challenges in storage sharing. For better storage utilization and parallel accesses, a simple solution to allocate multiple containers to a shared SSD is creating multiple disk partitions on the storage; each partition can be mounted to a specific container for its I/O services. However, concurrent SSD accesses for the logically partitioned storage volumes can introduce many flash-level resource conflicts, thereby degrading performance per partition. To be precise, we perform I/O interference tests on a high-performance NVMe SSD, and the results are shown in Figure 1. In this evaluation, we split a 2TB SSD (whose device configuration is explained in Table 1) into four disk partitions. We execute a Facebook-like workload (interspersing bursty writes into sustained reads [33]) as a representative I/O access pattern of cloud environment by executing four containerized FIO applications [34] on the different disk partitions. We measure the read latency on a partition while all peer partitions are in serving bursty writes; the writes on each peer partition are started with every 30 secs (to understand the I/O interference impact). One can observe from the figure is that the random read latency is around 280 us and very sustainable at the first 30 secs period. However, once the bursty writes on another partition begin, the latency increases as high as 400 us (1.42 times longer). When all other partitions are dedicated to handling the bursty writes, the latency on the read service is 4.5 times higher than the normal operations.

Overview. To address the performance degradation in storage sharing, our Divided SSD physically partitions all SSD resources in the datapath. Figure 2a illustrates a high-level organization of the proposed Divided-SSD. Even though there exists a single PCIe SSD device connected to the host, our Divided SSD exposes multiple storage volumes; each has its own flash firmware such as *flash translation layer* (FTL) and block address space. These multiple volumes (of our Divided SSD) are visible from the host as "physically" separated NVM sets. Thus, the host need to employ completely different storage stack instances across multiple NVM sets, which can be mounted to each of the different containers.

Hardware design. Figure 2b shows the internal design of an ideal Divided SSD. Specifically, the proposed Divided SSD employs multiple low-power cores, each having instruction/data memory (I-cache/D-cache). These cores have their
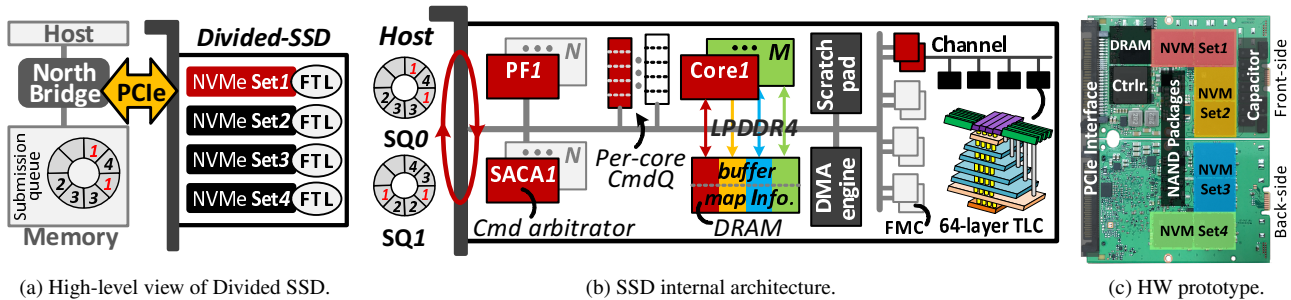
(a) High-level view of Divided SSD.   (b) SSD internal architecture.   (c) HW prototype.

Figure 2: NVM set design and implementation (Divided SSD).

own scratchpad memory for latency-sensitive and/or shared data accesses. The cores are all connected to a global DRAM through LPDDR4, which are also accessed by a direct memory access (DMA). The DRAM can buffer the host requests and manage the mapping information of FTLs. Each core can have one or more flash memory controllers (FMCs), each being also connected to flash buses, called *channels*. Each channel employs multiple flash dies, which operate in parallel as well, called *ways* [35].

To offer multiple and physically separated storage volumes (which can address the resource conflicts), we statically split the all SSD internal resources in the datapath per NVM set. Specifically, as shown in Figure 2b, the global DRAM's memory address space is partitioned and independently allocated to each core at design time. As we disable the shared last-level cache, flash firmware instances, operating across different cores do not interfere with each other. In addition, each core exposes the independent flash address space of its own underlying channel(s) to the corresponding NVMe controller. The design of our static partitions makes multi-core management easier and enables flash firmware to operate on different DRAM and storage regions in a separate way as well as work entirely in parallel. Thus, in our design, each FTL (assigned to each of statically partitioned DRAMs and cores) independently performs the address translation between logical to physical flash pages per NVM set.

NVM set control. To connect the underlying cores and firmware to the host processor(s), the device requires employing multiple PCIe *physical functions* (PFs). Note that, while the ideal design of Divided SSD can employ as many NVMe PFs as the number of NVM sets, the number of PFs that an SSD exploits in a real-world scenario would be limited; it can even be smaller than the number of cores. To address this practical challenge, we introduce an *NVM set-aware command arbitrator* (SACA) for each PF. SACA employs an NVMe command queue per NVM set (and core). SACA distributes all the incoming NVMe commands, associated with different NVM sets, to the appropriate NVMe command queue by fetching the commands from a specific NVMe *submission queue* (SQ) in a round-robin fashion. This design can make

the firmware on each core (statically assigned to different NVM sets) separately perform I/O processing on the designated NVM set (even with a limited number of PFs). Even though different host CPUs/cores of the host can in parallel issue the NVMe commands for the different NVM sets per SQ, SACA can appropriately serve them atop physically partitioned different flash channels (per NVM set). To manage the I/O completion, SACA collects the I/O completion messages (coming from different cores) and puts them into the target NVMe *completion queue* (CQ), which is associated with the submission queue. Thus, SACA can handle I/O requests, targeting to different NVM sets but being enqueued in the same SQ/CQ, with a design of limited PFs.

Prototype. Figure 2c shows the current Divided SSD that we prototyped. The prototype hardware has a front-side and back-end PCBs, each containing four flash channels and eight ways. In total, our prototype hardware employs 64 flash dies, each using 64 layered triple-level cell (TLC) flash technology. We connect them to four cores that run four different flash firmware instances. All these set-related cores are interconnected to 3GB LPDDR global memory (physically partitioned to each core). Thus, the prototype exposes four independent NVM sets (2 channels per set) over one PF, which is capable of storing 2TB data into the single storage pool of the device.

3   I/O Tacker: System Level Support

Page-frame reclaiming interference. While our Divided SSD physically separates its shared storage space into multiple NVM sets, each being mounted to different containers, the I/O interference, caused by noisy neighbors, can degrade the performance of peer containers. To be precise, we issue random reads to the first NVM set using FIO configured with the Facebook-like workload used in Section 2, while executing a memory hungry application on the other NVM sets. The configuration of such application, LEMP, is described in Section 4. Figures 3a and 3b respectively show the amount of storage accesses (footprint) and latency of the first NVM set when we increase the number of LEMP instances, each mounting different NVM sets. Even though there is no instance working on the first NVM set except for FIO, the storage footpoint

increases as the number of LEMP increases by 15.8% at most (Figure 3a). As shown in Figure 3b, this unfortunately makes the average and worst-case latency of random reads as high as 22% and 931%, respectively. The reason why we can observe the performance degradation on Divided SSD is that the memory subsystem of LEMP containers introduces extra I/O requests, targeting to the storage area, mounted to the first NVM set. In Linux, containers are treated just as processes, the I/O requests, caused by demand paging, are all together heading to the swap disk or file that the host OS designates, which cannot be solely addressed by device-level performance isolation.

Overview of ownership-based I/O isolation. To remove the impact of the memory-hungry container(s) from other peers, our I/O Tacker pins the swap area to the target NVM set based on container ownership. Note that, while cgroups' memory subsystem is configured by users (when the target container is created), page frame reclaiming is performed by the host kernel. Thus, we modify several data structures of kernel and cgroup to link the kernel and cgroup by being aware of the container ownership. In addition, we modify the Docker stack to inform our per-container swap area information to the Docker container engine. Putting it all together, noisy neighbors running page-out and page-in operations are dedicated their own NVM sets, which do not interfere with other peer containers.

Cgroup and kernel modification. To redesign the swap operations (related demand paging), we modify the kernel memory controller (`linux/mm/memcontrol.c`) and define the read and write actions of memory subsystem's cgroup. The actions are declared by our new function that modifies the memory cgroup structure (`mem_cgroup`). As shown in Figure 3c, we add two swap information to the `mem_cgroup`, one for swap type (`swap_type`) and another for swap destination (`swap_dst`). While the swap type explains whether the current cgroup uses a default page frame reclaiming method or our per-container demand paging technique, the swap destination includes the index of swap information list (`swap_info`). An entry of the swap information list (`swap_info_struct`) contains the actual swap disk or swap file information. Thus, when the host kernel evicts a page frame, it will only page in/out from the designated location, if the swap type indicates the per-container demand paging. We also modify the eviction-related kernel function, `get_swap_pages()`, to appropriately refer the target index of swap information list by referring our modified `mem_cgroup`.

Note that all page frames (`page_struct`) contain a pointer indicating `mem_cgroup`, associated with the owner processes (container as well). Thus, when the host kernel selects a victim page frame by looking up the page table at `get_swap_pages()`, the function can retrieve the index of swap information of `mem_cgroup`. In `get_swap_pages()`, we forward the retrieved index to `scan_swap_map()` such that it will update the corresponding page table entry (PTE)

SSD	Capacity	2TB	DRAM	LPDDR4
	# channel	8	# die	64
	# core for sets	4	NAND	64 Layered TLC
Testbed	CPU	Xeon E5-2690 v3 (48 cores)		
	Nginx	1.17.1	DRAM	DDR4 256GB
	MySQL	8.0.16	Linux	5.0.7+
	PHP	7.2	Docker	18.09.7+

Table 1: Important parameters of our device and system.

with `swp_entry_t` that consists of the retrieved index and the specific offset within the swap area target. Once the kernel puts the victim page to its swap cache, which will page out to the backend storage, the kernel can check a specific swap destination location from `swap_info_struct` using the list index and perform per-container demand paging.

Docker stack modification. In Docker stack, we modify the container engine (`containerd`) to write our pinned swap information parameter (indicated by `mem.pinned`) to `mem_cgroup`. Specifically, the modified `containerd` will update the swap type and destination fields of `mem_cgroup` by collaborating with our kernel memory controller. As shown in Figure 3d, when users create a container, `containerd` generates the file of `mem.pinned` under the corresponding container's memory subsystem directory (`sys/fs/cgroup/memory/docker/<container-ID>/mem.pinned`). In the file creation process, it will jump to our function and update `mem_cgroup` (associated with the host kernel) in indicating a specific NVM set, explained by `swap_info_struct`. Once this cgroup update is completed, `containerd` changes the processor control block (`task_struct`) to link the `mem.pinned` with the structure of `cgroups`. Thus, the host kernel can be aware of the swap information that the users designate.

Since the proposed I/O Tacker takes `mem.pinned` information when creating a container, we also modify Docker stack to deliver such new parameter from Docker client (`docker-cli`), Docker daemon (`dockerd`), to `containerd`, as shown in Figure 3e. As Docker modules in the stack communicate with `docker-cli` through HTTP, the modified `docker-cli` delivers the `mem.pinned` information by packing it into `containerConfig` structure using a JSON format [36]. Docker engine (`dockerd`) atop `containerd` then parses such parameter information and forwards it to `containerd` using a Google remote procedure call (gRPC). Therefore, `containerd` can access the `mem.pinned` information and create the container that uses per-container page frame reclaiming.

4 Evaluation

We prepare two real NVMe SSD hardware devices, one for a high-performance NVMe SSD and another for our Divided SSD prototype; the hardware configuration of those two is
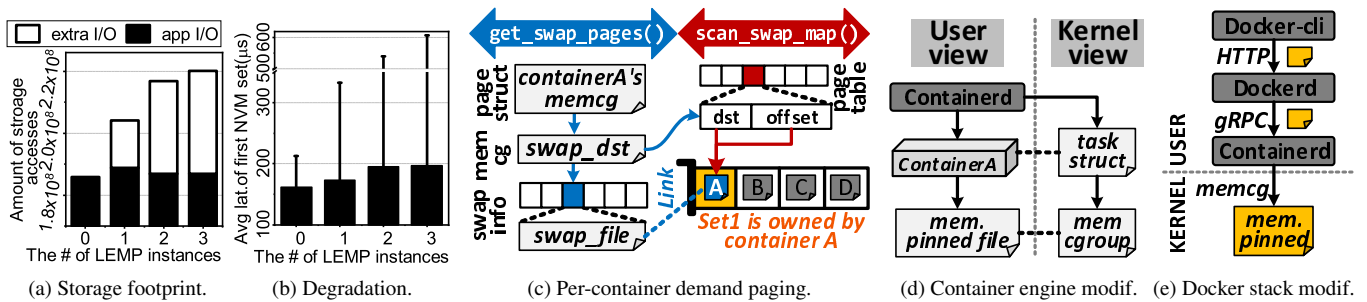
(a) Storage footprint.   (b) Degradation.   (c) Per-container demand paging.   (d) Container engine modif.   (e) Docker stack modif.

Figure 3: Overview of system-level support (I/O Tacker).

exactly same (e.g., the same number of cores, controllers, channels and flash devices). We connect each device into our testbeds; each of them employs two processors (total 48 cores), 256 GB DDR4 DRAM and uses PCIe 3.0 4 lanes for the SSD prototype connections. In the *Baseline* testbed, we connect the NVMe SSD and logically partition it into four (each offering 480 GB), while the *DC-Store* testbed uses four NVM sets of Divided SSD with I/O Tacker (modified by the same version of Linux kernel 5.0.7 that Baseline installs). All our testbeds use EXT4 as their file system. We use grep, wordcount (wc), minmax for containerized data-intensive applications, while sort is evaluated for a compute-intensive application. All the containerized applications generate their outputs by examining a set of randomly generated data vectors, stored to 630,000 files on EXT4. For noisy neighbors, we prepare LEMP, consisting of one Nginx, two PHP-FPM, and one MySQL per container. The important device information is summarized in Table 1.

Overall performance. Figure 4a shows the execution time of Baseline and DC-Store by co-running containerized applications with a LEMP stack. For the scenario of co-running data-intensive containers with LEMP, DC-Store finishes the applications in 174 secs, on average, which is 31% shorter than Baseline. Since LEMP performs demand paging, the corresponding I/O requests are sporadically issued to the NVMe volume that the data-intensive containers operate, which in turn degrades the performance of Baseline. In contrast, since Divided SSD completely splits the NVMe volumes through different channels, and I/O Tacker isolates the page frame reclaiming from the peer NVMe volumes in processing the data, the performance degradation is not observed with DC-Store. Specifically, grep, wc and minmax on DC-Store shorten the container-level execution times (compared to those of Baseline) by 29.2%, 29.2%, and 33.9%, on average, respectively. On the other hand, sort (compute-intensive) exhibits similar performance between Baseline and DC-Store (7.8% better than Baseline). Since most of the sort executions are involved in CPU processing (for data comparisons for sorting), the I/O requests from the containerized sort occasionally occur. We will dig deeper the performance of Divided SSD and I/O

Tacker in Sections 4.2 and 4.3, respectively.

Note that LEMP itself shows negligible performance difference (under 0.16%, on average) for all the co-running scenarios. Even though the degree of internal parallelism per NVM set would be lower than that of Baseline's NVMe SSD, LEMP's writes related to demand paging can be enqueued and buffered by the host-side memory. Thus, the LEMP execution can be tolerable at some extent. Specifically, we observe that, compared to Baseline, the target NVM set's I/O depth increases by 56%, on average. However, since the swap cache issues such I/O requests with long intervals (60 ms in average), all the requests are served within the long intervals thereby being invisible to the users.

Different levels of interference. We increase the number of LEMP containers from one to three to analyze the different levels of interference, imposed by noisy neighbors; each LEMP mounts different NVM sets, and all remaining sets are used for the containerized applications. Figure 4b shows the application execution time of DC-Store, normalized to that of Baseline. We observe that DC-Store exhibits all similar execution time behaviors, which have no impact with the varying numbers of noisy neighbors. However, as Baseline performance gets worse (except for grep) due to extra reads and writes brought by demand paging, the performance of DC-Store looks get better in this normalization comparison. For Baseline, we observe that grep is a bit more sensitive on the access pattern that other grep instances generate. As increasing the number of LEMP (and relatively decreasing the number of running grep instances), Baseline shortens grep's completion time by 20 secs. Since DC-Store provides deterministic performance, which does not have impact on other peer NVM sets, the benefits on grep are only around by 21% compare to Baseline, on average (the right most of Figure 4b).

Time series analysis. Figure 5 shows a time series analysis of grep and sort as the representative of data- and compute-intensive applications, respectively. One can observe from Figure 5a, that the latency of containerized grep fluctuates as much as LEMP generates page frame reclaiming related I/O services. In contrast, DC-Store shows highly predictable and sustainable latency, irrespective of LEMP's page frame re-
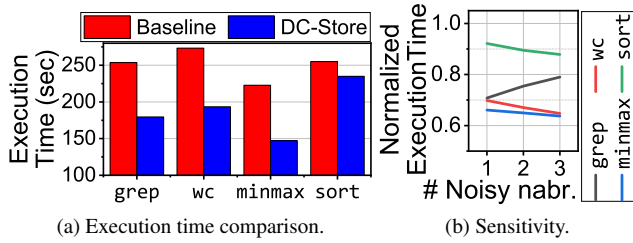
(a) Execution time comparison.   (b) Sensitivity.

Figure 4: DC-Store performance analysis.

(a) Data-intensive (grep).   (b) Compute-intensive (sort).

Figure 5: Timeline analysis.

claiming activities. This is because paging in/out of LEMP's memory subsystem occurs only to the NVM set that LEMP owns. In contrast, such I/O services of DC-Store (at the device-level) are completely isolated from other NVM sets. The storage latency of Baseline is shorter than DC-store when there is no interference from LEMP. This is because, unlike Divided SSD, Baseline NVMe SSD can utilize the internal DRAM optimized for sequential accesses. Note that the reason why there is no I/O from LEMP between 0 and 1 sec in Figure 5a is that all I/Os during this period were absorbed by the DRAM. We will further analyze this behavior in Section 4.2. For the containerized sort, it exhibits a different performance characteristic. Since the performance of sort is dominated by the host computing power, sort introduces I/O requests sparsely (Figure 5b). Nevertheless, its latency trend of Baseline still fluctuates, which is similar to that of LEMP's page frame reclaiming activities. This, in turn, makes the containers on Baseline in overall show 35% performance degradation, compared to the execution time with no I/O interference.

4.1 Eliminating Noisy Neighbors

4.2 Device-level Analysis

Figures 6a and 6b respectively show the level of I/O determinism for sequential and random access patterns in terms of latency. The evaluation is performed by the same scenario of flash workloads from Facebook [33] (read-intensive with bursty writes); the latency values are the read performance of the first NVM set while writing data into other NVM sets for every 30 secs. There are two insights observed in this evaluation. At the first test epoch (∼30 secs), Divided SSD is, on average, 5.8 times slower than Baseline NVMe for the sequential accesses (Figure 6a) while it exhibits better performance on random accesses (157% faster, on average). As Divided SSD physically partitions the underlying flash channels, it cannot take the full advantage of internal parallelism; Baseline's NVMe SSD reads data across all eight different channels in parallel, whereas Divided SSD only uses two channels. More importantly, this parallelism can increase if DRAM buffer is sufficiently large as the firmware can perform a read-ahead by enabling all the underlying channels. Divided SSD splits the internal DRAM region and assigns to each core, it exhibits
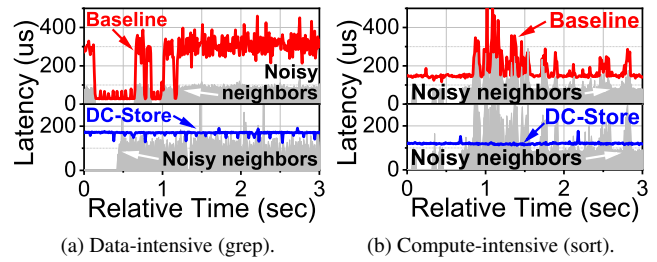
the performance degradation per NVM set on the sequential accesses. In contrast, the latency on random accesses is difficult to take a full benefit of the read-ahead and parallelism; for the random, the resource conflict can be higher, and there is a low spatial locality, which makes the latency of Divided SSD shorter than Baseline's NVMe SSD (Figure 6b).

Note that the key benefit of hardware-level NVM set implementation is to remove I/O interference and provide deterministic performance. As shown in the figure 6a, Baseline's NVMe SSD increases the latency for the remaining epochs and shows as high as 14.8 times longer latency, compared to the first epoch test. In contrast, Divided SSD exhibits 55%, 83%, and 86% shorter latency than Baseline's NVMe SSD for 2nd, 3rd and 4th epochs, on average, respectively. The main reason why Divided SSD offers deterministic I/O performance is to statically split all SSD resources per set such that the requests on other NVM sets cannot interfere the I/O services on the first NVM set.

4.3 I/O Tacker Performance Impact

We evaluate different methods of demand paging on multiple NMVe sets to see the performance benefits solely brought by I/O Tacker. *Default* uses the default demand paging configuration of kernel and *Round* pages in/out across different NVM sets in a round-robin fashion. In contrast, *Private* employs the per-container page frame reclaiming technique of I/O Tacker, which pins the swap area to a specific NVM set that LEMP owns. In this test, only a single containerized LEMP stack is executed, but it generates demand paging related I/O requests more than the test performed in Section 4.1 by double. The user-level execution time of all containers is shown in Figure 7a. Default requires 251, 260, 215 and 239 secs to execute grep, wc, minmax and sort (by co-running with LEMP), respectively. In contrast, Private requires 193, 197, 148 and 206 secs to complete their task, on average; Private for the data-intensive applications is faster than those on Default by 23% on average. Round also shows some benefits, compared to Default, but it is still slower than Private by 10%, on average.

Figure 7b shows the results of time series analysis for the containerized grep (as the representative of data-intensive applications). The latency of Default and Round fluctuates when the reads and writes related to demand paging occur
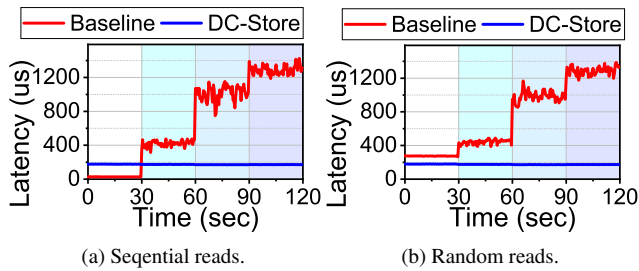
(a) Seqential reads. (b) Random reads.

Figure 6: Read-intensive with bursty writes.

(a) Execution time. (b) Time series analysis (grep).

Figure 7: I/O Tacker performance analysis.

even with Divided SSD. This is because even though Divided SSD implements hardware-level NVM sets with static resource partitioning, the I/O requests for demand paging are managed without the consideration of container ownership. In contrast, Private keeps the predictable and sustainable latency while serving such I/O requests by pinning the swap area to the NVM set that LEMP operates on. Importantly, we observe that the execution time of LEMP with Private is 60% and 22% shorter than those with Default and Round, respectively as well. This is because Private removes the serialized I/O requests on a specific NVM set for the page in/out of LEMP.

5 Related Work

There exist several studies that virtualize an SSD by isolating its channels or flash dies [37–39]. [37] proposed an over-provisioning space isolation to reduce garbage collection overheads, and similarly, [40] proposed an SSD-aware host-level I/O scheduler to improve proportionality on a shared storage system. [38] partitioned the flash resources of Open-Channel SSD (OCSSD) [41, 42] with a kernel-level modification to enhance the storage lifetime. In similar, [39] changed kernel driver(s) to make OCSSD support multi-tenants with artificial intermixed block-level I/O traces. While the prior work virtualizes the SSD with software support or performs simulation-based studies, we split all internal resources, including flash, buffer, and core, as well as memory subsystem (demand paging) in real hardware. Our prototype of DC-Store successfully achieves deterministic I/O performance and resource isolation.

On the other hand, some prior works [40, 43, 44] that tried to make the underlying SSD be aware of different degrees of QoS are unfortunately agnostic to container-level information and Docker storage stack. There are a few OS-level virtualization studies that attempt to satisfy different levels of QoS requirements of multiple applications in a shared storage system. Specifically, [28] and [45] modify a block layer and page cache in an attempt to utilize proportional bandwidth by being aware of per-container I/O weight. Unfortunately, these prior researches cannot address the storage-level interference issues for multi-container execution environment because the block layer and page cache that throttle the bandwidth (based
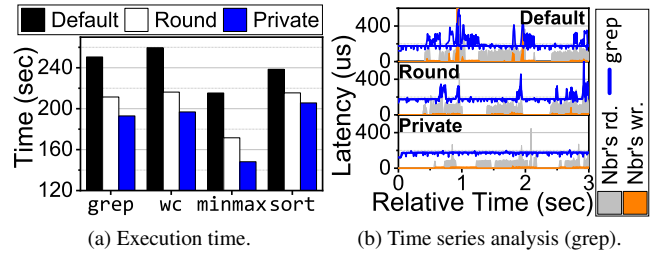
on per-container I/O weight) are not aware of the physical layout of SSDs and cannot split metadata I/O services from their I/O throttling.

6 Discussion

The number of NVMe set per device can be greater or smaller than the number of user applications running on Docker as the current prototype of our Divided SSD statically partitions all the internal resources at design time. While we consider designing different hardware platforms to dynamically allocate the different number of NVMe sets as future work, there are still several alternatives to map between different numbers of user applications and NVMe sets. In cases where users want to fully utilize all the bandwidth of Divided SSD with a small number of containers, it is possible to group multiple NVMe sets into a single shared storage partition using a conventional RAID scheme. In contrast, if there are containerized applications greater than the NVMe sets offered by Divided SSD, we can classify the applications based on the read and write ratios and then allocate them in homogenous groups to different NVMe sets. Even though this approach may not be optimal, it can provide a significant benefit by protecting read-intensive applications from the interference of heavy writes (requested by other containerized applications).

7 Conclusion

In this paper, we proposed DC-Store to offer deterministic I/O performance for a multi-container execution environment. We prototyped both hardware (Divided SSD) and software (I/O Tacker) modules of our DC-Store, and evaluate them in a real system, which shortens the data-intensive applications by 31%, on average, compared to those on a baseline system.

Acknowledgements

References

[1] A. Eivy, "Be wary of the economics of "serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.

[2] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[3] J. Higgins, V. Holmes, and C. Venters, "Orchestrating docker containers in the hpc environment," in *International Conference on High Performance Computing*, pp. 506–513, Springer, 2015.

[4] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," *Proceedings of the Cray User Group*, 2015.

[5] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, and N. Thoai, "Using docker in high performance computing applications," in *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pp. 52–57, IEEE, 2016.

[6] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[7] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[8] J. Shukla, "Application sandbox to detect, remove, and prevent malware," Jan. 17 2008. US Patent App. 11/769,297.

[9] "cgroup-v1." https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[10] "Linux namespace." http://man7.org/linux/man-pages/man7/namespaces.7.html.

[11] A. J. Younge, K. Pedretti, R. E. Grant, and R. Brightwell, "A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 74–81, IEEE, 2017.

[12] B. P. Rimal and M. Maier, "Workflow scheduling in multi-tenant cloud computing environments," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 1, pp. 290–304, 2016.

[13] S. Sakr, "Cloud-hosted databases: technologies, challenges and opportunities," *Cluster Computing*, vol. 17, no. 2, pp. 487–502, 2014.

[14] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th International Middleware Conference (Middleware)*, p. 1, ACM, 2016.

[15] S. McDaniel, S. Herbein, and M. Taufer, "A two-tiered approach to i/o quality of service in docker containers," in *2015 IEEE International Conference on Cluster Computing (Cluster)*, pp. 490–491, IEEE, 2015.

[16] J. Bhimani, Z. Yang, N. Mi, J. Yang, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Docker container scheduler for i/o intensive applications running on nvme ssds," *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)*, vol. 4, no. 3, pp. 313–326, 2018.

[17] "VirtualBox." https://www.virtualbox.org/.

[18] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *2015 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 386–393, IEEE, 2015.

[19] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *2014 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 610–614, IEEE, 2014.

[20] S. Park, E. Seo, J. Shin, S. Maeng, and J. Lee, "Exploiting internal parallelism of flash-based ssds," *IEEE Computer Architecture Letters (CAL)*, vol. 9, no. 1, pp. 9–12, 2010.

[21] E. H. Nam, B. S. J. Kim, H. Eom, and S. L. Min, "Ozone (o3): An out-of-order flash memory controller architecture," *IEEE Transactions on Computers (TC)*, vol. 60, no. 5, pp. 653–666, 2010.

[22] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren, "Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance," *IEEE Transactions on Computers (TC)*, vol. 62, no. 6, pp. 1141–1155, 2012.

[23] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the International Conference on Supercomputing (ICS)*, pp. 96–107, ACM, 2011.

[24] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 524–535, IEEE, 2014.

[25] M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. T. Kandemir, "Hios: A host interface i/o scheduler for solid state disks," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 289–300, 2014.

[26] M. Jung, E. H. Wilson III, and M. Kandemir, "Physically addressed queueing (paq) improving parallelism in solid state disks," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 404–415, 2012.

[27] "cgroup-v1 blkio." https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt.

[28] S. Ahn, K. La, and J. Kim, "Improving i/o resource sharing of linux cgroup for nvme ssds on multi-core systems," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[29] M. Jung, "Exploring parallel data access methods in emerging non-volatile memory systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 3, pp. 746–759, 2016.

[30] S. Koh, C. Lee, M. Kwon, and M. Jung, "Exploring system challenges of ultra-low latency solid state drives," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[31] "NVM Express Base Specification Revision 1.4." https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf.

[32] C. Petersen and A. Huffman, "Solving latency challenges with nvm express ssds at scale," *Flash Memory Summit*, 2017. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170809_SIT6_Petersen.pdf.

[33] C. Petersen, W. Zhang, and A. Naberezhnov, "Enabling nvme i/o determinism at scale," *Flash Memory Summit*, 2018. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180807_INVT-102A-1_Petersen.pdf.

[34] J. Axboe, "Flexible i/o tester." https://github.com/axboe/fio.

[35] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung, "Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 469–481, IEEE, 2018.

[36] "Docker engine." https://docs.docker.com/engine/api/v1.24/.

[37] J. Kim, D. Lee, and S. H. Noh, "Towards slo complying ssds through ops isolation," in *13th USENIX Conference on File and Storage Technologies (FAST)*, pp. 183–189, 2015.

[38] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds," in *15th USENIX Conference on File and Storage Technologies (FAST)*, pp. 375–390, 2017.

[39] J. González and M. Bjørling, "Multi-tenant i/o isolation with open-channel ssds," in *Nonvolatile Memory Workshop (NVMW)*, 2017.

[40] J. Kim, E. Lee, and S. H. Noh, "I/o scheduling schemes for better i/o proportionality on flash-based ssds," in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 221–230, IEEE, 2016.

[41] M. Bjørling, J. González, and P. Bonnet, "Lightnvm: The linux open-channel ssd subsystem," in *15th USENIX Conference on File and Storage Technologies (FAST)*, pp. 359–374, 2017.

[42] "Open-Channel Solid State Drives Specification Revision 2.0." http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf.

[43] W. Shin, M. Kim, K. Kim, and H. Y. Yeom, "Providing qos through host controlled flash ssd garbage collection and multiple ssds," in *2015 International Conference on Big Data and Smart Computing (BIGCOMP)*, pp. 111–117, IEEE, 2015.

[44] H. Park, S. Yoo, C.-H. Hong, and C. Yoo, "Storage sla guarantee with novel ssd i/o scheduler in virtualized data centers," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 8, pp. 2422–2434, 2015.

[45] K. Oh, J. Park, and Y. I. Eom, "Weight-based page cache management scheme for enhancing i/o proportionality of cgroups," in *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–3, IEEE, 2019.

GoSeed: Generating an Optimal Seeding Plan for Deduplicated Storage

Aviv Nachman, Gala Yadgar
Computer Science Department, Technion

Sarai Sheinvald
Braude College of Engineering

Abstract

Deduplication decreases the physical occupancy of files in a storage volume by removing duplicate copies of data chunks, but creates data-sharing dependencies that complicate standard storage management tasks. Specifically, data migration plans must consider the dependencies between files that are remapped to new volumes and files that are not. Thus far, only greedy approaches have been suggested for constructing such plans, and it is unclear how they compare to one another and how much they can be improved.

We set to bridge this gap for *seeding*—migration in which the target volume is initially empty. We present GoSeed, a formulation of seeding as an integer linear programming (ILP) problem, and three acceleration methods for applying it to real-sized storage volumes. Our experimental evaluation shows that, while the greedy approaches perform well on "easy" problem instances, the cost of their solution can be significantly higher than that of GoSeed's solution on "hard" instances, for which they are sometimes unable to find a solution at all.

1 Introduction

Data deduplication is one of the most effective ways to reduce the size of data stored in large scale systems. Deduplication consists of identifying duplicate data chunks in different files, storing a single copy of each unique chunk, and replacing the duplicate chunks with pointers to this copy. Deduplication reduces the total physical occupancy, but increases the complexity of management aspects of large-scale systems such as capacity planning, quality of service, and chargeback [53].

Another example, which is the focus of this study, is *data migration*—the task of moving a portion of a physical volume's data to another volume—typically performed for load balancing and resizing. Deduplication complicates the task of determining which files to migrate: the physical capacity freed on the source volume, as well as the physical capacity occupied on the target volume, both depend on the amount of deduplication within the set of migrated files, as well as between them and files outside the set (i.e., files that remain on the source volume and files that initially reside on the target volume). An efficient *migration plan* will free the required space on the source volume while minimizing the space occupied on the target. However, as it turns out, even *seeding*, in which the target volume is initially empty, is a computationally hard problem.

Data migration in deduplicated systems and seeding in particular are the subject of several recent studies, each focusing on a different aspect of the problem. Harnik et al. [32] address capacity estimation for general migration between volumes, while Duggal et al. [25] describe seeding a cloud-tier for an existing system. Rangoli [49] is designed for space reclamation—an equivalent problem to seeding. These studies propose greedy algorithms for determining the set of migrated files, but the efficiency of their resulting migration plans has never been systematically compared. Furthermore, in the absence of theoretical studies of this problem, it is unclear whether and to what extent they can be improved.

We present GoSeed, a new approach that bridges this gap for the seeding and, consequently, space reclamation problems. GoSeed consists of a formulation of seeding as an integer linear programming (ILP) problem, providing a theoretical framework for generating an optimal plan by minimizing its *cost*—the amount of data replicated. Although ILP is known to be NP-Hard, commercial optimizers can solve it efficiently for instances with hundreds of thousands of variables [1–4, 6]. At the same time, ILP instances representing real-world storage systems may consist of hundreds of millions of variables and constraints—too large even for the most efficient optimizers, that may require prohibitively long time to process these instances. Thus, GoSeed also includes three practical acceleration methods, each presenting a different tradeoff between runtime and optimality.

The first method, *solver timeout*, utilizes the optimizer's ability to return a feasible suboptimal solution when its runtime exceeds a predetermined threshold. A larger timeout value allows the optimizer to continue its search for the optimal solution, but increasing the timeout may yield diminishing returns. The second method, *fingerprint sampling*, is similar to the sketches used in [32], and generates an ILP instance from a probabilistically sampled subset of the system's chunks. An optimal seeding plan generated on a sample will not necessarily be optimal for the original system. Thus, increasing the sample size may reduce the plan's cost, but will necessarily increase the required processing time of the solver.

Our third method, *container aggregation*, generates an ILP instance in terms of containers—the basic unit of storage and I/O in many deduplication systems. Containers typically store several hundreds of chunks, where chunks in the same container likely belong to the same files. When they do, containers

represent the same data sharing constraints as their chunks. In addition to reducing the problem size, migrating entire containers can be done without decompressing them, and without increasing the system's fragmentation. At the same time, a container-based ILP instance may introduce "false" sharing between files, resulting in a suboptimal plan.

We implement GoSeed with the Gurobi [2] commercial optimizer, and with the three acceleration methods. We generate seeding plans for volumes based on deduplication snapshots from two public repositories [7, 47]. Our evaluation reveals the limitations of the greedy algorithms proposed for seeding thus far—while they successfully generate good plans for "easy" problems (with modest deduplication), GoSeed generates better solutions for the harder problems, for which the greedy approaches sometimes return no solution.

Our analysis further demonstrates the efficiency of the acceleration methods in GoSeed. It shows that (1) the suboptimal solution returned by GoSeed after a timeout is often better than the greedy solutions, (2) fingerprint sampling "hides" some of the data sharing in volumes with modest deduplication, but provides an accurate representation of systems with substantial deduplication, and (3) GoSeed's container-based solutions are optimal if entire containers are migrated. Our results suggest several rules of thumb for applying and combining these three methods in practical settings.

The rest of this paper is organized as follows. Section 2 provides background on deduplication and ILP, as well as related previous work. We present the ILP formulation of GoSeed in Section 3, and its acceleration methods in Section 4. Our experimental setup and evaluation are described in Section 5, with a discussion in Section 6. Section 7 concludes this work.

2 Background and Related Work

2.1 Deduplication

The smallest unit of data in a deduplication system is a *chunk*, which typically consists of 8-64KB. The incoming data is split into chunks of fixed or variable size, and the *fingerprint* of each chunk is used to identify duplicates and to replace them with pointers to existing copies. In many systems, new chunks are written to durable storage in *containers*, which are the system's I/O unit, and typically consist of hundreds of chunks [20, 29, 38, 41, 68]. New chunks are added to containers in a log structure. Thus, chunks belonging to the same file will likely reside in adjacent containers. Designs that do not employ containers typically also persist the chunks in a log structure, and thus adjacent chunks will likely belong to the same files [16, 18, 24, 54].

To recover a file, all the containers pointed to by the file *recipe* are fetched into memory, after which the file's chunks are collected. The efficiency of this process, in terms of I/O and memory usage, strongly depends on the file's *fragmentation*: the physical location of the different containers and the portion of the container's chunks that belong to the requested file [50]. Some systems reduce the amount of fragmentation by limiting the number of containers a file may point to, or their age [28, 40].

Over the last decade, numerous studies addressed the various aspects of deduplication system design, such as characterizing and estimating the system's deduplication potential [27, 33, 46, 47, 57, 61], efficient chunking and fingerprinting [9, 44, 48, 63, 64], indexing and lookups [10, 54, 68], restore performance [28, 36, 40, 67], compression [42, 65], and security [14, 34, 39, 55]. Their success (among others) has made it possible to use deduplication for primary storage and not just for archives. Additional studies explored ways to adapt the concept of deduplication to related domains such as page caching [35, 38], minimizing network bandwidth [9, 48], management of memory resident VM pages [17, 31, 62], and minimizing flash writes [16, 26, 30, 38, 52, 60].

Recently, Shilane et al. [53] described the "next" challenge in the design of deduplication systems: providing these systems with a set of management functions that are available in traditional enterprise storage systems, one of which is capacity management in deduplicated systems, and specifically, fast and effective data migration.

2.2 Data migration

Data migration is typically performed in the background, according to a *migration plan* specifying which data is moved to which new location. Typical objectives when generating a migration plan include minimizing the amount of data transferred, optimizing load balancing, or minimizing its effect on ongoing jobs.

The effectiveness of data migration and the resources it consumes may greatly affect the system's performance. Thus, efforts have been made to optimize its various aspects including service down-time, geolocation, provisioning, memory consumption, and system-specific performance requirements and constraints [43, 45, 56, 59]. Hippodrome [12] and Ergastulum [13] formulated the storage allocation problem as an instance of bin-packing, while Anderson et al. [11] experimentally evaluated several theoretical algorithms, concluding that their theoretical bounds are overly pessimistic.

The distinction between logical and physical capacity in deduplicated systems introduces additional complexity to the data migration problem. For optimal read and restore performance, the physical copies of a file's chunks must reside on the same storage volume. Thus, when migrating a file from one volume to another, this file's chunks that also belong to another file must be copied (duplicated), rather than moved. As a result, migrating data from a full volume to an empty one is likely to increase the total physical capacity of the system. Migrating data between two non-empty volumes can either increase or decrease the total physical capacity, depending on the degree of duplication between the migrated data and the data on the target volume. Intuitively, to optimize the system's overall storage utilization, a migration plan should minimize the amount of data that is duplicated as a result.

Deduplication complicates other related tasks in a similar manner. Garbage collection must consider the logical as well as the physical relationships between chunks, files, and containers. Unfortunately, specific approaches for optimizing garbage collection are not directly applicable to data migration [23, 28, 40]. As another example, online assignment of streams to servers in distributed systems must consider both content similarity and load balancing. Current solutions distribute data to servers in the granularity of individual chunks [24], super-chunks [21], files [15], or users [22], considering server load as a secondary objective. These online solutions are based on partial knowledge of the data in the system, and may result in suboptimal plans if applied directly to data migration.

2.3 Existing data migration approaches

A recent paper describes the Data Domain Cloud Tier, in which customers maintain two tightly connected deduplication domains, in an on-premises system and in a remote object store [25]. They dedicate special attention to the process of *seeding* the cloud-tier—migrating a portion of the on-premises system into an initially empty object store. While the choice of the exact files to migrate is deferred to the client, the general use-case is to keep older backups in the cloud-tier and newer ones on-premises. The authors refer to "many days or weeks possibly required to transfer a large dataset to the cloud", strongly motivating our goal to minimize the amount of data replicated during migration.

Rangoli is a greedy algorithm for space reclamation in a deduplicated system [49]. Although it predates [25] by several years, its problem formulation is equivalent: choose a set of files for migration from an existing volume to a new empty volume. Rangoli constructs a migration plan by greedily grouping files into roughly equal-sized bins according to the blocks they share, and then chooses for migration the bin whose files have the least amount of data shared with other bins. The migration objective is specified as the number of bins.

In another recent paper, Harnik et al. address migration in the broader context of load balancing [32]. Their system consists of several non-empty volumes, each operating as an independent deduplication domain. The goal is to estimate the amount of deduplication between files on different volumes, to determine the potential occupancy reduction achieved by migrating files between volumes.[1] The focus of the study is a sketching technique that facilitates this estimation. In their evaluation, the authors propose a greedy algorithm that iteratively migrates files from one volume to another, with the goal of minimizing the overall physical occupancy in the system.

Capacity planning and space reclamation in deduplicated systems are relatively new challenges. Current solutions are either naïve—migrating backups according to their age—or greedy. At the same time, migration carries significant costs in

terms of physical capacity and bandwidth consumption, and it is unclear whether and how much the greedy solutions can be improved upon. This gap is the main motivation of our study.

2.4 Integer linear programming (ILP)

Integer linear programming (ILP) is a well-known optimization problem. The input to ILP is a set Ax of linear constraints, each of the form $a_0x_0 + a_1x_1 \cdots + a_{n-1}x_{n-1} \leq c$, where $a_1, \ldots, a_n, c \in \mathbb{Z}$, and an objective function of the form $Tx = t_0x_0 + t_1x_1 + \cdots + t_{n-1}x_{n-1}$. The problem is finding, given Ax and Tx, an integer assignment to $x_0, x_1, \ldots, x_n$ that satisfies A_x and maximizes Tx. There is no known efficient algorithm for solving ILP. In particular, when the variables are restricted to Boolean assignments (0 or 1), then merely deciding whether Ax has a solution has been long known to be NP-Complete [37].
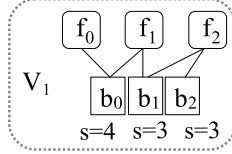
Nevertheless, ILP is used in various fields for modeling a wide range of problems [8, 51, 66, 69]. This wide use has been made possible by efficient *ILP solvers*—designated heuristic-based tools that can handle and solve very large instances. Thus, despite its theoretical hardness, ILP can in many cases be solved in practice for instances that contain hundreds of thousands and even millions of variables and constraints.

Most ILP solvers are based on the Simplex algorithm [19], which efficiently solves linear programming where the variables are not necessarily integers. They then search for an optimal integer solution, starting the search at the vicinity of the non-integer one. The wide variety of ILP solvers includes open-source solvers such as SYMPHONY [6], lp_solve [4], and GNU LP Kit [3]. Industrial tools include IBM CPLEX [1] and the Gurobi optimizer [2]. In this research, we take advantage of these highly-optimized solvers for finding the optimal migration plan in a deduplicated storage system.

3 GoSeed ILP optimization

We formulate the goal of generating a migration plan as follows. *Move physical data of size M from one volume to another, while minimizing R, the total size of the physical data that must be copied (replicated) as a result.* In a *seeding plan*, the target volume is initially empty. We refer to R as the *cost* of the migration. Note that in a seeding plan, minimizing R minimizes the total capacity of the system, as well as the amount of data *transferred* between volumes during the migration.

Problem definition. For a storage volume V, let $B_V = \{b_0, b_1, \ldots, b_{m-1}\}$ be the set of unique blocks stored on V, and let $s(b)$ be the size of block b. The storage cost of the volume is the total size of the blocks stored on it, i.e., $s(V) = \Sigma_{b_i \in B_V} s(b_i)$. Let $F_V = \{f_0, f_1, \ldots, f_{n-1}\}$ be the set of files mapped to V, and let $I_V \subseteq B_V \times F_V$ be an inclusion relation, where $(b, f) \in I_V$ means that block b is included in file f. We intentionally disregard the order of blocks in a file, or blocks that appear several times in one file. While this information is required for restoring the original file, it is irrelevant for the allocation of blocks to volumes.

---

[1]In the original paper, migration is described in terms of moving logical volumes between physical servers. Thus, their volumes are equivalent to what we refer to as files, for simplicity.

(1) $0 \le x_0, x_1, x_2, m_0, m_1, m_2, r_0, r_1, r_2 \le 1$

(2) $m_0 \le x_0, \ m_0 \le x_1, \ m_1 \le x_1, \ m_1 \le x_2, \ m_2 \le x_2$

(3) $x_0 \le m_0 + r_0, \ x_1 \le m_0 + r_0, \ x_1 \le m_1 + r_1,$
    $x_2 \le m_1 + r_1, \ x_2 \le m_2 + r_2$

(4) $4 \cdot m_0 + 3 \cdot m_1 + 3 \cdot m_2 = 3$

Goal: minimize $4 \cdot r_0 + 3 \cdot r_1 + 3 \cdot r_2$

Figure 1: Example system and its formulation as an ILP problem, where the goal is to migrate 30% of the physical space ($M = 3$).

We require that all the blocks included in a file are stored on the volume this file is mapped to. Thus, if a file f is *remapped* from V_1 to V_2, then every block that is included in f must be either *migrated* to V_2 or *replicated*. Similarly, if we migrate a block b from volume V_1 to volume V_2, then every file f such that $(b, f) \in I_{V_1}$ must be remapped from V_1 to V_2.

The ***seeding problem*** is to decide, given a source volume V_1 with $B_{V_1}, F_{V_1}, I_{V_1}$, an empty destination volume V_2, a target size M and a threshold size R, whether there exists a set $B' \subseteq B_{V_1}$ of blocks whose total size is M, that can be migrated from V_1 to V_2, such that the total size of blocks that are replicated is at most R. In practice, we are interested in the respective optimization problem. Namely, the *seeding optimization problem* is to find such a set B' while minimizing R. A solution to the seeding optimization problem is a migration plan: the list of files that are remapped, the list of blocks that are replicated, and B'—the list of blocks that are migrated from V_1 to V_2.

We prove that the seeding problem is NP-hard using a reduction from the Clique problem (omitted due to space considerations). Intuitively, the relationship between files and blocks influences the quality of the solution, because the decision whether to migrate a specific block depends on the decision regarding other blocks. In this aspect, seeding is similar to many other set-selection problems such as Set Cover, Vertex Cover, and Hitting Set, that are known to be NP-hard [37].

ILP formulation. We model the seeding optimization problem as an ILP problem as follows. For every file $f_i \in F_{V_1}$ we allocate a Boolean variable x_i. Assigning 1 to x_i means that f_i is remapped from V_1 to V_2. For every block $b_i \in B_{V_1}$ we allocate two Boolean variables, m_i, r_i. Assigning 1 to m_i means that b_i is migrated from V_1 to V_2, and assigning 1 to r_i means that b_i is replicated and will be stored in both V_1 and V_2.

We model the problem constraints as a set of linear inequalities, as follows.

1. All variables are Boolean: $0 \le x_j \le 1$, $0 \le m_i \le 1$, and $0 \le r_i \le 1$ for every $f_j \in F_{V_1}$ and $b_i \in B_{V_1}$.

2. If a block b is migrated, then every file that b is included in is remapped: $m_i \le x_j$ for every i, j such that $(b_i, f_j) \in I_{V_1}$.
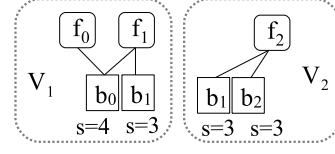
· Block b_2 is migrated: $m_2 = 1$

· File f_2 is remapped: $x_2 = 1$

· Block b_1 is replicated: $r_1 = 1$

· The remaining files and blocks are untouched:
  $x_0 = x_1 = m_0 = m_1 = r_0 = r_2 = 0$

· The total cost is $R = 3 \cdot r_1 = 3$

Figure 2: The system from Figure 1 after applying the optimal migration plan with $M = 3$.

3. If a file f is remapped, then every block that is included in f is either migrated or replicated: $x_j \le m_i + r_i$ for every i, j such that $(b_i, f_j) \in I_{V_1}$.

4. The total size of migrated blocks is M:
   $\Sigma_{b_i \in B_{V_1}} s(b_i) \cdot m_i = M$.

The objective function minimizes the total size of blocks that are replicated: minimize $\Sigma_{b_i \in B_{V_1}} s(b_i) \cdot r_i$.

Another intuitive constraint is that a block cannot be migrated and replicated at the same time: $m_i + r_i \le 1$ for every $b_i \in B_{V_1}$. This constraint will be satisfied implicitly in any optimal solution—if a block is migrated ($m_i = 1$) then replicating it will only increase the value of the objective function, and thus r_i will remain 0. This is also true for all the solutions in the space defined by the Simplex algorithm, and consequently for suboptimal solutions returned when the solver times out.

A solution to the ILP instance is an assignment of values to the Boolean variables. We note, however, that such an assignment does not necessarily exist. If a solution does not exist, Simplex-based solvers will return quickly—we observed a few minutes in our evaluation. If a solution to the ILP instance exists, we find B' by returning every block b_i such that $m_i = 1$, and the list of replicated blocks by returning every block b_i such that $r_i = 1$. The list of files to remap is given by every file f_i such that $x_i = 1$.

Figure 1 shows an example of a simple deduplicated system, and the formulation as an ILP instance of the respective seeding optimization problem with $M = 3$. The optimal solution, depicted in Figure 2, is to migrate b_2, replicate b_1, and remap f_2, which yields $R = 3$. Another feasible solution is to migrate b_1, whose size is also 3. However, migrating b_1 results in replicating both b_0 and b_2, which yields $R = 7$.

Refinements. The requirement to migrate blocks whose total size is exactly M may severely limit the possibility of finding a solution. Fortunately, in real settings, there is some range of acceptable migrated capacities. For example, for the file system in Figure 1, a solution exists for $M = 3$ but not for $M = 2$. In realistic systems, feasible solutions may be easier to find but their cost, R, might be unnecessarily high. Thus, we redefine
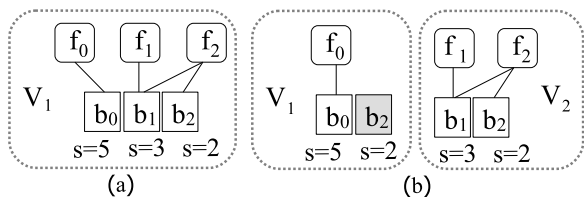
Figure 3: A migration plan with an orphan block. The goal is to migrate 30% ($M = 3$) of the system in (a). b_2 is the orphan—it was duplicated when f_2 was remapped (b).

our problem by adding a *slack value*, ε, as follows.

For a given $B_{V_1}, F_{V_1}, I_{V_1}$, target size M, and slack value ε, the *seeding optimization problem with slack* is to find $B' \subseteq B_{V_1}$ of blocks whose total size is M', $M - \varepsilon \leq M' \leq M + \varepsilon$, that can be migrated from V_1 to V_2. In the formulation as an ILP problem, we require that the total size of migrated blocks is $M \pm \varepsilon$: $M - \varepsilon \leq \Sigma_{b_i \in B_{V_1}} s(b_i) \cdot m_i \leq M + \varepsilon$. For example, for the system in Figure 1, the optimal solution for $M = 2$ and $\varepsilon = 1$, is the solution given above for $M = 3$.

Another refinement in the problem formulation is required to prevent "leftovers" on the source volume V_1. An *orphan* block is copied because a file it is included in is remapped, but no other file that includes it remains in V_1. For example, consider the system in Figure 3(a), with a migration objective of $M = 3$. For simplicity, assume that $\varepsilon = 0$. The only feasible solution is depicted in Figure 3(b), where b_1 is migrated, f_1 and f_2 are remapped, and b_2 is replicated. b_2 cannot be migrated because this would exceed the target migration size, $M = 3$. Replicating b_2 leaves an extra copy of this block in V_1, where it is not contained in any file.

Although a migration plan with orphan blocks represents a feasible solution to the ILP problem, it is an inefficient one. For example, b_2 in Figure 3(b) consists of 20% of the system's original capacity. Orphans can be eliminated by garbage collection, or even as part of the migration process [25]. This is essentially equivalent to migrating the orphan blocks, rather than replicating them, resulting in a migrated capacity which exceeds the original objective. For example, removing b_2 from volume V_2 in Figure 3(b) is equivalent to a migration plan with $M = 5$, rather than the intended $M = 3$.

We eliminate such solutions by adding the following constraint: if a block b is copied, then at least one file it is included in is not remapped: $r_i \leq \Sigma_{\{j | (b_i, f_j) \in I_{V_1}\}} (1 - x_j)$ for every $b_i \in B_{V_1}$. This additional constraint may result in the solver returning without a solution. Such cases should be addressed by increasing ε or modifying M. Nevertheless, the decision whether to prevent orphan blocks in the migration plan or to eliminate them during its execution is a design choice that can easily be realized by adding or removing the above constraint.

Complexity. The number of constraints in the ILP formulation is linear in the size of I_V—the number of pointers from files to blocks in the system. Although the size of I_V can be at most $|B_V| \cdot |F_V|$, it is likely considerably smaller in practice: the majority of the files are small, and the majority of the blocks

are included in a small number of files [47].

In general, the time required for an ILP solver to find an optimal solution depends on many factors, including the number of variables, the connections between them (represented by the constraints), and the number of feasible solutions. In our context, the size of the problem is determined by the number of files and blocks, and its complexity depends on the deduplication ratio and on the pattern of data sharing between the files. It is difficult to predict how each of these factors will affect the solving time in practice. Furthermore, small changes in the target migration size and in the slack value may significantly affect the solver's performance. We evaluate the sensitivity of GoSeed to these parameters in Section 5.

4 GoSeed Acceleration Methods

The challenge in applying ILP solvers to realistic migration problems is their size. In a system with an average chunk size of 8KB, there will be approximately 130M chunks in each TB of physical capacity. Thus, the runtime for generating a migration plan for a source volume with several TBs of data would be unacceptably long. In this section, we present three methods for reducing this generation time. We describe their advantages and limitations and the ways in which they may be combined, and evaluate their effectiveness in Section 5.

4.1 Solver timeout

The runtime of an ILP solver can be limited by specifying a timeout value. When a timeout is reached before the optimal solution is found, the solver will halt and return the best feasible solution found thus far. This approach has the advantage of letting the solver process the unmodified problem. It does not require any preprocessing, and, theoretically, the solver may succeed in finding the optimal solution. The downside is that when the solver is timed out, we cannot necessarily tell how far the suboptimal solution is from the optimal one.

4.2 Fingerprint sampling

Sampling is a standard technique for handling large problems, and has been used in deduplication systems to increase the efficiency of the deduplication process [15, 16, 41], to route streams to servers [21], for estimating deduplication ratios [33], and for managing volume capacities [32]. We use sampling in the same way it is used in [32]. Given a *sampling degree* k, we include in our sample all the chunks whose fingerprint contains k leading zeroes, and all the files containing those chunks. When the fingerprint values are uniformly distributed, the sample will include $\frac{1}{2^k}$ chunks. Harnik et al. show in [32] that $k = 13$ guarantees small enough errors for estimating the capacity of deduplicated volumes larger than 100GB.

Sampling reduces the size of the ILP instance by a predictable factor: incrementing the sampling degree k by one reduces the number of blocks by half. Combining sampling and timeouts presents an interesting tradeoff: a smaller sampling factor results in a larger ILP instance that more accurately represents the sampled system. However, solving a larger instance
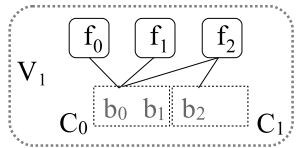
Figure 4: The system from Figure 1 with container aggregation.

is more likely to time out and return a suboptimal solution. It is not clear which combination will result in a better migration plan—a suboptimal solution on a large instance, or an optimal solution on a small instance. Our analysis in Section 5 shows how the answer depends on the original (unsampled) instance and on the length of the timeout.

4.3 Container-based aggregation

Aggregation is often employed as a first step in analysing large datasets. In deduplication systems, containers are a natural basis for aggregation. Containers are often compressed before being written to durable storage, and are decompressed when they are fetched into memory for retrieving individual chunks. Thus, generating and executing a migration plan in the granularity of containers holds the advantage of avoiding decompression as well as an increase in the fragmentation in the system by migrating individual chunks from containers.

To formulate the migration problem with containers we coalesce chunks that are stored in the same container into a single block, and remove parallel edges, i.e., pointers from the same file to different chunks in the same container. Figure 4 shows the container view of the volume from Figure 1. In a real system, formulating the migration problem with containers is more efficient than with chunks: when processing file recipes, we can ignore the chunk fingerprints and use only the container IDs for generating the variables and constraints.

In a system that stores chunks in containers, the container-based migration problem accurately represents the system's original constraints. At the same time, we can further leverage container-based aggregation as an acceleration method by artificially increasing the container size beyond the size used by the system. With *aggregation degree K*, we coalesce every K adjacent containers into one, like we do for chunks. Thus, a system with 4MB containers can be represented as one with $4K$-MB containers by coalescing every K original containers. Containers typically store hundreds of chunks, which means that the size of the resulting ILP problem will be smaller by several orders of magnitude. Furthermore, containers are allocated as fixed-size extents, which further reduces the ILP problem complexity: the optimization goal of minimizing the *total size* of migrated blocks becomes a simpler goal of minimizing their *number*.

A container-based seeding plan can be obtained more quickly than a chunk-based one. Thus, if aggregation is combined with solver timeouts, a container-based suboptimal solution will likely be closer to the optimal (container-based) solution than in an execution solving the chunk-based instance. At the same time, container-based aggregation (like any aggre-

gation method) reduces the granularity of the solution, which affects its efficiency as an acceleration method for the original chunk-based problem. Namely, an optimal container-based migration plan is not necessarily optimal if the migration is executed in the granularity of chunks.

Consider a migration plan generated with containers, and let F_{V_2} be the set of files that are remapped to V_2 as a result of that plan. F_{V_1} is the set of files that remain on V_1. If a container is not part of the migration plan, this means that *all* of its chunks are contained only in files from F_{V_1}. When a container is marked for migration, this means that *all* of its chunks are contained only in files from F_{V_2}. When a container includes *at least one chunk* that is contained in a file from F_{V_1} as well as in a file from F_{V_2}, the entire container is marked for replication. However, this container may also contain some *"false positives"*—chunks that are contained only in files from F_{V_1} (and should not be part of the migration), or only in files from F_{V_2} (and should be migrated rather than replicated).

These false positives increase the cost of the container-based solution, and can be eliminated by performing the actual migration in the granularity of chunks, as done in [25]. However, this would eliminate the advantages of migrating entire containers, and may cause the solver to "miss" the migration plan that would have been optimal for the chunk-based ILP instance. We observe this effect in Section 5

5 Evaluation

The goal of our experimental evaluation is to answer the following questions:
- What is the difference, in terms of cost, between the ILP-based migration plan and the greedy ones?
- How do the ILP instance parameters (its size, M, and ε) affect it's complexity, indicated by the solver's runtime?
- How does timing out the solver affect the quality (cost) of the returned solution?
- How do the sampling and aggregation degrees affect the solver's runtime and the cost of the migration plan?

5.1 Experimental setup

Deduplication snapshots. We use static file system snapshots from two publicly available repositories. The UBC dataset [47] includes file systems of 857 Microsoft employees available via SNIA IOTTA [5]. The FSL dataset [7] includes daily snapshots of a Mac OS X Snow Leopard server and of student home directories at the File System and Storage Lab (FSL) at Stony Brook University [57, 58]. The snapshots include, for each file, the fingerprints calculated for each of its chunks, as well as the chunk size in bytes. Each snapshot file represents one entire file system, which is the migration unit in our model, and is represented as one file in our ILP instance.

To obtain a mapping between files and unique chunks, we emulate the ingestion of each snapshot into a simplified deduplication system. We assume that all duplicates are detected and eliminated. We emulate the assignment of chunks to containers

Volume	Files	Chunks	Dedupe	Containers	Logical
UBC-50	50	27M	0.59	122K	807 GB
UBC-100	100	73M	0.34	317K	3.5 TB
UBC-200	200	138M	0.32	570K	6.7 TB
UBC-500	500	382M	0.31	1.6M	19.5 TB
Homes	81	19M	0.13	295K	8.9 TB
MacOS-Week	102	6M	0.02	93K	20 TB
MacOS-Daily	200	6.3M	0.01	99K	43 TB

Table 1: Volume snapshots in our evaluation. The container size is 4MB. Dedupe is the deduplication ratio—the ratio between the physical and logical size of each volume. Logical is the logical size.

by assuming that unique chunks are added to containers in the order of their appearance in the original snapshot file. We create snapshots of entire volumes by ingesting several file-system snapshots one after the other, thus eliminating duplicates across individual snapshots. The resulting volume snapshot represents an independent deduplication domain.

The volume snapshots used in our experiments are detailed in Table 1. The *UBC-X* volumes contain the first *X* file systems in the UBC dataset. These snapshots were created with variable-sized chunks with Rabin fingerprints, whose specified average chunk size is 64KB. In practice, however, many chunks are 4KB or less. The FSL snapshots were also generated with Rabin fingerprints and average chunk size of 64KB.[2] The *MacOS-Daily* volume contains all available daily snapshots of the server between May 14, 2015 and May 8, 2016, while the *MacOS-Week* volume contains weekly snapshots, which we emulate by ingesting the snapshots from all the Fridays in repository. The Homes volume contains weekly snapshots of nine users between August 28 and October 23, 2014 (nine weeks in total).

***GoSeed* Implementation.** We use the commercial Gurobi optimizer [2] as our ILP solver, and use its `C++` interface to define our problem instances. The problem variables (x_i, m_i, r_i) are declared as Binary and represented by the `GRBVar` data type. The constraints and objective are declared as linear expressions. M and ε are given in units of percents of the physical capacity. We specify three parameters for each execution: a *timeout* value, the *parallelism* degree (number of threads), and a *random seed*. These parameters do not affect the optimality of the solution, but they do affect the solver's runtime. Specifically, the starting point for the search for an integer solution is chosen at random, which may lead some executions to complete earlier than others. If the solver times out, different executions might return solutions with slightly different costs. In our evaluation, we solve each ILP instance in three separate executions, each with a different random seed, and present the average of their execution times and costs. Our wrapper program for converting a volume snapshot into an ILP instance in Gurobi consists of approximately 400 lines of code.[3]

We ran our experiments on a server running Ubuntu 18.04.3,

---

equipped with 64GB 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. We let Gurobi use 38 CPUs, and specify a timeout of six hours, to allow for experiments with a wide range of setup and problem parameters.

Comparison to existing approaches. We use our volume snapshots to evaluate the quality of the migration plans generated by the existing approaches described in Section 2.3. We implement ***Rangoli*** according to the original paper [49]. We convert our migration objective M into a number of bins B, such that $B = \frac{1}{M}$. We modified Rangoli to comply with the restriction that the migrated capacity is between $M - \varepsilon$ and $M + \varepsilon$: when choosing one of B bins for migration, our version of Rangoli chooses only from those bins whose capacity is within the specified bounds.

For evaluation purposes, we implemented a seeding version of the greedy load balancer that was used for evaluating the capacity sketches in [32]. We refer to this algorithm as ***SGreedy***. In each iteration, SGreedy chooses one file from V_1 to remap to V_2. The remapped file is the one which yields the best space-saving ratio, i.e., the ratio between the space freed from V_1 and that added to V_2. The iterations continue until the migrated capacity is at least $M - \varepsilon$, and if, at this point, it does not exceed $M + \varepsilon$, a solution is returned. SGreedy returns a seeding plan in the form of a list of files that are remapped from V_1 to V_2. We then use a dedicated "cost calculator" to derive the cost of the migration plan on the original (unsampled) system.

Our calculator creates an array of the volume's chunks and their sizes, and two bit indicators, V_1 and V_2, that are initialized to FALSE for each chunk. It then traverses the files in the volume snapshot and updates the indicators of their blocks as follows. If a file is remapped, then the V_2 indicators of all its chunks are set to TRUE. If a file is not remapped, then the V_1 indicators of all its chunks are set to TRUE. A final pass over the chunk array calculates the replication cost by summing the sizes of all the chunks whose V_1 and V_2 indicators are both TRUE. The migrated capacity is the sum of the sizes of all the chunks whose V_2 indicator is TRUE and V_1 indicator is FALSE.

5.2 Results

Comparison of different algorithms. We first analyze the migration cost incurred by the different algorithms on the various volume snapshots. Figure 5 shows our results with three values of M (10,20,33) and $\varepsilon = 2$. A missing bar of an algorithm indicates that it did not find a solution for that instance. GoSeed-K and SGreedy-K depict the results obtained by these algorithms running on a snapshot created with sampling degree K (the cost was calculated on the original snapshot).

Rangoli does not perform well on most of the volume snapshots. It incurs the highest replication cost on the UBC snapshots, except UBC-100 with $M = 33$, for which it does not find a solution. On the FSL snapshots, it finds a good solution only

(a) UBC-50   (b) UBC-100   (c) UBC-200   (d) UBC-500
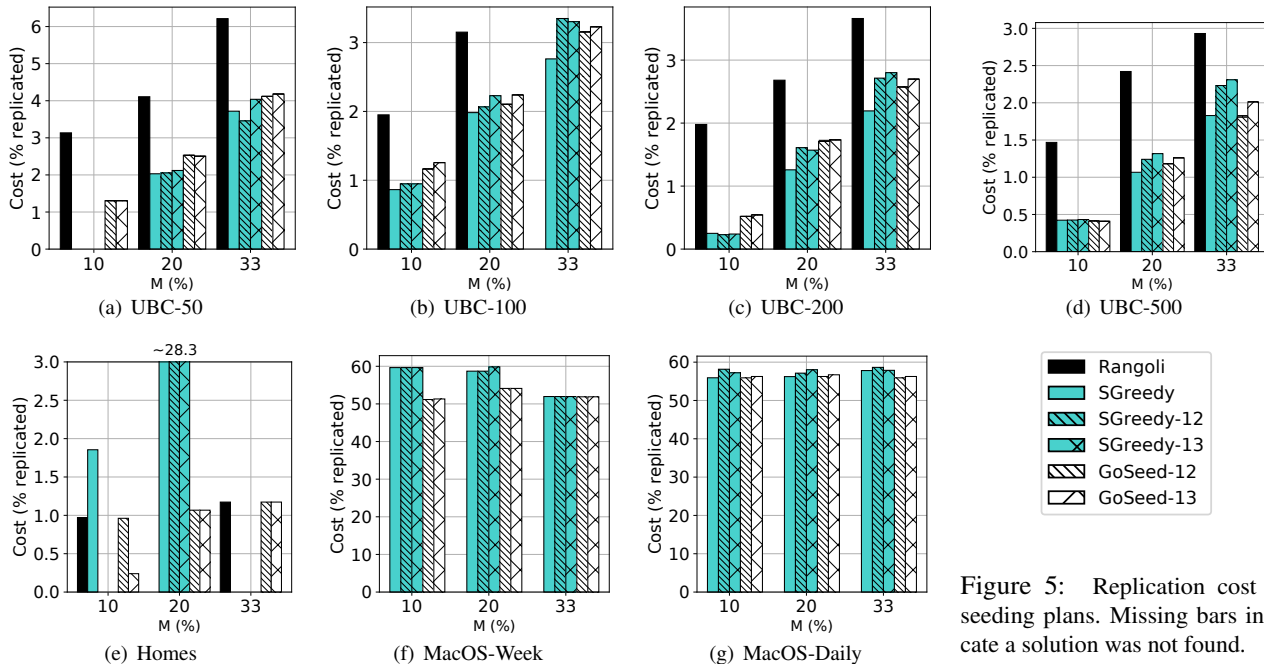
(e) Homes   (f) MacOS-Week   (g) MacOS-Daily

Figure 5: Replication cost of seeding plans. Missing bars indicate a solution was not found.

for the Homes volume (with $M = 10$ and $M = 33$), but not for the remaining instances. The backups on the MacOS volumes share most of their data, with a very low deduplication ratio. In these circumstances, Rangoli fails because it is unable to partition the files into separate bins of the required size.

SGreedy returns a solution in all but two instances (UBC-50 with $M = 10$ and Homes with $M = 33$). For the UBC snapshots, the cost of its solution is 37%-87% lower than the cost of Rangoli's solution. When SGreedy is applied to a sampled snapshot, as it was originally intended, this cost increases by as much as 28% and 27%, for sample degrees 12 and 13, respectively. This increase is expected, as the sampled snapshot "hides" some of the data sharing in the real system. However, the increase is smaller in most instances. It is also interesting to note a few cases where SGreedy returns a better solution (with a lower replication cost) on the sampled snapshot than on the original one, such as for UBC-50 with $M = 33$. These situations can happen when "hiding" some of the sharing helps the greedy process find a solution that it wouldn't find otherwise.

We can now classify our volumes into three rough categories. We refer to the UBC volumes as *easy*—their data sharing is modest and the greedy algorithms find good solutions for them. We refer to the Homes volume as *hard*—its data sharing is substantial and the greedy algorithms mostly return solutions with high costs (up to 29%), or don't find a solution at all. We consider the MacOS volumes to be *very hard* because of their exceptionally high degree of sharing between files. This sharing prevents Rangoli from finding any solution, and incurs very high costs (up to 60%) in the plan generated by SGreedy.

GoSeed cannot find a solution for the full snapshots, which translate to ILP instances with hundreds of millions of constraints. We thus use fingerprint sampling to apply GoSeed

to the volume snapshots, with two sampling degrees, 12 and 13. Our results show that GoSeed finds a solution for all the volumes and all values of M. It generates slightly better plans with a smaller sampling degree, when more of the system's constraints are manifested in the ILP instance.

In the easy (UBC) volumes, the cost of GoSeed's migration plan is similar to that of SGreedy's plan on the sampled snapshots. It is higher for four instances (UBC-50 with $M = 20, 33$, and for UBC-100 and UBC-200 with $M = 10$) and equal or lower for the rest. This shows that greedy solutions may suffice for volumes with modest data sharing between files.

The picture is different for the hard volumes. For Homes, GoSeed consistently finds a better migration plan, while each of the greedy algorithms finds a solution for some values of M but fails to find one for others. The biggest gap between the greedy and optimal solutions occurs for $M = 20$: SGreedy (with and without sampling) replicates 27%-28% of the volume's capacity, while the replication cost of the plan generated by GoSeed is only 1%. This demonstrates a known property of greedy algorithms—their solutions are good enough most of the time, but very bad in the worst case.

Finally, for the very hard (MacOS) volumes, GoSeed finds similar or better solutions than SGreedy, with or without sampling. Although more than 50% of the volume is replicated in all of the migration plans, the replication cost of GoSeed for MacOS-Weekly with $M = 10$ and $M = 20$ is 14% and 8% lower than that of SGreedy, respectively. The exceptionally high degree of sharing in this volume indicates that better solutions likely do not exist. This conclusion was supported in our attempt to apply the "user's" migration plan from [25], remapping the oldest backups (files, in our case) to a new tier. In MacOS-Weekly and MacOS-Daily, remapping the single

(a) UBC-50

(b) UBC-100

(c) UBC-200

(e) Homes

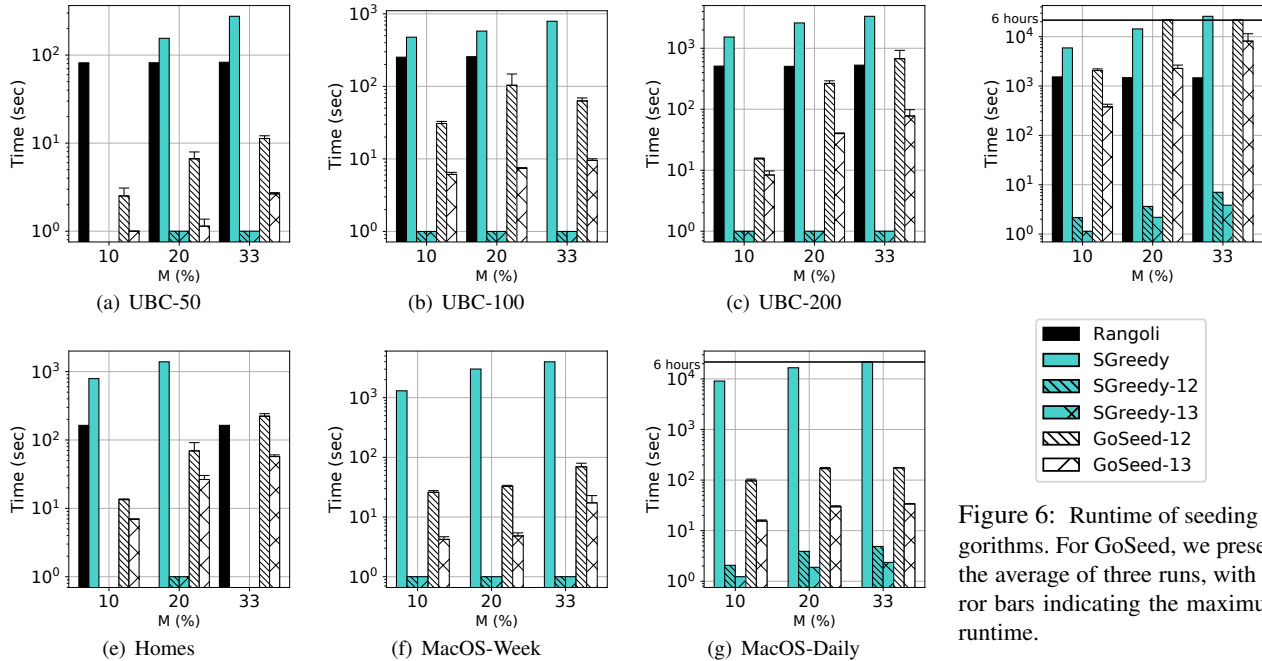(f) MacOS-Week

(g) MacOS-Daily

Figure 6: Runtime of seeding algorithms. For GoSeed, we present the average of three runs, with error bars indicating the maximum runtime.
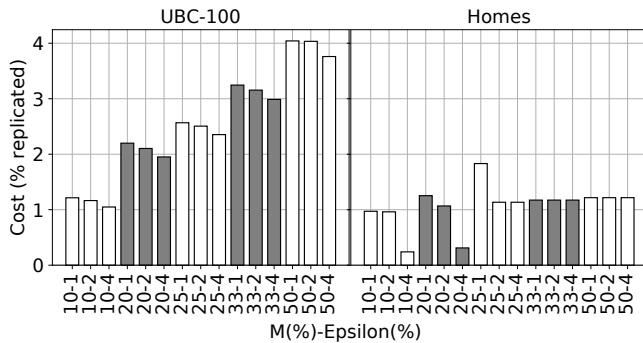

Figure 7: GoSeed plans generated with sampling degree K=12.

oldest backup to a new volume resulted in migrating 0.2% and 0.3% of the volume's capacity, and replicating 49% and 55% of it, respectively.

Figure 6 shows the runtime of the different algorithms. The runtime of GoSeed is longer than that of SGreedy on the sampled snapshot, but shorter than that of SGreedy and Rangoli on the original snapshots. GoSeed timed out at six hours only in one execution (UBC-500 and $K = 12$). The rest of the instances were solved by GoSeed in less than one hour (UBC) or five minutes (FSL). We note, though, that GoSeed utilizes 38 threads, while the greedy algorithms use only one. For a migration plan transferring several TBs of data across a wide area network or a busy interconnect, these runtimes and resources are acceptable.

Effect of ILP parameters. We first investigate how M and ε affect the solver's ability to find a good solution. We compare the cost of the plan generated by GoSeed with five values of M (used in [49]) and three values of ε on an easy (UBC-100) volume and on a hard one (Homes). The results in Figure 7 show that in the easy volume, higher values of M result in a

higher cost, and that this cost can be somewhat reduced by increasing ε, which increases the number of feasible solutions. We observe a similar effect in Homes, but to a much smaller extent. We note that this effect is also shared by the greedy algorithms (not shown for lack of space), for which differences in ε often make a difference between finding a feasible solution or returning without one. Increasing M also exponentially increases the runtime of the solver—migrating more blocks results in more feasible solutions in the search space. We omit the runtimes of this experiment, but the effect can be observed in Figure 6.

We next investigate how the size of the snapshot affects the time required to solve the ILP instance. We compare problems with similar constraints and different sizes by generating sampled snapshots with K between 7 and 13 of the above two volumes. Figure 8 shows the average runtime of GoSeed on these snapshots with $M = 20$ and ε = 2. Error bars mark the minimum and maximum runtimes. Note that both axes are log-scaled—incrementing K by one doubles the number of blocks in the ILP instance. As we expected, the time increases exponentially with the number of blocks. The figure also shows that the runtime of the same instance with one random seed can be as much as $1.45\times$ longer than with another seed. We discuss the implications of this difference below.

Effect of solver timeout. To evaluate the effect of timeouts on the cost of the generated plan, we generate a volume snapshot by sampling UBC-100 with $K = 8$, for which the solver's execution time is approximately four hours. We repeatedly solve this instance (with the same random seed) with increasing timeout values. We set the timeouts to fixed portions of the full runtime, after having measured the complete execution. We repeat this process for three different seeds. To eliminate the
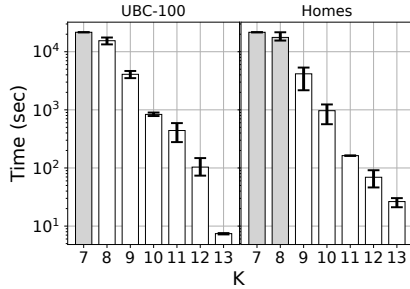
Figure 8: Solving time increases exponentially with instance size (gray bars indicate that the solver timed out).
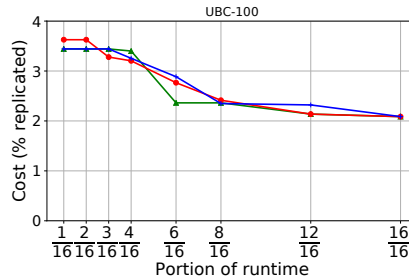
Figure 9: Migration cost decreases when timeout increases (costs are shown for three random seeds).
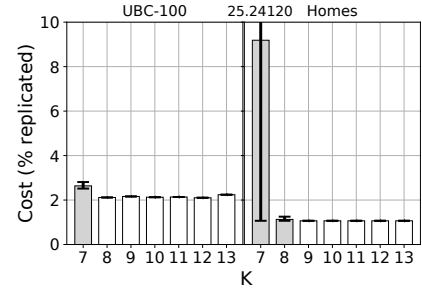
Figure 10: Cost is hardly affected by the sampling degree, unless the instance becomes too large.

effect of sampling, we present the cost of migration assuming the sample represents the entire system.

The results in Figure 9 show that the most substantial cost reduction occurs in the first half of the execution, after which the quality of the solution does not improve considerably. The three processes converge to the same optimal solution at different speeds, corresponding to the different runtimes in Figure 8. At the same time, we note that the largest differences in cost occur between suboptimal solutions returned in the first half of the execution, when the solver makes most of its progress. The cost difference is relatively small and does not exceed 22% (at $\frac{6}{16}$)—a much smaller difference than the difference in time required to find the optimal solution.

Gurobi provides an interface for querying the solver for intermediate results without halting its execution. We did not use this interface because it might compromise the accuracy of our time measurements. However, it can be used to periodically check the rate at which the intermediate solution improves. When the rate decreases and begins to converge, continuing the execution yields diminishing returns, and it can be halted.

Effect of fingerprint sampling. We evaluate the effect of the sampling degree on the cost of the solution by calculating the costs of the plans generated for UBC-100 and Homes with $M = 20$, $\varepsilon = 2$, and K between 7 and 13. Figure 10 shows that the difference between the cost of optimal solutions is very small. However, when the solver times out, the cost of the suboptimal solution can be as much as $24\times$ higher.

Our results for varying the ILP instance parameters and sampling degrees suggest the following straightforward heuristic for obtaining the best seeding plan within a predetermined time frame. Generate a sample of the system with degree between 10 and 13—smaller degrees are better for smaller systems. If the solver times out, increase the sampling degree by one. If the solver completes and there is still time, solve instances with increasing values of ε until the end of the time frame is reached. This process results in a set of solutions that form a Pareto frontier—their cost decreases as their migrated capacity is farther from the original objective M. The final solution should be chosen according to the design objectives of the system.

Efficiency of container-based plans. The container-based aggregation generates a reduced ILP instance which is an accu-

rate representation of the connections between files and containers. This representation can also be used to generate container-based migration plans with Rangoli and SGreedy. Thus, our next experiment compares the costs of GoSeed and the greedy algorithms on the same instances. Our results in Figure 11 show that in these circumstances, GoSeed can reduce the migration cost obtained by Rangoli and SGreedy by as much as 87% and 66%, respectively. These results are not surprising given the size of the ILP instances—they consist of several hundred thousand variables, well within Gurobi's capabilities. As a result, even in experiments in which Gurobi times out (indicated by the small triangles in the figure), its suboptimal solutions are considerably better than the greedy ones. The costs with aggregated containers (GoSeed-C×2) are higher because of the false dependencies described in Section 4.3.

We used our cost calculator to compare the chunk-level cost of the container-based migration plan to the greedy plans generated for the original system (figures omitted due to space considerations). For the MacOS volumes and for UBC-50, GoSeed's container-based plan outperforms Rangoli and is comparable to SGreedy. However, for the larger UBC volumes and for Homes, SGreedy and Rangoli find solutions with as much as $7.6\times$ and $13.6\times$ lower cost, respectively. On these instances, Gurobi returned a suboptimal solution which was close to the container-based optimum, but far from the chunk-based optimum, due to the reasons described in Section 4.3. We therefore recommend using GoSeed with container-based aggregation if the migration is to be performed with entire containers, and with fingerprint sampling otherwise.

6 Discussion

Data migration within a large-scale deduplicated system can reallocate tens of terabytes of data. This data is possibly transferred over a wide area network or a busy interconnect, and some of it might be replicated as a result. The premise of our research is that the potentially high costs of data migration justify solving a complex optimization problem with the goal of minimizing these costs.

Thus, in contrast to existing greedy *heuristics* to this hard problem, GoSeed attempts to *solve* it. By formulating data migration as an ILP instance, GoSeed can "hide" its complexity
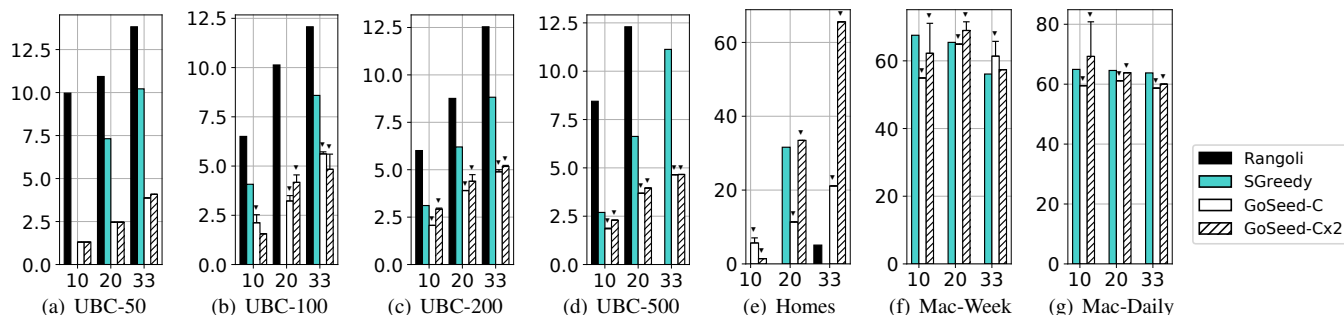
Figure 11: Replication cost of container-based migration plans. For GoSeed, we present the average of three runs (error bars indicate the maximum cost). Triangles indicate experiments in which the solver timed out. GoSeed outperforms the greedy solutions by as much as 87%.

by leveraging off-the-shelf highly optimized solvers. This approach is independent of specific design and implementation details of the deduplication system or the ILP solver. However, it introduces an inherent tradeoff between the time spent generating a seeding plan, and the cost of executing it. As this cost depends on the system's characteristics, such as network speed, cost of storage, and read and restore workload, the potential for cost saving by GoSeed is system dependent as well.

Our evaluation showed that the benefit of GoSeed is high in two scenarios. The first is when the problem's size allows the solver to find the optimal (or near optimal) solution within the allocated time. Container-based migration is an example of this case, where GoSeed significantly reduced the migration cost of the greedy algorithms. The second case is when a high degree of data sharing in the system makes it hard for the greedy solutions to find a good migration plan, causing them to produce a costly solution or no solution at all. At the same time, for systems with low or exceptionally high degrees of data sharing, the greedy solutions and that of GoSeed are comparable.

Accurately identifying the large instances for which GoSeed would significantly improve on the greedy solution is not straightforward, and requires further research. Fortunately, a simple hybrid approach can provide 'the best of both worlds': one can run the greedy algorithm, followed by GoSeed, and execute the migration plan whose cost is lower.

Generalizations. Seeding is the simplest form of data migration in large systems. A natural next step to this work is to generalize our ILP-based approach to more complex migration scenarios, such as migration into a non-empty volume, and migration where both source and target volumes are chosen as part of the plan. Each generalization introduces additional aspects, and might require reformulating not only the ILP constraints, but also its objective function.

For example, when the destination volume is not empty, the optimal migration plan can be the one that minimizes the total storage capacity on the source and destination volumes combined. An alternative formulation might minimize the total amount of data that must be transferred from the source volume to the destination. In the most general case, generating the migration plan also entails determining either the source or the destination volume, or both, such that the migration goal is achieved and the objective is optimized. Data migration in general introduces additional objectives, such load balancing between volumes, or optimizing the migration process under certain network conditions and limitations. The problem can be further extended by allowing some files to be split between volumes, introducing a new tradeoff between the cost of migration and that of file access.

The ILP formulation of these problems will result in considerably more complex instances than those of the seeding problem. As a result, we might need to apply our acceleration methods more aggressively, e.g., by increasing the fingerprint sampling degree, or construct new methods. Thus, each generalization of the seeding problem introduces non-trivial challenges as well as additional tradeoffs between the solving time and the cost of migration.

7 Conclusions

We presented GoSeed, an algorithm for generating theoretically optimal seeding plans in deduplicated systems, and three acceleration methods for applying it to realistic storage volumes. Our evaluation demonstrated the effectiveness of the acceleration methods: GoSeed can produce an optimal seeding plan on a sample of the system in less than an hour, even in cases where the greedy solutions do not find a feasible solution to the problem. When executed on the original system, GoSeed's solution is not theoretically optimal, but it can substantially reduce the cost of the greedy solutions.

Finally, our formulation of data migration as an ILP problem, combined with the availability of numerous ILP solvers, opens up new opportunities for additional contributions in this domain, and for making data migration more efficient.

Acknowledgments

References

[1] CPLEX Optimizer. https://www.ibm.com/analytics/cplex-optimizer. Accessed: 2019-12-29.

[2] The fastest mathematical programming solver. http://www.gurobi.com/. Accessed: 2019-12-29.

[3] GLPK (GNU Linear Programming Kit). https://www.gnu.org/software/glpk/. Accessed: 2019-12-29.

[4] Introduction to lp_solve 5.5.2.5. http://lpsolve.sourceforge.net/5.5/. Accessed: 2019-12-29.

[5] SNIA IOTTA Repository. http://iotta.snia.org/tracetypes/6. Accessed: 2019-12-29.

[6] SYMPHONY development home page. https://projects.coin-or.org/SYMPHONY. Accessed: 2019-12-29.

[7] Traces and snapshots public archive. http://tracer.filesystems.org/. Accessed: 2019-12-29.

[8] Jeph Abara. Applying integer linear programming to the fleet assignment problem. *Interfaces*, 19(4):20–28, 1989.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Eric Anderson, Mahesh Kallahalla, Susan Spence, Ram Swaminathan, and Qiang Wan. *Ergastulum: quickly finding near-optimal storage system designs*. HP Laboratories, June 2002.

[14] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage*, 9(4):12:1–12:33, November 2013.

[15] 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.

[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 *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[17] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. CMD: Classification-based memory deduplication through page access characteristics. In *10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 14)*, 2014.

[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] George B. Dantzig. *Linear programming and extensions*. Rand Corporation Research Study. Princeton Univ. Press, Princeton, NJ, 1963.

[20] 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.

[21] 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.

[22] Fred Douglis, Deepti Bhardwaj, Hangwei Qian, and Philip Shilane. Content-aware load balancing for distributed backup. In *25th International Conference on Large Installation System Administration (LISA 11)*, 2011.

[23] 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.

[24] 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.

[25] 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.

[26] EMC Corporation. *INTRODUCTION TO THE EMC XtremIO STORAGE ARRAY (Ver. 4.0)*, rev. 08 edition, April 2015.

[27] Jingxin Feng and Jiri Schindler. A deduplication study for host-side caches in virtualized data center environments. In *29th IEEE Symposium on Mass Storage Systems and Technologies (MSST 13)*, 2013.

[28] 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.

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

[30] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[31] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *8th USENIX Conference on Operating Systems Design and Implementation (OSDI 08)*, 2008.

[32] 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.

[33] Danny Harnik, Ety Khaitzin, and Dmitry Sotnikov. Estimating unseen deduplication-from theory to practice. In *14th Usenix Conference on File and Storage Technologies (FAST 16)*, 2016.

[34] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security Privacy*, 8(6):40–47, Nov 2010.

[35] Charles B. Morrey III and Dirk Grunwald. Content-based block caching. In *23rd IEEE Symposium on Mass Storage Systems and Technologies (MSST 06)*, 2006.

[36] 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.

[37] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[38] 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.

[39] Jin Li, Xiaofeng Chen, Mingqiang Li, Jingwei Li, Patrick PC Lee, and Wenjing Lou. Secure deduplication with efficient and reliable convergent key management. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1615–1625, June 2014.

[40] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use in-line chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013.

[41] 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.

[42] 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.

[43] 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.

[44] Udi Manber. Finding similar files in a large file system. In *USENIX Winter 1994 Technical Conference (WTEC 94)*, 1994.

[45] Keiichi Matsuzawa, Mitsuo Hayasaka, and Takahiro Shinagawa. The quick migration of file servers. In *11th ACM International Systems and Storage Conference (SYSTOR 18)*, 2018.

[46] Dirk Meister, Jürgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A study on data deduplication in HPC storage systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC 12)*, 2012.

[47] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *9th USENIX Conference on File and Stroage Technologies (FAST 11)*, 2011.

[48] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *18th ACM Symposium on Operating Systems Principles (SOSP 01)*, 2001.

[49] P. C. Nagesh and Atish Kathpal. Rangoli: Space management in deduplication environments. In *6th International Systems and Storage Conference (SYSTOR 13)*, 2013.

[50] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David H. C. Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *2011 IEEE International Conference on High Performance Computing and Communications (HPCC 11)*, 2011.

[51] A. Richards and J. P. How. Aircraft trajectory planning with collision avoidance using mixed integer linear programming. In *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)*, volume 3, pages 1936–1941, May 2002.

[52] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide page deduplication in virtual environments. In *21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC 12)*, 2012.

[53] 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.

[54] 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.

[55] Mark W. Storer, Kevin Greenan, Darrell D.E. Long, and Ethan L. Miller. Secure data deduplication. In *ACM International Workshop on Storage Security and Survivability (StorageSS '08)*, 2008.

[56] John D. Strunk, Eno Thereska, Christos Faloutsos, and Gregory R. Ganger. Using utility to provision storage systems. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, 2008.

[57] 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.

[58] 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.

[59] 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.

[60] Carl A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review - OSDI '02*, 36(SI):181–194, December 2002.

[61] 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.

[62] Nai Xia, Chen Tian, Yan Luo, Hang Liu, and Xiaoliang Wang. UKSM: Swift memory deduplication via hierarchical and adaptive memory region distilling. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.

[63] 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.

[64] 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.

[65] 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.

[66] Yanhua Zhang, X. Sun, and Baowei Wang. Efficient algorithm for k-barrier coverage based on integer linear programming. *China Communications*, 13(7):16–23, July 2016.

[67] 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.

[68] 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.

[69] Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. Coverage-based trace signal selection for fault localisation in post-silicon validation. In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference (HVC 12)*, 2012.

Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook

Zhichao Cao[†‡]   Siying Dong[‡]   Sagar Vemuri[‡]   David H.C. Du[†]

[†]University of Minnesota, Twin Cities   [‡]Facebook

Abstract

Persistent key-value stores are widely used as building blocks in today's IT infrastructure for managing and storing large amounts of data. However, studies of characterizing real-world workloads for key-value stores are limited due to the lack of tracing/analyzing tools and the difficulty of collecting traces in operational environments. In this paper, we first present a detailed characterization of workloads from three typical RocksDB production use cases at Facebook: UDB (a MySQL storage layer for social graph data), ZippyDB (a distributed key-value store), and UP2X (a distributed key-value store for AI/ML services). These characterizations reveal several interesting findings: first, that the distribution of key and value sizes are highly related to the use cases/applications; second, that the accesses to key-value pairs have a good locality and follow certain special patterns; and third, that the collected performance metrics show a strong diurnal pattern in the UDB, but not the other two.

We further discover that although the widely used key-value benchmark YCSB provides various workload configurations and key-value pair access distribution models, the YCSB-triggered workloads for underlying storage systems are still not close enough to the workloads we collected due to ignorance of key-space localities. To address this issue, we propose a key-range based modeling and develop a benchmark that can better emulate the workloads of real-world key-value stores. This benchmark can synthetically generate more precise key-value queries that represent the reads and writes of key-value stores to the underlying storage system.

1 Introduction

In current IT infrastructure, persistent key-value stores (KV-stores) are widely used as storage engines to support various upper-layer applications. The high performance, flexibility, and ease of use of KV-stores have attracted more users and developers. Many existing systems and applications like file systems, object-based storage systems, SQL databases, and even AI/ML systems use KV-stores as backend storage to achieve high performance and high space efficiency [10, 16, 28, 36].

However, tuning and improving the performance of KV-stores is still challenging. First, there are very limited studies of real-world workload characterization and analysis for KV-stores, and the performance of KV-stores is highly related to the workloads generated by applications. Second, the analytic methods for characterizing KV-store workloads are different from the existing workload characterization studies for block storage or file systems. KV-stores have simple but very different interfaces and behaviors. A set of good workload collection, analysis, and characterization tools can benefit both developers and users of KV-stores by optimizing performance and developing new functions. Third, when evaluating underlying storage systems of KV-stores, it is unknown whether the workloads generated by KV-store benchmarks are representative of real-world KV-store workloads.

To address these issues, in this paper, we characterize, model, and benchmark workloads of RocksDB (a high-performance persistent KV-store) at Facebook. To our knowledge, this is the first study that characterizes persistent KV-store workloads. First, we introduce a set of tools that can be used in production to collect the KV-level query traces, replay the traces, and analyze the traces. These tools are open-sourced in RocksDB release [20] and are used within Facebook for debugging and performance tuning KV-stores. Second, to achieve a better understanding of the KV workloads and their correlations to the applications, we select three RocksDB use cases at Facebook to study: 1) UDB, 2) ZippyDB, and 3) UP2X. These three use cases are typical examples of how KV-stores are used: 1) as the storage engine of a SQL database, 2) as the storage engine of a distributed KV-store, and 3) as the persistent storage for artificial-intelligence/machine-learning (AI/ML) services.

UDB is the MySQL storage layer for social graph data at Facebook, and RocksDB is used as its backend storage engine. Social graph data is maintained in the MySQL tables, and table rows are stored as KV-pairs in RocksDB. The conversion from MySQL tables to RocksDB KV-pairs is achieved by MyRocks [19, 36]. ZippyDB is a distributed KV-store that uses RocksDB as the storage nodes to achieve data persistency and reliability. ZippyDB usually stores data like photo metadata and the metadata of objects in storage. In this paper, the workloads of ZippyDB were collected from shards that store the metadata of an object storage system at Facebook

(called *ObjStorage* in this paper). The key usually contains the metadata of an ObjStorage file or a data block, and the value stores the corresponding object address. UP2X is a special distributed KV-store based on RocksDB. UP2X stores the profile data (e.g., counters and statistics) used for the prediction and inferencing of several AI/ML services at Facebook. Therefore, the KV-pairs in UP2X are frequently updated.

Based on a set of collected workloads, we further explore the specific characteristics of KV-stores. From our analyses, we find that 1) read dominates the queries in UDB and ZippyDB, while read-modify-write (Merge) is the major query type in UP2X; 2) key sizes are usually small and have a narrow distribution due to the key composition design from upper-layer applications, and large value sizes only appear in some special cases; 3) most KV-pairs are cold (less accessed), and only a small portion of KV-pairs are frequently accessed; 4) Get, Put, and Iterator have strong key-space localities (e.g., frequently accessed KV-pairs are within relatively close locations in the key-space), and some key-ranges that are closely related to the request localities of upper-layer applications are extremely hot (frequently accessed); and 5) the accesses in UDB explicitly exhibit a diurnal pattern, unlike those in ZippyDB and UP2X, which do not show such a clear pattern.

Benchmarks are widely used to evaluate KV-store performance and to test underlying storage systems. With real-world traces, we investigate whether the existing KV benchmarks can synthetically generate real-world-like workloads with storage I/Os that display similar characteristics. YCSB [11] is one of the most widely used KV benchmarks and has become the gold standard of KV-store benchmarking. It provides different workload models, various query types, flexible configurations, and supports most of the widely used KV-stores. YCSB can help users simulate real-world workloads in a convenient way. However, we find that even though YCSB can generate workloads that have similar key-value (KV) query statistics as shown in ZippyDB workloads, the RocksDB storage I/Os can be quite different. This issue is mainly caused by the fact that the YCSB-generated workloads ignore key-space localities. In YCSB, hot KV-pairs are either randomly allocated across the whole key-space or clustered together. This results in an I/O mismatch between accessed data blocks in storage and the data blocks associated with KV queries. Without considering key-space localities, a benchmark will generate workloads that cause RocksDB to have a bigger read amplification and a smaller write amplification than those in real-world workloads.

To develop a benchmark that can more precisely emulate KV-store workloads, we propose a workload modeling method based on the hotness of key-ranges. The whole key-space is partitioned into small key-ranges, and we model the hotness of these small key-ranges. In the new benchmark, queries are assigned to key-ranges based on the distribution of key-range hotness, and hot keys are allocated closely in each key-range. In our evaluation, under the same configura-

tion, YCSB causes at least 500% more read-bytes and delivers only 17% of the cache hits in RocksDB compared with real-world workloads. The workloads generated by our proposed new benchmark have only 43% more read-bytes and achieve about 77% of the cache hits in RocksDB, and thus are much closer to real-world workloads. Moreover, we use UDB as an example to show that the synthetic workloads generated by the new benchmark have a good fit of the distributions in key/value sizes, KV-pair accesses, and Iterator scan lengths.

This paper is organized as follows. First, we introduce RocksDB and the system background of three RocksDB use cases in Section 2. We describe our methodology and tools in Section 3. The detailed workload characteristics of the three use cases including the general query statistics, key and value sizes, and KV-pair access distributions are presented in 4, 5, and 6 respectively. In Section 7, we present the investigation results of the storage statistics of YCSB, and describe the proposed new modeling and benchmarking methods. We also compare the results of YCSB with those of the new benchmark. Related work is described in Section 8, and we conclude the paper in Section 9.

2   Background

In this section, we first briefly introduce KV-stores and RocksDB. Then, we provide background on three RocksDB use cases at Facebook, UDB, ZippyDB, and UP2X, to promote understanding of their workloads.

2.1   Key-Value Stores and RocksDB

KV-store is a type of data storage that stores and accesses data based on {key, value} pairs. A key uniquely identifies the KV-pair, and the value holds the data. KV-stores are widely used by companies as distributed hash tables (e.g., Amazon Dynamo [14]), in-memory databases (e.g., Redis [39]), and persistent storage (e.g., BigTable [8] from Google and RocksDB [15, 21] from Facebook).

RocksDB is a high-performance embedded persistent KV-store that was derived from LevelDB [23] by Facebook [15, 21] and was optimized for fast storage devices such as Solid State Drives. RocksDB is also used by many large websites, like Alibaba [44], Yahoo [37], and LinkedIn [24]. At Facebook, RocksDB is used as the storage engine for several data storage services, such as MySQL [19, 36], Laser [9], Cassandra [16], ZippyDB [1], and AI/ML platforms.

RocksDB supports KV interfaces like Get, Put, Delete, Iterator (scan), SingleDelete, DeleteRange, and Merge. Get, Put, and Delete are used to read, write, and delete a KV-pair with certain key respectively. Iterator is used to scan a set of consecutive KV-pairs beginning with a *start-key*. The scan direction can be either forward (calling Nexts) or backward (calling Prevs). SingleDelete can only be used to delete a KV-pair that has not been overwritten [22]. DeleteRange is used to delete a range of keys between [start, end) (the end-key is excluded from the deletion). RocksDB encapsulates
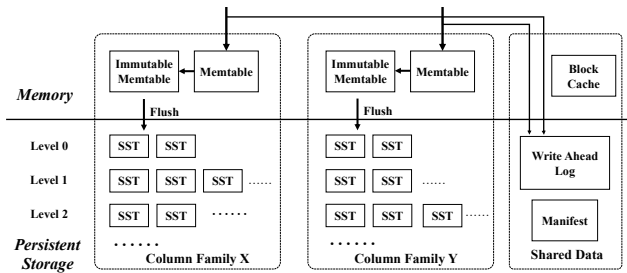
Figure 1: The basic architecture of RocksDB.

the semantics for read-modify-write into a simple abstract interface called Merge [17], which avoids the performance overhead of a random Get before every Put. Merge stores the delta of the write to RocksDB, and these deltas can be stacked or already combined. This incurs a high read overhead because a Get to one key requires finding and combining all the previously stored deltas with the same key inserted by a Merge. The combine function is defined by users as a RocksDB plugin.

RocksDB adopts a Log-Structured Merge-Tree [38] to maintain the KV-pairs in persistent storage (e.g., file systems). The basic architecture of RocksDB is shown in Figure 1. One RocksDB maintains at least one logical partition called *Column Family* (CF), which has its own in-memory write buffer (Memtable). When a Memtable is full, it is flushed to the file system and stored as a *Sorted Sequence Table* (SST) file. SST files persistently store the KV-pairs in a sorted fashion and are organized in a sequence of levels starting from Level-0. When one level reaches its limit, one SST file is selected to be merged with the SST files in the next level that have overlapping key-ranges, which is called *compaction*. Detailed information about RocksDB is described in [15, 21]

2.2 Background of Three RocksDB Use Cases

We discuss three important and large-scale production use cases of RocksDB at Facebook: 1) **UDB**; 2) **ZippyDB**; and 3) **UP2X**. Sharding is used in all three use cases to achieve load balancing. Therefore, the workloads are very similar among all shards, and we randomly select three RocksDB instances from each use case to collect the traces.

UDB: Social graph data at Facebook is persistently stored in UDB, a sharded MySQL database tier [4]. The cache read misses and all writes to social graph data are processed by UDB servers. UDB relies on the MySQL instance to handle all queries, and these queries are converted to RocksDB queries via MyRocks [19, 36]. Much of the social graph data is presented as *objects* and *associations*, and is maintained in different MySQL tables following the model introduced in [4]. RocksDB uses different Column Families (CFs) to store object- and association-related data.

There are 6 major CFs in UDB: Object, Assoc, Assoc_count, Object_2ry, Assoc_2ry, and Non_SG. Object stores social graph object data and Assoc stores

social graph association data, which defines connections between two objects. Assoc_count stores the number of associations of each object. Association counters are always updated with new values and do not have any deletions. Object_2ry and Assoc_2ry are the CFs that maintain the secondary indexes of objects and associations, respectively. They are also used for the purpose of ETL (Extract, Transform, and Load data from databases). Non_SG stores data from other non-social graph related services.

Because the UDB workload is an example of KV queries converted from SQL queries, some special patterns exist. We collected the traces for 14 days. Since the workload characteristics of three UDB servers are very similar, we present only one of them. The total trace file size in this server is about 1.1 TB. For some characteristics, daily data is more important. Thus, we also analyzed the workload of the last day in the 14-day period (24-hour trace) separately.

ZippyDB: A high-performance distributed KV-store called ZippyDB was developed based on RocksDB and relies on Paxos [29] to achieve data consistency and reliability. KV-pairs are divided into shards, and each shard is supported by one RocksDB instance. One of the replicas is selected as the primary shard, and the others are secondary. The primary shard processes all the writes to a certain shard. If strong consistency is required for reads, read requests (e.g., *Get* and *Scan*) are only processed by the primary shard. One ZippyDB query is converted to a set of RocksDB queries (one or more).

Compared with the UDB use case, the upper-layer queries in ZippyDB are directly mapped to the RocksDB queries, and so the workload characteristics of ZippyDB are very different. We randomly selected three primary shards of ZippyDB and collected the traces for 24 hours. Like UDB, we present only one of them. This shard stores the metadata of ObjStorage, which is an object storage system at Facebook. In this shard, a KV-pair usually contains the metadata information for an ObjStorage file or a data block with its address information.

UP2X: Facebook uses various AI/ML services to support social networks, and a huge number of dynamically changing data sets (e.g., the statistic counters of user activities) are used for AI/ML prediction and inferencing. UP2X is a distributed KV-store that was developed specifically to store this type of data as KV-pairs. As users use Facebook services, the KV-pairs in UP2X are frequently updated, such as when counters increase. If UP2X called Get before each Put to achieve a read-modify-write operation, it would have a high overhead due to the relatively slow speed of random Gets. UP2X leverages the RocksDB Merge interface to avoid Gets during the updates.

KV-pairs in UP2X are divided into shards supported by RocksDB instances. We randomly selected three RocksDB instances from UP2X and then collected and analyzed the 24-hour traces. Note that the KV-pairs inserted by Merge are cleaned during compaction via *Compaction Filter*, which uses custom logic to delete or modify KV-pairs in the background during compaction. Therefore, a large number of KV-pairs

are removed from UP2X even though the delete operations (e.g., Delete, DeleteRange, and SingleDelete) are not used.

3 Methodology and Tool Set

To analyze and characterize RocksDB workloads from different use cases and to generate synthetic workloads, we propose and develop a set of KV-store tracing, replaying, analyzing, modeling, and benchmarking tools. These tools are already open-sourced in RocksDB release [20]. In this section, we present these tools and discuss how they are used to characterize and generate KV-store workloads.

Tracing The tracing tool collects query information at RocksDB public KV interfaces and writes to a trace file as records. It stores the following five types of information in each trace record: 1) query type, 2) CF ID, 3) key, 4) query specific data, and 5) timestamp. For *Put* and *Merge*, we store the value information in the query-specific data. For Iterator queries like *Seek* and *SeekForPrev*, the scan length (the number of *Next* or *Prev* called after *Seek* or *SeekForPrev*) is stored in the query-specific data. The timestamp is collected when RocksDB public interfaces are called with *microsecond* accuracy. In order to log the trace record of each query in a trace file, a lock is used to serialize all the queries, which will potentially incur some performance overhead. However, according to the performance monitoring statistics in production under the regular production workloads, we did not observe major throughput degradation or increased latency caused by the tracing tool.

Trace Replaying The collected trace files can be replayed through a Replayer tool implemented in *db_bench* (special plugins like MergeOperator or Comparator are required if they are used in the original RocksDB instance). The replay tool issues the queries to RocksDB based on the trace record information, and the time intervals between the queries follow the timestamps in the trace. By setting different fast forward and multithreading parameters, RocksDB can be benchmarked with workloads of different intensities. However, query order is not guaranteed with multithreading. The workloads generated by Replayer can be considered as real-world workloads.

Trace Analyzing Using collected traces for replaying has its limitations. Due to the potential performance overhead of workload tracing, it is difficult to track large-scale and long-lasting workloads. Moreover, the content of trace files is sensitive and confidential for their users/owners, so it is very hard for RocksDB users to share the traces with other RocksDB developers or developers from third-party companies (e.g., upper-layer application developers or storage vendors) for benchmarking and performance tuning. To address these limitations, we propose a way of analyzing RocksDB workloads that profiles the workloads based on information in the traces.

The trace analyzing tool reads a trace file and provides the following characteristics: 1) a detailed statistical summary of the KV-pairs in each CF, query numbers, and query types; 2) key size and value size statistics; 3) KV-pair popularity;

4) the key-space locality, which combines the accessed keys with all existing keys from the database in a sorted order; and 5) Queries Per Second (QPS) statistics.

Modeling and Benchmarking We first calculate the Pearson correlation coefficients between any two selected variables to ensure that these variables have very low correlations. In this way, each variable can be modeled separately. Then, we fit the collected workloads to different statistical models to find out which one has the lowest fitting error, which is more accurate than always fitting different workloads to the same model (like Zipfian). The proposed benchmark can then generate KV queries based on these probability models. Details are discussed in Section 7.

4 General Statistics of Workloads

In this section, we introduce the general workload statistics of each use case including query composition in each CF, KV-pair hotness distributions, and queries per second.

4.1 Query Composition

By analyzing query composition, we can figure out query intensity, the ratio of query types in different use cases, and the popularity of queries. We find that: **1) Get is the most frequently used query type in UDB and ZippyDB, while Merge dominates the queries in UP2X, and 2) query composition can be very different in different CFs**.

UDB In this UDB server, over 10.2 billion queries were called during the 14-day period, and there were about 455 million queries called during the last 24 hours. There are six CFs being used in UDB as discussed in 2.2. Although those CFs are stored in the same RocksDB database, the workloads are very different. It is difficult to analyze and model such a mixed workload without the separation of different CFs. The query composition in each CF is shown in Figure 2. Get, Put, and Iterator are three major query types in UDB, especially in `Object`, `Assoc`, and `Non_SG`. Get does not show up in the secondary indexes of objects (`Object_2ry`) and associations (`Assoc_2ry`). `Object_2ry` is built for the purpose of ETL, so Iterator is the major query type. `Assoc` mostly checks the existence of an association between two objects via Get, while the secondary index (`Assoc_2ry`) lists the objects that are associated with one target object. Since KV-pairs in `Assoc_2ry` have no repeating updates, SingleDelete is used in this CF to delete the invalid KV-pairs. In other CFs, regular Delete is called to remove the invalid KV-pairs. `Assoc_count` stores the number of associations of each object. Therefore, Get and Put are the major query types used in this CF to read and update the counters.

ZippyDB There is only one CF being used in ZippyDB. Get, Put, Delete, and Iterator_seek (forward Iterator) are the four query types that are used. Over the 24-hour period, there were about 420 million queries called in this shard. The ratios of each query type are: 78% Get, 13% Put, 6% Delete, and 3%
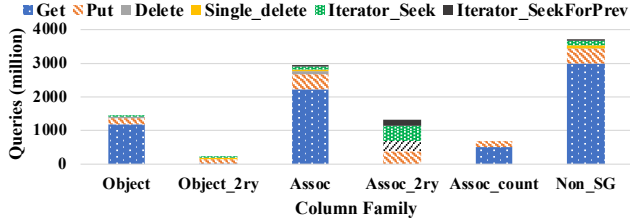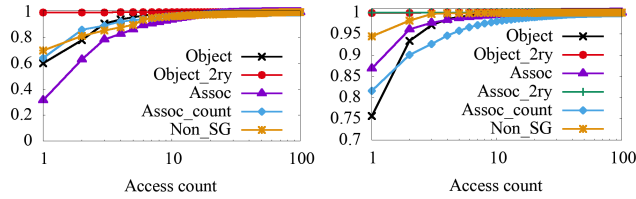
Figure 2: Distribution of different query types in 14 days.

(a) The KV-pair access count CDF by Get   (b) The KV-pair access count CDF by Put

Figure 3: The KV-pair access count distribution queried by Get and Put in each CF during 24 hours.

Iterator, respectively. Get is the major query type in ZippyDB, which aligns with the read-intensive workload of ObjStorage.

UP2X Over the 24-hour period, the RocksDB instance received 111 million queries. Among them, about 92.53% of the queries are Merge, 7.46% of them are Get, and fewer than 0.01% of the queries are Put. The query composition is very different from the UDB and ZippyDB use cases, which are read dominated. About 4.71 million KV-pairs were accessed by Merge, 0.47 million by Get, and 282 by Put. Read-and-modify (Merge) is the major workload pattern in UP2X.

4.2 KV-Pair Hotness Distribution

To understand the hotness of KV-pairs in each use case, we count how many times each KV-pair was accessed during the 24-hour tracing period and show them in cumulative distribution function (CDF) figures. The X-axis is the access count, and the Y-axis is the cumulative ratio between 0 and 1. We find that **in UDB and ZippyDB, most KV-pairs are cold**.

UDB We plot out the KV-pair access count CDFs for Get and Put. For Iterator, we show the start-key access count distribution and the scan length distribution. The CDFs of Get and Put are shown in Figure 3. Looking at Figure 3(a), more than 70% of the KV-pairs in Assoc are Get requests that occurred at least 2 times. In contrast, this ratio in other CFs is lower than 40%. It indicates that read misses of Assoc happen more frequently than the others. As shown in 3(b), in all CFs, more than 75% of the KV-pairs are Put only one time and fewer than 2% of the KV-pairs are Put more than 10 times. The majority of the KV-pairs are rarely updated.

We plot out the access count CDF of the start-keys of Iterators over the 24-hour period, as shown in Figure 4(a). Most of the start-keys are used only once, which shows a low access locality. Fewer than 1% of the start-keys are used multiple times
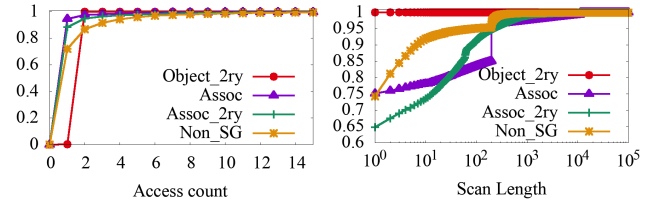
(a) The Iterator start-key access count (b) The Iterator scan length CDF distribu-
CDF distribution   tion

Figure 4: The Iterator scan length and start-key access count CDF of four CFs during 24 hours.

by Iterators. The scan length of more than 60% of the Iterators is only 1 across all CFs, as shown in Figure 4(b). About 20% of the Iterators in Assoc scan more than 100 consecutive keys, while the ratios for Assoc_2ry and Non_SG are about 10% and 5%, respectively. A very large scan length (higher than 10,000) is very rare, but we can still find some examples of this type in Non_SG and Assoc. The configured range query limit in MySQL creates some special scan lengths. For example, there is a jump at 200 in both Assoc and Non_SG.

We also count the number of unique keys being accessed in different time periods. As shown in Table 1, during the last 24 hours, fewer than 3% of the keys were accessed. During the 14-day period, the ratio is still lower than 15% for all CFs. In general, most of the keys in RocksDB are "cold" in this use case. On one hand, most read requests are responded to by the upper cache tiers [5, 7]. Only the read misses will trigger queries to RocksDB. On the other hand, social media data has a strong temporal locality. People are likely accessing the most recently posted content on Facebook.

ZippyDB The average access counts per accessed KV-pair of the four query types (Get, Put, Delete, and Iterator_seek) are: 15.2, 1.7, 1, and 10.9, respectively. Read queries (Get and Iterator_seek) show very good locality, while the majority of the KV-pairs are only Put and Deleted once in the last 24-hour period. The access count distribution is shown in Figure 5. For about 80% of the KV-pairs, Get requests only occur once, and their access counts show a long tail distribution. This indicates that a very small portion of KV-pairs have very large read counts over the 24-hour period. About 1% of the KV-pairs show more than 100 Get requests, and the Gets to these KV-pairs are about 50% of the total Gets that show strong localities. In contrast, about 73% of the KV-pairs are Put only once, and fewer than 0.001% of the KV-pairs are Put more than 10 times. Put does not have as clear a locality as Get does. The CDF of Iterator_seek start-key access counts has a special distribution that we can observe very clearly through the 4 "steps" in the figure. About 55% of the KV-pairs are used as the start-key of Iterator_seek 1 time, 6% of the KV-pairs 11 times, 11% of the KV-pairs 12 times, 5% of the KV-pairs 13 times, 10% of the KV-pairs 23 times, and 10% of the KV-pairs 46 times. The special access count distribution of start-keys is caused by the metadata scanning requests in ObjStorage. For

Table 1: The ratios of KV-pairs among all existing KV-pairs being accessed during different time periods in UDB

CF name	Object	Object_2ry	Assoc	Assoc_2ry	Assoc_count	Non_SG
24 hours	2.72%	0.62%	1.55%	1.75%	0.77%	1.38%
14 days	14.14%	6.10%	13.74%	10.37%	14.05%	11.29%
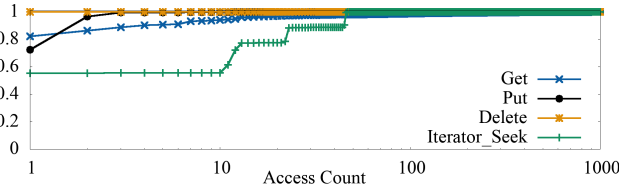
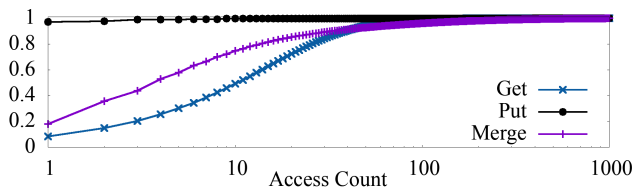Figure 5: The KV-pair access count distribution of ZippyDB.

Figure 6: The access count distribution of UP2X.

example, if one KV-pair stores the metadata of the first block of a file, it will always be used as the start-key of Iterator_seek when the whole file is requested.

UP2X The CDF distribution of KV-pair access counts is shown in Figure 6. Merge and Get have wide distributions of access counts. If we define a KV-pair accessed 10 times or more during the 24-hour period as a hot KV-pair, about 50% of the KV-pairs accessed by Get and 25% of the KV-pairs accessed by Merge are hot. On the other hand, the ratio of very hot KV-pairs (accessed 100 times or more in the 24-hour period) for Merge is 4%, which is much higher than that of Get (fewer than 1%). Both Merge and Get have a very long tail distribution, as shown in the figure.

4.3   QPS (Queries Per Second)

The QPS metric shows the intensiveness of the workload variation over time. **The QPS of some CFs in UDB have strong diurnal patterns, while we can observe only slight QPS variations during day and night time in ZippyDB and UP2X. The daily QPS variations are related to social network behaviors.**

UDB The QPS of UDB is shown in Figure 7. Some CFs (e.g., Assoc and Non_SG) and some query types (e.g., Get and Put) have strong diurnal patterns due to the behaviors of Facebook users around the world. As shown in Figure 7(a), the QPS for either Get or Put usually increases from about 8:00 PST and reaches a peak at about 17:00 PST. Then, the QPS quickly drops and reaches its nadir at about 23:00 PST. The QPS of Delete, SingleDelete, and Iterator shows variations, but it is hard to observe any diurnal patterns. These queries are triggered by Facebook internal services, which

have low correlation with user behaviors. The QPS of six CFs are shown in Figure 7(b). Assoc and Non_SG have a strong diurnal variation, but the QPS of Non_SG is spikier. Since ETL requests are not triggered by Facebook users, the QPS of Object_2ry is spiky and we cannot find any clear patterns.

ZippyDB The QPS of ZippyDB is different from that of UDB. The QPS of ZippyDB varies over the 24-hour period, but we do not find a diurnal variation pattern, especially for Put, Delete, and Iterator_Seek. Since ObjStorage is an object stored at Facebook, object read is related to social network behaviors. Therefore, the QPS of Get is relatively lower at night and higher during the day (based on Pacific Standard Time). Because range queries (Iterator_Seek) are usually not triggered by Facebook users, the QPS for this query type is stable and is between 100 and 120 most of the time.

UP2X The QPS of either Get or Put in UP2X does not have a strong diurnal variation pattern. However, the usage of Merge is closely related to the behavior of Facebook users, such as looking at posts, likes, and other actions. Therefore, the QPS of Merge is relatively lower at night (about 1000) and higher during the day (about 1500).

5   Key and Value Sizes

Key size and value size are important factors in understanding the workloads of KV-stores. They are closely related to performance and storage space efficiency. The average (AVG) and standard deviation (SD) of key and value sizes are shown in Table 2, and the CDFs of key and value sizes are shown in Figure 8. In general, **key sizes are usually small and have a narrow distribution, and value sizes are closely related to the types of data. The standard deviation of key sizes is relatively small, while the standard deviation of value size is large. The average value size of UDB is larger than the other two**.

UDB The average key size is between 16 and 30 bytes except for Assoc_2ry, which has an average key size of 64 bytes. The keys in Assoc_2ry consist of the 4-byte MySQL table index, two object IDs, the object type, and other information. Therefore, the key size of Assoc_2ry is usually larger than 50 bytes and has a long tail distribution as shown in Figure 8(a). For other CFs, the keys are composed of the 4-byte MySQL table index as the prefix, and 10 to 30 bytes of primary or secondary keys like object IDs. Thus, the keys show a narrow distribution. Note that the key sizes of a very small number of KV-pairs are larger than 1 KB, which is not shown in the key size CDF due to the X-axis scale limit.

The value size distribution is shown in Figure 8(b). Object
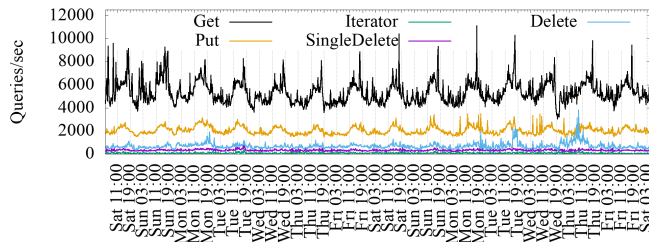
(a) Overall QPS for each query type at different dates and times in a 14-day time span
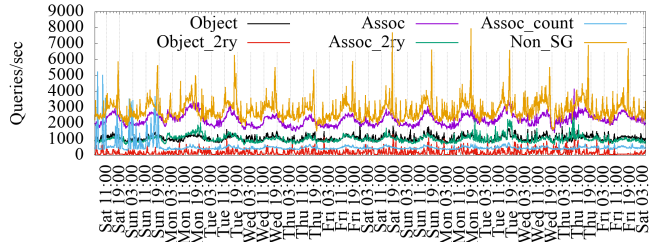
(b) Overall QPS of each CF at different dates and times in a 14-day time span

Figure 7: The QPS variation at different dates and times in a 14-day time span.

Table 2: The average key size (AVG-K), the standard deviation of key size (SD-K), the average value size (AVG-V), and the standard deviation of value size (SD-V) of UDB, ZippyDB, and UP2X (in bytes)

	AVG-K	SD-K	AVG-V	SD-V
UDB	27.1	2.6	126.7	22.1
ZippyDB	47.9	3.7	42.9	26.1
UP2X	10.45	1.4	46.8	11.6

and `Assoc` have a long tail distribution. The value sizes of `Object` vary from 16 bytes to 10 KB, and more than 20% of the value sizes are larger than 1 KB. The average value size of KV-pairs in `Object` is about 1 KB and the median is about 235B, which is much larger than those in other CFs. User data, like the metadata of photos, comments, and other posted data, is stored in this CF, which leads to a large value size. In `Assoc`, the value sizes are relatively small (the average is about 50 bytes) and vary from 10 bytes to 200 bytes.

A very special case is `Assoc_count`, whose key size and value size are exactly 20 bytes. According to the design of this CF, the key is 20 bytes (*bigint* association ID) and is composed of a 10-byte counter and 10 bytes of metadata. Since all the information used in secondary index CFs (`Assoc_2ry` and `Object_2ry`) is stored in its key, the value does not contain any meaningful data. Therefore, the average value size is less than 8 bytes and there are only three possible value sizes in the distribution (1 byte, 6 bytes, or 16 bytes) as shown in Figure 8(b). For CFs with large value sizes like `Object`, optimizations like separating key and value [32] can effectively improve performance.

ZippyDB Since a key in ZippyDB is composed of ObjStorage metadata, the key sizes are relatively large. The CDF of the key sizes is shown in Figure 8(c). We can find several "steps" in the CDF. **Nearly all of the key sizes are in the two size ranges: [48, 53] and [90, 91].** The ratio of KV-pairs in these two key size ranges are different for different query types. For example, about 60% of the key sizes of Get are in the [48, 53] range, while the ratio for Put is about 31%.

The value sizes are collected from Put queries. As shown in Figure 8(d), the value size distribution has a very long tail: about 1% of the value sizes are larger than 400 bytes, and
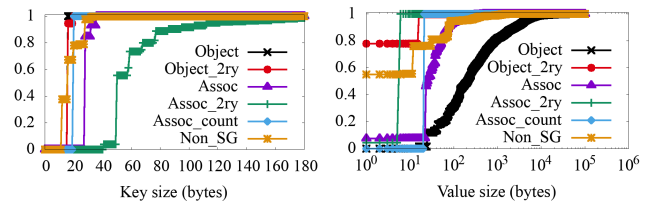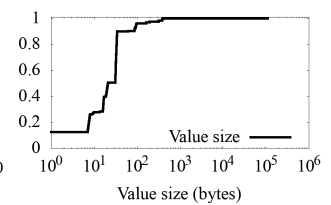

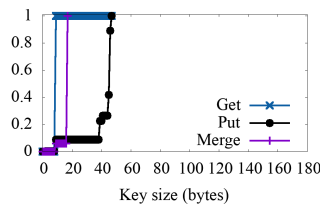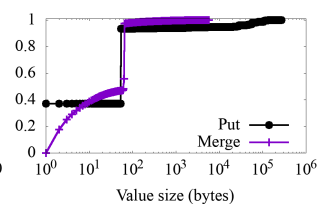(a) UDB key size CDF

(b) UDB value size CDF
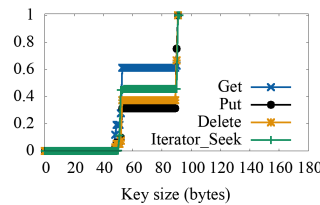
(c) ZippyDB key size CDF

(d) ZippyDB value size CDF

(e) UP2X key size CDF

(f) UP2X value size CDF

Figure 8: The key and value size distributions of UDB, ZippyDB, and UP2X.

about 0.05% of the value sizes are over 1 KB. Some of the value sizes are even larger than 100 KB. However, most of the KV-pairs have a small value. More than 90% of the value sizes are smaller than 34 bytes, which is even smaller than the key sizes.

UP2X The key sizes do not have a wide distribution, as shown in Figure 8(e). **More than 99.99% of the KV-pairs accessed by Get have a key size of 9 bytes**. About 6% of the KV-pairs inserted by Merge have a key size of 9 bytes, and 94% are 17 bytes. The 17-byte KV-pairs are all cleaned during compaction, and they are never read by upper-layer applications through Get. Put is rarely used in UP2X. Among the 282 KV-pairs inserted by Put, about 8.9% of the key sizes
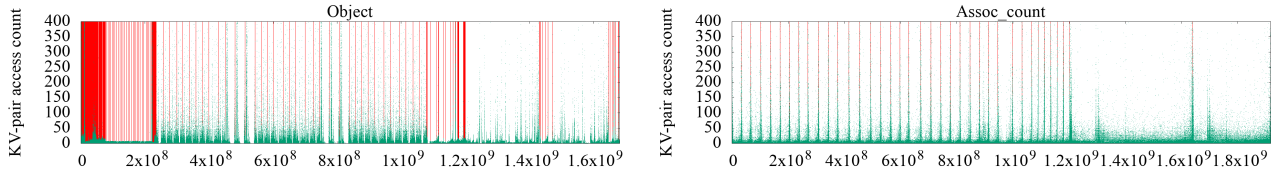
Figure 9: The heat-map of Get in `Object` and `Assoc_count` during a 24-hour period. The X-axis represents the key-ID of keys in the whole key-space, and the Y-axis represents the KV-pair access counts. The red vertical lines are the MySQL table boundaries.

are smaller than 10 bytes, and 47% of them are 46 bytes.

The value size distribution is shown in Figure 8(f). The value sizes of some KV-pairs inserted by Put are extremely large. The average is about 3.6 KB, and about 3% of the KV-pairs are over 100 KB. The value sizes of KV-pairs inserted by Merge have a special distribution. About 40% of the values are smaller than 10 bytes, and about 52% of the values are exactly 64 bytes. A large portion of the updates in UP2X are the counters and other structured information. Thus, the value sizes of those KV-pairs are fixed to 64 bytes.

6 Key-Space and Temporal Patterns

KV-pairs in RocksDB are sorted and stored in SST files. In order to understand and visualize the key-space localities, we sort all the existing keys in the same order as they are stored in RocksDB and plot out the access count of each KV-pair, which is called the *heat-map* of the whole key-space. Each existing key is assigned a unique integer as its *key-ID*, based on its sorting order and starting from 0. We refer to these *key-IDs* as the *key sequence*.

The KV-pair accesses show some special temporal patterns. For example, some KV-pairs are intensively accessed during a short period of time. In order to understand the correlation between temporal patterns and key-space locality, we use a time series sequence to visualize these patterns. We sort the keys in ascending order and assign them with key-IDs as previously discussed, and this *key sequence* is used as the X-axis. The Y-axis shows the time when a query is called. To simplify the Y-axis value, we shift the timestamp of each query to be relative to the tracing start time. Each dot in the time series figure represents a request to a certain key at that time. In the UDB use case, the first 4 bytes of a key are the MySQL table index number due to the key composition of MyRocks. We separate the key-space into different key-ranges that belong to different tables by red vertical lines.

The heat-maps of the three use cases show a strong key-space locality. Hot KV-pairs are closely located in the key-space. The time series figures of Delete and SingleDelete for UDB and Merge for UP2X show strong temporal locality. For some query types, KV-pairs in some key-ranges are intensively accessed during a short period of time.
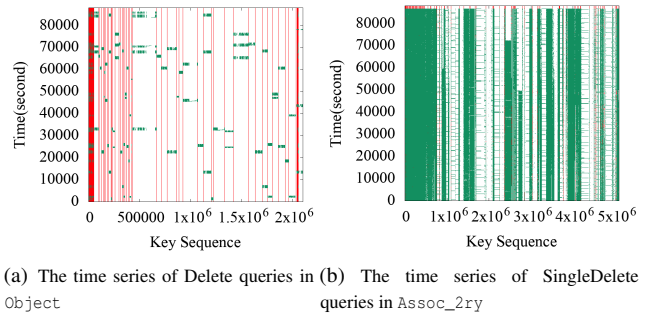

(a) The time series of Delete queries in `Object`

(b) The time series of SingleDelete queries in `Assoc_2ry`

Figure 10: The time series figure of a 24-hour trace.

UDB We use the heat-map of Get in `Object` and `Assoc_count` over a 24-hour period as an example to show the key-space localities. As shown in Figure 9, hot KV-pairs (with high access counts) are usually located in a small key-range and are close to each other. That is, they show a strong key-space locality (indicated by the dense green areas). **Some MySQL tables (the key-ranges between the red vertical lines) are extremely hot (e.g., the green dense area in `Object`), while other tables have no KV-pair accesses.** One interesting characteristic is that the KV-pairs with high access counts in `Assoc_count` are skewed toward the end of the table. In social graphs, new objects are assigned with relatively larger IDs, and new associations are frequently added to the new objects. Therefore, new KV-pairs in `Assoc_count` are hot and are usually at the end of the MySQL table. Moreover, the heat-maps of Get and Put are similar. Usually, the keys with the most Get queries are the ones with the most Put queries.

Most KV-pairs are deleted only once, and they are unlikely to be reinserted. Therefore, there are no hot KV-pairs in Delete and SingleDelete queries. However, they show some special patterns. For example, some nearby KV-pairs are deleted together in a short period of time as shown in Figure 10.

In Figure 10(a), the deleted KV-pairs in the same table for `Object` are removed together in a short period of time (indicated by green dots with close Y values). After that, deletions will not happen for a long period of time. Similar patterns also appear in the SingleDelete time series for `Assoc_2ry`, as shown in Figure 10(b). In some MySQL tables, SingleDelete is intensively called in several short time intervals to remove
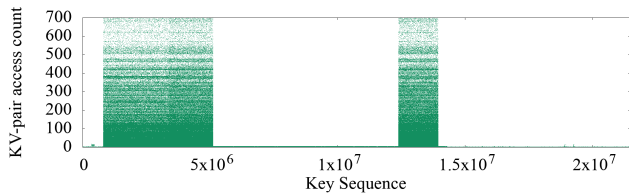
Figure 11: Heat-map of KV-pairs accessed by Get in Zip-pyDB.

Figure 12: Heat-map of KV-pairs accessed by Get in UP2X.

Figure 13: The time series of Merge in UP2X.

KV-pairs in the same table. Between any two sets of intensive deletions, SingleDelete is never called, which causes the "green blocks" in the time series figures.

In general, KV-pairs are not randomly accessed in the whole key-space. The majority of KV-pairs are not accessed or have low access counts. Only a small portion of KV-pairs are extremely hot. These patterns appear in the whole key-space and also occur in different key-ranges. KV-pairs belonging to the same MySQL table are physically stored together. Some SST files at different levels or data blocks in the same SST file are extremely hot. Thus, the compaction and cache mechanisms can be optimized accordingly.

ZippyDB The heat-map of Get in ZippyDB shows a very good key-space locality. For example, as shown in Figure 11, **the KV-pairs accessed by Get have high access counts and are concentrated in several key-ranges (e.g., between** 1×10^6 **and** 5×10^6**).** Hot KV-pairs are not randomly distributed: instead, these KV-pairs are concentrated in several small key-ranges. The hotness of these key-ranges is closely related to cache efficiency and generated storage I/Os. The better a key-space locality is, the higher the RocksDB block cache hit ratio will be. Data blocks that are associated with hot key-ranges will most likely be cached in the RocksDB block cache. These data blocks are actually cold from a storage point of view. With a good key-space locality, the number of data block reads from SST files will be much lower than a random distribution. A similar locality is also found in the Put and Iterator_seek heat-maps. Since all the KV-pairs are deleted once, we did not observe any key-space locality for Delete. In general, the ZippyDB workload is read-intensive and has very good key-space locality.

UP2X If we look at the heat-map of all KV-pairs accessed by Get as shown in Figure 12, **we can find a clear boundary between hot and cold KV-pairs**. Note that the whole key-space was collected after the tracing was completed. In the heat-map, the KV-pairs from 0 to about 550,000 are never accessed by Gets, but the KV-pairs from 550,000 to 900,000 are frequently accessed. A similar locality is also shown in the heat-map of Merge. While KV-pairs from 0 to about 550,000 are sometimes accessed by Merge, their average access counts are much lower than those of the KV-pairs from 550,000 to 900,000. This special locality might be caused by a unique behavior of AI/ML services and their data update patterns.

The UP2X use case shows a very strong key-space locality and temporal locality in Merge. However, about 90% of
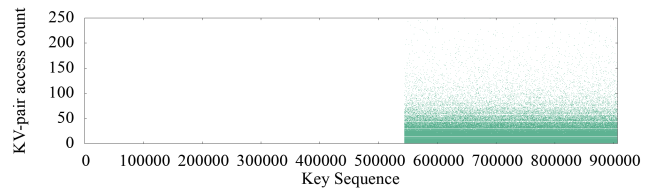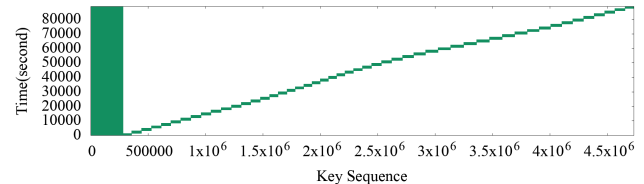
the KV-pairs inserted by Merge are actually cleaned during compaction. Since the key-space heat-map does not show the existence of KV-pairs cleaned by compactions, we plot out the time series sequence for Merge, which can indicate Merge accesses of all KV-pairs. As shown in Figure 13, KV-pairs between 0 and 250,000 are frequently accessed during the 24-hour period. These are KV-pairs between 0 and 900,000 in the whole key-space. The KV-pairs between 250,000 and 4,700,000 show very special key-space and temporal localities. The green blocks indicate that **a small range of KV-pairs are intensively called by Merge during half an hour**. After that, a new set of KV-pairs (with incrementally composed keys) are intensively accessed by Merge during the next half an hour. These KV-pairs are cleaned during compactions. Get and Put do not have similar temporal and key-space localities.

7 Modeling and Benchmarking

After understanding the characteristics of some real-world workloads, we further investigate whether we can use existing benchmarks to model and generate KV workloads that are close to these realistic workloads. We do not consider deletions in our current models.

7.1 How Good Are the Existing Benchmarks?

Several studies [6, 26, 47] use YCSB/db_bench + LevelDB/RocksDB to benchmark the storage performance of KV-stores. Researchers usually consider the workloads generated by YCSB to be close to real-world workloads. YCSB can generate queries that have similar statistics for a given query type ratio, KV-pair hotness distribution, and value size distribution as those in realistic workloads. However, it is unclear whether their generated workloads can match the I/Os for underlying storage systems in realistic workloads.

To investigate this, we focus on storage I/O statistics such as *block reads*, *block cache hits*, *read-bytes*, and *write-bytes* collected by *perf_stat* and *io_stat* in RocksDB. To exclude other factors that may influence the storage I/Os, we replay

the trace and collect the statistics in a clean server. The benchmarks are also evaluated in the same server to ensure the same setup. To ensure that the RocksDB storage I/Os generated during the replay are the same as those in production, we replay the trace on a snapshot of the same RocksDB in which we collected the traces. The snapshot was made at the time when we started tracing. YCSB is a benchmark for NoSQL applications and ZippyDB is a typical distributed KV-store. Therefore, the workloads generated by YCSB are expected to be close to the workloads of ZippyDB, and we use ZippyDB as an example to investigate. Due to special plugin requirements and the workload complexities of UDB and UP2X, we did not analyze storage statistics for those two use cases.

Before we run YCSB, we set the YCSB parameters of `workloada` and `workloadb` to fit ZippyDB workloads as much as possible. That is, we use the same cache size, ensure that the request distribution and scan length follows Zipfian, set the fieldlength as the average value size, and use the same Get/Put/Scan ratios as those shown in Section 4. Since we cannot configure the compression ratio in YCSB to make it the same as ZippyDB, we use the default configuration in YCSB. We normalize the results of the RocksDB storage statistics based on those from the trace replay.

The number of block reads from YCSB is at least 7.7x that of the replay results, and the amount of read-bytes is about 6.2x. The results show an extremely high read amplification. Although the collected amount of write-bytes from YCSB is about 0.74x that of the replay, the actual amount of write-bytes is much lower if we assume YCSB achieves the same compression ratio as ZippyDB (i.e., if the YCSB compression ratio is 4.5, the amount of write-bytes is about 0.41x that of the replay). Moreover, the number of block cache hits is only about 0.17x that of the replay results. This evaluation shows that, even though the overall query statistics (e.g., query number, average value size, and KV-pair access distribution) generated by YCSB are close to those of ZippyDB workloads, the RocksDB storage I/O statistics are actually quite different. db_bench has a similar situation.

Therefore, using the benchmarking results of YCSB as guidance for production might cause some misleading results. For example, the read performance of RocksDB under a production workload will be higher than what we tested using YCSB. The workload of YCSB can easily saturate the storage bandwidth limit due to its extremely high read amplification. Also, the write amplification estimated from the YCSB benchmarking results are lower than in real production. The write performance can be overestimated and might also lead to incorrect SSD lifetime estimates.

With detailed analyses, we find that the main factor that causes this serious read amplification and fewer storage writes is the ignorance of key-space locality. RocksDB reads data blocks (e.g., 16 KB) instead of a KV-pair from storage to memory when it encounters a cache miss. In YCSB, even though the overall KV-pair hotness follows the real-world

workload distribution, the hot KV-pairs are actually randomly distributed in the whole key-space. The queries to these hot KV-pairs make a large number of data blocks hot. Due to the cache space limit, a large number of hot data blocks that consist of the requested KV-pairs will not be cached, which triggers an extremely large number of block reads. In contrast, in ZippyDB, hot KV-pairs only appear in some key-ranges, so the number of hot data blocks is much smaller. Similarly, a random distribution of hot KV-pairs causes more updated KV-pairs to be garbage collected in the newer levels during compactions. Therefore, old versions of cold KV-pairs that are being updated are removed earlier in the newer levels, which leads to fewer writes when compacting older levels. In contrast, if only some key-ranges are frequently updated, old versions of cold KV-pairs are continuously compacted to the older levels until they are merged with their updates during compactions. This causes more data to be written during compactions.

7.2 Key-Range Based Modeling

Unlike workloads generated by YCSB, real-world workloads show strong key-space localities according to the workload characteristics presented in Sections 6. Hot KV-pairs are usually concentrated in several key-ranges. Therefore, to better emulate a real-world workload, we propose a key-range based model. The whole key-space is partitioned into several smaller key-ranges. Instead of only modeling the KV-pair accesses based on the whole key-space statistics, we focus on the hotness of those key-ranges.

How to determine the key-range size (the number of KV-pairs in the key-range) is a major challenge of key-range based modeling. If the key-range is extremely large, the hot KV-pairs are still scattered across a very big range. The accesses to these KV-pairs may still trigger a large number of data block reads. If the key-range is very small (e.g., a small number of KV-pairs per range), hot KV-pairs are actually located in different key-ranges, which regresses to the same limitations as a model that does not consider key-ranges. Based on our investigation, when the key-range size is close to the average number of KV-pairs in an SST file, it can preserve the locality in both the data block level and SST level. Therefore, we use average number of KV-pairs per SST file as key-range size.

We first fit the distributions of key sizes, value sizes, and QPS to different mathematical models (e.g., Power, Exponential, Polynomial, Webull, Pareto, and Sine) and select the model that has the minimal fit standard error (FSE). This is also called the root mean squared error. For example, for a collected workload of ZippyDB, the key size is fixed at either 48 or 90 bytes, the value sizes follow a Generalized Pareto Distribution [25], and QPS can be better fit to Cosine or Sine in a 24-hour period with very small amplitude.

Then, based on the KV-pair access counts and their sequence in the whole key-space, the average accesses per KV-pair of each key-range is calculated and fit to the distribution

model (e.g., power distribution). This way, when one query is generated, we can calculate the probability of each key-range responding to this query. Inside each key range, we let the KV-pair access count distribution follow the distribution of the whole key-space. This ensures that the distribution of the overall KV-pair access counts satisfies that of a real-world workload. Also, we make sure that hot KV-pairs are allocated closely together. Hot and cold key-ranges can be randomly assigned to the whole key-space, since the locations of key-ranges have low influence on the workload locality.

Based on these models, we further develop a new benchmark using db_bench. When running the benchmark, the QPS model controls the time intervals between two consecutive queries. When a query is issued, the query type is determined by the probability of each query type calculated from the collected workload. Then, the key size and value size are determined by the probability function from the fitted models. Next, based on the access probability of each key-range, we choose one key-range to respond to this query. Finally, according to the distribution of KV-pair access counts, one KV-pair in this key range is selected, and its key is used to compose the query. In this way, the KV queries are generated by the benchmark and follow the expected statistical models. At the same time, it better preserves key-space locality.

7.3 Comparison of Benchmarking Results

We fit the ZippyDB workload to the proposed model (Delete is excluded) and build a new benchmark called Prefix_dist [20]. To evaluate the effectiveness of key-range-based modeling, we also implement three other benchmarks with different KV-pair allocations: 1) Prefix_random models the key-range hotness, but randomly distributes the hot and cold KV-pairs in each key-range; 2) similar to YCSB, All_random follows the distribution of KV-pair access counts, but randomly distributes the KV-pairs across the whole key-space; and 3) All_dist puts the hot keys together in the whole key-space instead of using a random distribution. All four benchmarks achieve a similar compression ratio as that of ZippyDB.

Similar to the process described in Section 7.1, we configure YCSB *workloada* and *workloadb* to fit the ZippyDB workload as closely as possible. We run YCSB with the following 4 different request distributions: 1) uniform (YCSB_uniform), 2) Zipfian (YCSB_zipfian), 3) hotspot (YCSB_hotspot), and 4) exponential (YCSB_exp). We use the same pre-loaded database (with 50 million randomly inserted KV-pairs that have the same average key and value sizes as those of a real-world workload) for the 8 benchmarks. The RocksDB cache size is configured with the same value as the production setup. We run each test 3 times (the following discussion uses average value) and normalize the results based on that of replay.

Figure 14 compares the I/O statistics of the 8 benchmarks. The total number of block reads and the amount of read-bytes by YCSB_zipfian workloads are at least 500% higher than those of the original replay results. Even worse, the num-
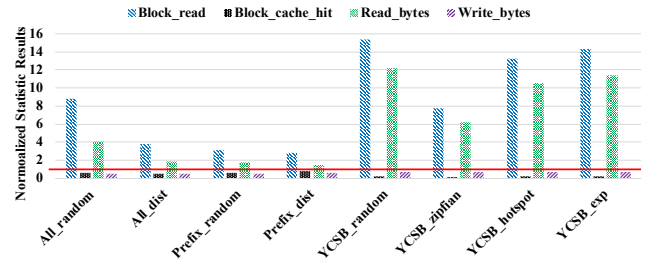

Figure 14: The normalized block read, block cache hit, read-bytes, and write-bytes of benchmarks based on that of the replay. We collected statistics from ZippyDB trace replay results and normalized the statistics from 8 benchmarks. The red line indicates the normalized replay results at 1. The closer the results are to the red line, the better.

ber of block reads and the amount of read-bytes of the other three YCSB benchmarking results are even higher, at 1000% or more compared with the replay results. In contrast, the amount of read-bytes of Prefix_dist are only 40% higher, and are the closest to the original replay results. If we compare the 4 benchmarks we implemented, we can conclude that Prefix_dist can better emulate the number of storage reads by considering key-space localities. All_dist and Prefix_random reduce the number of extra reads by gathering the hot KV-pairs in different granularities (whole key-space level vs. key-range level). Note that if YCSB achieves a similar compression ratio, the RocksDB storage I/Os can be about 35-40% lower. However, this is still much worse than the storage I/Os of All_dist, Prefix_random, and Prefix_dist.

If the same compression ratio is applied, the actual amount of write-bytes by YCSB should be less than 50% of the original replay. Prefix_dist achieves about 60% write-bytes of the original replay. Actually, the mismatch between key/value sizes and KV-pair hotness causes fewer write-bytes compared with the original replay results. In general, YCSB can be further improved by: 1) adding a key-range based distribution model as an option to generate the keys, 2) providing throughput control to simulate the QPS variation, 3) providing key and value size distribution models, and 4) adding the ability to simulate different compression ratios.

7.4 Verification of Benchmarking Statistics

We select the `Assoc` workload from UDB as another example to verify whether our benchmark can achieve KV query statistics that are very similar to those of real-world workloads. Since 90% of keys are 28 bytes and 10% of keys are 32 bytes in `Assoc`, we can use these two fixed key sizes. We find that Generalized Pareto Distribution [25] best fits the value sizes and Iterator scan length. The average KV-pair access count of key-ranges can be better fit in a two-term power model [33, 34], and the distribution of KV-pair access counts follows a power-law that can be fit to the simple power model [33, 34]. As we discussed in Section 4.3, because the QPS variation has a strong diurnal pattern, it can be better fit
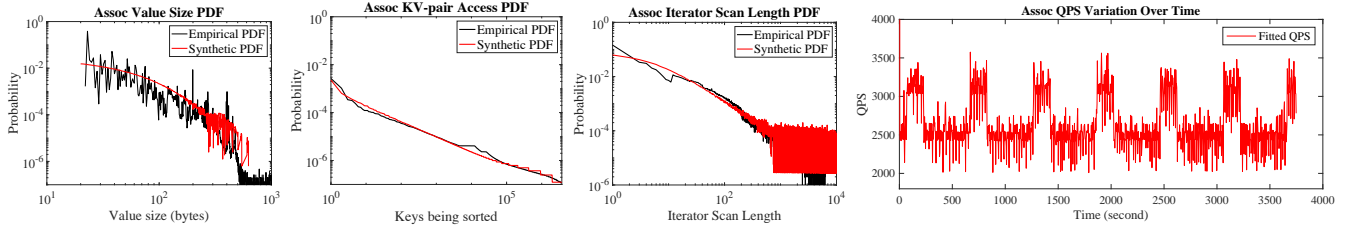
Figure 15: The synthetic workload QPS, and the PDF comparisons between the collected workload and the synthetic workload.

to the Sine model [35] with a 24-hour period.

To compare the workload statistics obtained from benchmarking with those of realistic workloads, we run the new benchmark with a different workload scale: 1) 10 million queries, 2) 30 million existing keys, 3) a 600-second period of QPS *sine*, and 4) a {Get, Put, Iterator} ratio of {0.806, 0.159, 0.035}, respectively (the same as in UDB `Assoc`). We collect the trace during benchmarking and analyze the trace. Figure 15 shows the QPS variation and the probability density function (PDF) comparison of value sizes, KV-pair access counts, and Iterator scan lengths between the UDB `Assoc` workload and the generated workload. Although the scale of the workload generated from our benchmark is different from that of UDB `Assoc`, the PDF figures show that they have nearly the same distribution. This verifies that the generated synthetic workload is very close to the UDB `Assoc` workload in terms of those statistics.

8   Related Work

During the past 20 years, the workloads of storage systems, file systems, and caching systems have been collected and analyzed in many studies. Kavalanekar *et al.* collected block traces from production Windows servers at Microsoft and provided workload characterizations that have benefitted the design of storage systems and file systems tremendously [27]. Riska *et al.* analyzed the disk-level workload generated by different applications [40]. The file system workloads were studied by industrial and academic researchers at different scales [30, 41, 42]. The workloads of the web server caches were also traced and analyzed [2, 3, 43, 46]. While the web cache can be treated as a KV-store, the query types and workloads are different from persistent KV-stores.

Although KV-stores have become popular in recent years, the studies of real-world workload characterization of KV-stores are limited. Atikoglu *et al.* analyzed the KV workloads of the large-scale Memcached KV-store at Facebook [5]. They found that reads dominate the requests, and the cache hit rate is closely related to the cache pool size. Some of their findings, such as the diurnal patterns, are consistent with what we present in Section 4. Major workload characteristics of RocksDB are very different from what Atikoglu *et al.* found in Memcached. Other KV-store studies, such as SILT [31], Dynamo [14], FlashStore [12], and SkimpyStash [13], evaluate

designs and implementations with some real-world workloads. However, only some simple statistics of the workloads are mentioned. The detailed workload characteristics, modeling, and synthetic workload generation are missing.

Modeling the workloads and designing benchmarks are also important for KV-store designs and their performance improvements. Several benchmarks designed for big data NoSQL systems, such as YCSB [11], LinkBench [4], and BigDataBench [45], are also widely used to evaluate KV-store performance. Compared with these benchmarks, we further provide the tracing, analyzing, and key-range based benchmarking tools for RocksDB. The users and developers of RocksDB can easily develop their own specific benchmarks based on the workloads they collect with better emulation in both the KV-query level and storage level.

9   Conclusion and Future Work

In this paper, we present the study of persistent KV-store workloads at Facebook. We first introduce the tracing, replaying, analyzing, and benchmarking methodologies and tools that can be easily used. The findings of key/value size distribution, access patterns, key-range localities, and workload variations provide insights that can help optimize KV-store performance. By comparing the storage I/Os of RocksDB benchmarked by YCSB and those of trace replay, we find that many more reads and fewer writes are generated by benchmarking with YCSB. To address this issue, we propose a key-range based model to better preserve key-space localities. The new benchmark not only provides a good emulation of workloads at the query level, but also achieves more precise RocksDB storage I/Os than that of YCSB.

We have already open-sourced the tracing, replaying, analyzing, and the new benchmark in the latest RocksDB release (see the Wiki for more details [20]). The new benchmark is part of the benchmarking tool of *db_bench* [18]. We are not releasing the trace at this time. In the future, we will further improve YCSB workload generation with key-range distribution. Also, we will collect, analyze, and model the workloads in other dimensions, such as correlations between queries, the correlation between KV-pair hotness and KV-pair sizes, and the inclusion of additional statistics like query latency and cache status.

Acknowledgments

References

[1] M. Annamalai. Zippydb: a modern, distributed key-value data store. https://www.youtube.com/watch?v=DfiN7pG0D0k, 2015.

[2] M. Arlitt and T. Jin. A workload characterization study of the 1998 world cup web site. *IEEE network*, 14(3):30–37, 2000.

[3] M. F. Arlitt and C. L. Williamson. Web server workload characterization: The search for invariants. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):126–137, 1996.

[4] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1185–1196. ACM, 2013.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[6] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani. Speicher: Securing lsm-based key-value stores using shielded execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 173–190, 2019.

[7] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[9] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz. Realtime data processing at facebook. In *Proceedings of the International Conference on Management of Data*, pages 1087–1098. ACM, 2016.

[10] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[12] B. Debnath, S. Sengupta, and J. Li. Flashstore: high throughput persistent key-value store. In *Proceedings of the VLDB Endowment*, volume 3, pages 1414–1425. VLDB Endowment, 2010.

[13] B. Debnath, S. Sengupta, and J. Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the ACM SIGMOD International Conference on Management of data*, pages 25–36, 2011.

[14] 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 *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[15] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strumm. Optimizing space amplification in RocksDB. In *CIDR*, volume 3, page 3, 2017.

[16] Facebook. Cassandra on rocksdb at instagram. https://developers.facebook.com/videos/f8-2018/cassandra-on-rocksdb-at-instagram. 2018.

[17] Facebook. Merge operator. https://github.com/facebook/rocksdb/wiki/Merge-Operator, 2018.

[18] Facebook. db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2019.

[19] Facebook. Myrocks. http://myrocks.io/, 2019.

[20] Facebook. Rocksdb trace, replay, analyzer, and workload generation. https://github.com/facebook/rocksdb/wiki/RocksDB-Trace%2C-Replay%2C-Analyzer%2C-and-Workload-Generationr, 2019.

[21] Facebook. Rocksdb. https://github.com/facebook/rocksdb/, 2019.

[22] Facebook. Single delete. https://github.com/facebook/rocksdb/wiki/Single-Delete. 2019.

[23] S. Ghemawat and J. Dean. Leveldb. *URL: https://github.com/google/leveldb,% 20http://leveldb. org*, 2011.

[24] A. Gupta. Followfeed: Linkedin's feed made faster and smarter. https://engineering.linkedin.com/blog/2016/03/followfeed--linkedin-s-feed-made-faster-and-smarter, 2016.

[25] J. R. Hosking and J. R. Wallis. Parameter and quantile estimation for the generalized pareto distribution. *Technometrics*, 29(3):339–349, 1987.

[26] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi. Slm-db: single-level key-value store with persistent memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, 2019.

[27] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *IEEE International Symposium on Workload Characterization (IISWC 08)*, pages 119–128. IEEE, 2008.

[28] 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 *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST 15)*, pages 1–14, 2015.

[29] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[30] A. W. Leung, S. Pasupathy, G. R. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 08)*, volume 1, pages 2–5, 2008.

[31] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 11)*, pages 1–13. ACM, 2011.

[32] 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.

[33] MathWorks. Power series models. https://www.mathworks.com/help/curvefit/power.html. 2019.

[34] MathWorks. Power series. https://en.wikipedia.org/wiki/Power_series. 2019.

[35] MathWorks. Sine fitting. https://www.mathworks.com/matlabcentral/fileexchange/66793-sine-fitting. 2019.

[36] Y. Matsunobu. Innodb to myrocks migration in main mysql database at facebook. USENIX Association, May 2017.

[37] S. Nanniyur. Sherpa scales new heights. https://yahooeng.tumblr.com/post/120730204806/sherpa-scales-new-heights, 2015.

[38] 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.

[39] Redis. Redis documentation. https://redis.io/documentation, 2019.

[40] A. Riska and E. Riedel. Disk drive level workload characterization. In *Proceedings of the USENIX Annual Technical Conference (ATC 06)*, pages 97–102, 2006.

[41] D. Roselli and T. E. Anderson. *Characteristics of file system workloads*. University of California, Berkeley, Computer Science Division, 1998.

[42] D. S. Roselli, J. R. Lorch, T. E. Anderson, et al. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 00)*, pages 41–54, 2000.

[43] W. Shi, R. Wright, E. Collins, and V. Karamcheti. Workload characterization of a personalized web site and its implications for dynamic content caching. In *Proceedings of the 7th International Workshop on Web Caching and Content Distribution (WCW 02)*, 2002.

[44] J. Wang. Myrocks: best practice at alibaba. https://www.percona.com/live/17/sessions/myrocks-best-practice-alibaba, 2017.

[45] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499, 2014.

[46] A. Williams, M. Arlitt, C. Williamson, and K. Barker. Web workload characterization: Ten years later. In *Web content delivery*, pages 3–21. Springer, 2005.

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

A Appendix

A.1 Trace Replay

./db_bench –benchmarks=replay –trace_file=./trace_<Trace Name> –num_column_families=1 -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -trace_replay_fast_forward=24 –perf_level=2 –trace_replay_threads=3 -use_existing_db=true -db=./<Directory of Existing RocksDB Database for Replay>

A.2 Trace Analyzing

./trace_analyzer -analyze_get -analyze_put -analyze_merge -analyze_delete -analyze_single_delete -analyze_iterator -output_access_count_stats -output_dir=./result_<Trace Name> -output_key_stats -output_qps_stats -output_value_distribution -output_key_distribution -output_time_series -print_overall_stats -print_top_k_access=6 -value_interval=1 -output_prefix= <Trace Name>_result -trace_path=./trace_<Trace Name> ./ <Trace Name>_general.txt

A.3 New Benchmarks

Before running the benchmark, user needs to compile RocksDB db_bench and run it via command lines. Note that, if user runs the benchmark following the 24 hours Sine period, it will take about 22-24 hours. In order to speedup the benchmarking, user can increase the sine_d to a larger value such as 45000 to increase the workload intensiveness and also reduce the sine_b accordingly.

Create a database with 50 million random inserted KV-pairs

./db_bench –benchmarks=fillrandom –perf_level=3 -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -cache_size=268435456 -key_size=48 -value_size=43 -num=50000000 -db=./<Directory of Generated Database with 50 million KV-pairs>

All_random

./db_bench –benchmarks="mixgraph" -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -cache_size=268435456 -keyrange_num=1 -value_k=0.2615 -value_sigma=25.45 -iter_k=2.517 -iter_sigma=14.236 -mix_get_ratio=0.83 -mix_put_ratio=0.14 -mix_seek_ratio=0.03 -sine_mix_rate_interval_milliseconds=5000 -sine_a=1000 -sine_b=0.000073 -sine_d=4500 –perf_level=2 -reads=420000000 -num=50000000 -key_size=48 -

db=./<Directory of Generated Database with 50 million KV-pairs> -use_existing_db=true

All_dist

./db_bench –benchmarks="mixgraph" -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -cache_size=268435456 -key_dist_a=0.002312 -key_dist_b=0.3467 -keyrange_num=1 -value_k=0.2615 -value_sigma=25.45 -iter_k=2.517 -iter_sigma=14.236 -mix_get_ratio=0.83 -mix_put_ratio=0.14 -mix_seek_ratio=0.03 -sine_mix_rate_interval_milliseconds=5000 -sine_a=1000 -sine_b=0.000073 -sine_d=4500 –perf_level=2 -reads=420000000 -num=50000000 -key_size=48 -db=./< Directory of Generated Database with 50 million KV-pairs> -use_existing_db=true

Prefix_random

./db_bench –benchmarks="mixgraph" -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -cache_size=268435456 -keyrange_dist_a=14.18 -keyrange_dist_b=-2.917 -keyrange_dist_c=0.0164 -keyrange_dist_d=-0.08082 -keyrange_num=30 -value_k=0.2615 -value_sigma=25.45 -iter_k=2.517 -iter_sigma=14.236 -mix_get_ratio=0.83 -mix_put_ratio=0.14 -mix_seek_ratio=0.03 -sine_mix_rate_interval_milliseconds=5000 -sine_a=1000 -sine_b=0.000073 -sine_d=4500 –perf_level=2 -reads=420000000 -num=50000000 -key_size=48 -db=./< Directory of Generated Database with 50 million KV-pairs> -use_existing_db=true

Prefix_dist

./db_bench –benchmarks="mixgraph" -use_direct_io_for_flush_and_compaction=true -use_direct_reads=true -cache_size=268435456 -key_dist_a=0.002312 -key_dist_b=0.3467 -keyrange_dist_a=14.18 -keyrange_dist_b=-2.917 -keyrange_dist_c=0.0164 -keyrange_dist_d=-0.08082 -keyrange_num=30 -value_k=0.2615 -value_sigma=25.45 -iter_k=2.517 -iter_sigma=14.236 -mix_get_ratio=0.83 -mix_put_ratio=0.14 -mix_seek_ratio=0.03 -sine_mix_rate_interval_milliseconds=5000 -sine_a=1000 -sine_b=0.000073 -sine_d=4500 –perf_level=2 -reads=420000000 -num=50000000 -key_size=48 -db=./< Directory of Generated Database with 50 million KV-pairs> -use_existing_db=true

FPGA-Accelerated Compactions for LSM-based Key-Value Store

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[†]

[*]Alibaba Group

[†]Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang University
{jason.zt,beilou.wjy,xuntao.cxt,haoke.xh,qushan, tieying.zhang,
dengcheng.hedc,lifeifei,mingsong.cw}@alibaba-inc.com
{yunanlong,hzd, sunjl}@zju.edu.cn

Abstract

Log-Structured Merge Tree (LSM-tree) key-value (KV) stores
have been widely deployed in the industry due to its high
write efficiency and low costs as a tiered storage. To main-
tain such advantages, LSM-tree relies on a background com-
paction operation to merge data records or collect garbages
for housekeeping purposes. In this work, we identify that
slow compactions jeopardize the system performance due
to unchecked oversized levels in the LSM-tree, and resource
contentions for the CPU and the I/O. We further find that the
rising I/O capabilities of the latest disk storage have pushed
compactions to be bounded by CPUs when merging short
KVs. This causes both query/transaction processing and back-
ground compactions to compete for the bottlenecked CPU
resources extensively in an LSM-tree KV store.

In this paper, we propose to offload compactions to FPGAs
aiming at accelerating compactions and reducing the CPU bot-
tleneck for storing short KVs. Evaluations have shown that the
proposed FPGA-offloading approach accelerates compactions
by 2 to 5 times, improves the system throughput by up to 23%,
and increases the energy efficiency (number of transactions
per watt) by up to 31.7%, compared with the fine-tuned CPU-
only baseline. Without loss of generality, we implement our
proposal in X-Engine, a latest LSM-tree storage engine.

1 Introduction

Key-value (KV) stores developed based on the Log-Structured
Merge Tree (LSM-tree) have emerged as the backbone
database storage system serving many applications in the
cloud that are sensitive to both the cost and the perfor-
mance, such as instant messaging, online retail and advertise-
ments. Notable examples include LevelDB [8], BigTable [2],
RocksDB [13], Dynamo [9], WiredTiger [35], and X-Engine
[16] from various companies. LSM-tree is favoured in these
cases because its advantages on the write efficiency and stor-
age costs compensate the shortcomings of the widely adopted
Solid State Drives (SSDs) in industrial storages [13]. For ex-
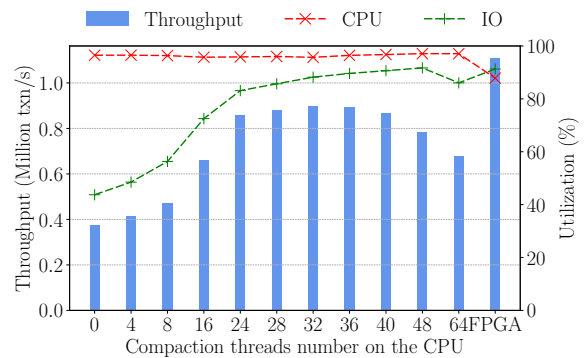ample in the online retail context, the underlying database

Figure 1: Impacts of added CPU threads for compactions on
the overall system throughput.

storage is prefered to be able to provide a high write through-
put for the placement of new orders, an acceptable read latency
for users to query hot items, and a low storage cost for storing
all records of the stockpiles, logistics, payments and other
business-critical e-commerce details on top of a web-scale
storage built upon SSDs [16].

KV stores built atop the LSM-tree usually exploit other
database structures to offer a general-purpose storage ser-
vice that excels in read, write performance and the cost at
the same time. For example, indexes and caches contribute
significantly to answering point lookups, which are among
the majorities in the common write and point read-intensive
(WPI) workloads [7]. Indexes offer a fast path to navigate in
the huge storage, the size of which is often amplified by Multi-
version Concurrency Control (MVCC) [16]. And, caches help
reducing disk I/Os by buffering hot items in the main mem-
ory [16, 25].

Despite these efforts made, we have observed that the per-
formance of LSM-tree KV stores often fatigues after serving
the WPI workloads for long hours due to poorly maintained
shapes (i.e., oversized levels) of the LSM-tree under work-
load pressures. An LSM-tree keeps multiple levels of records
with inclusive key ranges in the disk (details introduced in

Section 2.1). This data organization often forces a query to traverse in multiple levels to merge scattered records for a complete answer or seek a record, even with indexes. Such operations carry the extra overhead of skipping over invalid records marked for deletion. To contain these drawbacks, background compaction operations (a.k.a., merge) are introduced to merge inclusive data blocks between adjacent levels and remove deleted records, intending to keep the LSM-tree in a properly tiered shape. However, in this work, we find that there exists a difficult trade-off between resources devoted to query and transaction processing in the critical path and resources devoted to background compactions which consume heavy computation and disk I/O resources, especially for WPI workloads containing both reads and writes. If we allocate more software threads to compactions, we throttle up the background maintenance of the storage at the risk of hurting the overall performance by depriving CPUs of actually processing queries and transactions.

Figure 1 plots the throughputs of an LSM-tree KV store processing a typical WPI workload (75% point lookups, 25% writes) using an increasing number of CPU threads for compactions. With the number of threads scaling up to 32, the total compaction throughput increases, resulting in significant performance benefits. However, with more threads added for compactions, CPUs become saturated and contended. Consequently, the system throughput drops after 32 threads as shown in Figure 1. We also find out with detailed profilings (introduced in Section 4) that compactions are still not fast enough to deal with the above-introduced problems with 32 or more threads.

Research efforts exploring two major approaches have been made to optimize compactions [6, 16, 18, 28, 29, 40]. One is to reduce the workload per compaction task by exploiting data distributions (e.g., almost sorted, not overlapped ranges) to avoid unnecessary merges [16, 29], or by splitting data into multiple partitions and schedule compactions for each partition when needed separately [18]. The other is to optimize the scheduling of compactions in terms of when and where compaction shall be executed [28]. Ideally, compaction should be completed when it is most needed for the sake of performance, and its execution has minimal resource contentions with other operations in the system. However, these two conditions often contradict with each other in WPI workloads, because the needs for compactions and the needs for high KV throughput often peak at the same time. Thus, the resource contentions in the storage system for both CPUs and I/O remain as a challenge, limiting the scalability of compaction speed with added CPU threads, and leaving the performance fatigue of LSM-tree storage system as an open problem.

In this paper, we propose to offload compactions from CPUs to FPGAs for accelerated executions due to four considerations. Firstly, offloading compactions away from CPUs relieves CPUs from such I/O-intensive operations. This enables the storage system to use less CPUs or increase the

throughput using the same number of CPUs, both of which translate to monetary savings for users, especially in the public cloud. Secondly, FPGAs suit the requirement to accelerate compactions which are pipelines of computational tasks, compared with GPUs which are prefered by computations of the SIMD paradigm. We have also found out in this work that compactions merging short KV records (pairs) are surprisingly bounded by computation, partially because the disk I/O bandwidth has been improved significantly recently. This calls for dedicated performance optimization. Thirdly, the high power efficiency of FPGAs offers a competitive advantage on reducing the total cost of ownership (TCO), compared with other solutions. And, users of LSM-tree KV stores are sensitive to such costs, given that they pay for the storage almost permanently. We argue in this work that offloading compactions to FPGAs increase the economical value of LSM-tree KV stores in the cloud.

On the FPGA, we design and implement compaction as a pipeline including three stages on decoding/encoding inputs/outputs, merging data, and managing intermediate data in buffers. We also implement an FPGA driver, and an asynchronous compaction task scheduler to facilitate the offloading and improve its efficiency. We integrate our FPGA offloading solution with X-Engine, a state-of-the-art LSM-tree storage system [16]. We evaluate X-Engine with and without our proposal using typical WPI workloads with varying compositions. Experimental results show that the proposed FPGA offloading approach accelerates compactions by 2 to 5 times, improves the overall throughput of the storage engine by 23%, compared with the best CPU baseline. Overall, we make the following contributions:

- We have identified the slow and heavy compaction as a major bottleneck causing resource contentions, causing oversized level L_0 and other problems that eventually lead to the performance degradation of the LSM-tree KV store.

- We have designed and implemented an efficient multi-staged compaction pipeline on FPGAs (i.e., Compaction Unit), supporting merge and delete operations. We have introduced an asynchronous scheduler (i.e., Driver) to coordinate CPUs and the FPGA for the offloading of compactions. To the best of our knowledge, this is the first work on offloading compactions to FPGAs.

- We have modeled the FPGA compaction throughput analytically with validations. Using this model, we are able to predict the performance of the proposal for different offloaded compaction tasks. And, we have identified future optimization opportunities on the FPGA for compactions.

- We have compared the performance and energy consumptions of a state-of-the-art LSM-tree KV store (X-Engine [16]) using both the CPU-only baseline and the FPGA-offloading solution for compactions using both micro- and

macro-benchmarks. Evaluation results show that our proposal increases the overall throughput of the KV store by 23% and increases the energy efficiency (number of transactions per watt) by 31.7%, compared with the CPU-only baseline.

This paper is organized as follows. Section 2 introduces backgrounds of LSM-tree KV stores, FPGAs and our motivations to offload compactions to FPGAs. Section 3 explains the overview of our design and introduces design details of compactions on the FPGA and the offloading process, including both implementation details and analytical models. We evaluate our proposals in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 Background and Motivation

2.1 LSM-tree KV-Store

LSM-trees have been extensively studied in academia and deployed in the industry due to its high write efficiency and low storage cost on SSDs. In the original design [23] shown in Figure 2a, an LSM-tree contains two tree-like components C_0 and C_1, residing in the main memory and the disk, respectively. For fast writes, incoming KV records are inserted into C_0, accessing only the main memory. When C_0 is full in the main memory, parts of it are merged with C_1 in the disk, leaving space in the main memory for new data. The overhead of this merge operation increases as the size of C_1 grows, because leaf nodes of C_0 may overlap with many leaf nodes of C_1. To bound such overhead, it is preferable to divide a single disk component into multiple ones: C_1, C_2, ..., C_k, where each component C_{i+1} is larger than its previous one C_i. However, a single KV record now has to be merged multiple times among these components over time, causing write amplification. And, a lookup may also have to access multiple components with inclusive key ranges, which are referred to as read amplifications in this work.
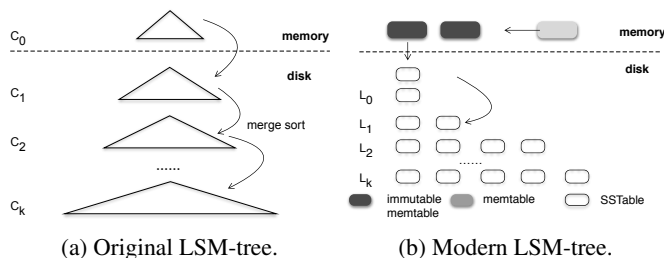

(a) Original LSM-tree.   (b) Modern LSM-tree.

Figure 2: Original LSM-tree and Modern LSM-tree Architectures.

To bound write and read amplifications, many studies have envolved LSM-tree into a tiered storage with optimized main memory data structure, multi-level disk components with each level consisting of multiple files or fine-grained data blocks.

Figure 2b shows a typical LSM-tree architecture as in many industrial systems such as RocksDB [13] and X-Engine [16]. Incoming data are inserted into memtables (often implemented as skiplists [24]). Once filled, memtables are switched to be immutable and flushed to the first level L_0 in the disk. A level L_k is similar to component C_k in the original design with a major difference that L_k is partitioned into many files (e.g., Sorted Sequence Tables as in RocksDB [13]) or data blocks (e.g., extents as in X-Engine [16]). There are two types of policies for the merging operation (i.e., compaction). For each batch to be merged into the next level (or component), one policy is to merge it with existing data in the target level, know as the *leveling* policy. This approach keeps data in a level in a nicely sorted order at the expense of the compaction speed. The other approach is to simply append data into the next level without merging, known as the *tiering* policy. In this way, the compaction itself is fast at the expense of the sorted order within a level. Dayan et al. have compared these two policies analytically [5]. Huang et al. proposed a data reuse technique allowing compaction to only physically merge data blocks with overlapping key ranges and reuse the rest to reduce the total I/O accesses [16]. Caches, indexes, and bloom filters have been introduced to compensate the shortcoming of the log-structured storage on lookup performance [7, 13, 16]. With skewed data accesses, these optimizations can reduce disk I/Os significantly [16].

2.2 Motivations

Despite the state-of-the-art optimizations introduced above, we find that slow compactions still lead to the performance fatigue problem of an LSM-tree KV store running the WPI workloads for the following reasons:

Problem 1: Shattered L_0. In the first level L_0, data blocks often have overlapping key ranges because they are directly flushed from the main memory without being merged. Unless compactions merge them in time, a point lookup may have to check multiple blocks for a single key, even with indexes. As time passes in such cases with slow compactions, data blocks in L_0 stockpiles, continuously increasing the lookup overhead. Such shattered L_0 have significant performance impacts because records flushed into L_0 are still very hot (i.e., very likely to be accessed) due to the data locality.

Problem 2: Shifting Bottlenecks. A compaction operations naturally consist of multiple stages: decoding, merging and encoding because KV records are often prefix encoded. To identify the bottleneck among these stages, we profile a single compaction task performed by a single CPU thread on an SSD. Figure 3 shows the execution time breakdown. As the value size increases, the percentage of computation time (*decoding*, *merging*, *encoding*) decreases. For short KVs, computation takes up to 60% time of the whole compaction procedure. When the value size exceeds 128 bytes, I/O operations occupy most of the CPU time. This breakdown shows

that the bottleneck transfers from the CPU to the I/O with increasing KV sizes. This phenomenon suggests that the compaction is bounded by computation while merging short KVs and bounded by I/O in other cases.
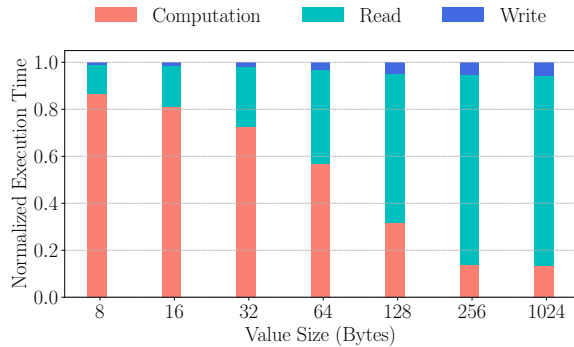

Figure 3: The breakdown of CPU compaction (key=8 bytes).

In this work, we map the above introduced three stages to a pipeline and offload it to accelerators. With faster compactions, data blocks in L_0 are more frequently merged. With offloading, CPUs are relieved from the heavy compaction operations, leaving more resources to process queries and transactions. We choose to offload compactions to FPGAs due to its suitability to accelerate pipelines of computational tasks like compactions, low power consumption for low TCO, and its flexibility as pluggable PCIe-attached accelerators.

2.3 FPGA offloading

A Filed-programmable Gate Array (FPGA) chip is an array of programmable hardware units, including look-up tables, flip-flops, and DSPs (digital signal processors). Software developers enjoy the flexibility in configuring FPGAs for dedicated purposes without other software overheads (e.g., the OS). FPGAs have been widely applied to accelerate pipelines of computational tasks for high performance and low power consumptions, compared with other accelerators such as GPUs, and many-core CPU processors [30, 33, 41, 42]. FPGAs suit such applications because programmed hardware units can be easily wired to implement pipelines with multiple stages. And, each hardware unit executes fixed operations, specified by designs, at every cycle with no extra overhead, achieving a high level of efficiency. In the market, FPGA chips are usually embedded in an SoC (system-on-chip) together with other hardware (e.g, RAM, CPUs, PCIe interfaces, SSD controllers) to meet different system requirements. It is also available in the public cloud (e.g., Amazon EC2 F1 instances, Alibaba Cloud F3 instances) to offer customizable computation services.

There are two main approaches to integrate FPGAs in a KV store. One is the "bump-in-the-wire" design that places the FPGA between the CPU and the disk [37]. This approach, in which the FPGA serves as a data filter, is favoured when the on-chip RAM of FPGAs are small in size so that it can only temporarily hold a slice of a data stream. As the on-chip RAMs grow in capacity, FPGAs are now capable to serve as a co-processor, onto which we can offload operations with large data volume to process. This approach suits asynchronous tasks, during which CPUs are not stalled by the offloaded tasks. In this paper, we adopt the latter approach to offload compactions as a whole to FPGAs and only use the CPU for task generations. Such solutions based on FPGA-offloading can naturally evolve into a computational storage with coupled FPGA-SSD devices emerging in the market (e.g., Samsung SmartSSD).

3 Design and Implementation

3.1 Overview

Figure 4 shows our design of the FPGA-offloading for compactions, integrated with X-Engine. X-Engine is one of the state-of-the-art LSM-tree KV store [16]. It exploits memtables to buffer newly inserted KV records, and caches to buffer hot pairs and data blocks. In the disks, each level of the LSM-tree contains multiple extents. Each extent in turn stores KV records with associated indexes and filters. X-Engine maintains a global index to accelerate lookups. In the baseline system without offloading compactions, all put, get, delete, multi-get operations in addition to background flush and compaction operations are processed by CPUs. We refer to this as the CPU-only baseline.

To offload compactions, we first design a **Task Queue** to buffer newly triggered compaction tasks, and a **Result Queue** to buffer compacted KV records in the main memory. Software-wise, we introduce a **Driver** to offload compactions, including managing the data transfer from the host and the FPGA (the device). On the FPGA, we design and deploy multiple **Compaction Units (CUs)**, responsible for merging KV records.

3.2 Driver

3.2.1 Managing Compaction Tasks

In LSM-tree KV stores such as X-Engine, compaction between level L_i and L_{i+1} are triggered when the data size of L_i (i.e., the number of extents as in X-Engine) reaches a pre-defined threshold. To facilitate offloading, we maintain three types of threads: builder threads, dispatcher threads and driver threads at the CPU side to build compaction tasks, dispatch tasks to CUs in the FPGA, and install compacted data blocks back into the storage, respectively. We illustrate this process in Figure 5 and introduce the details of these threads in the following.
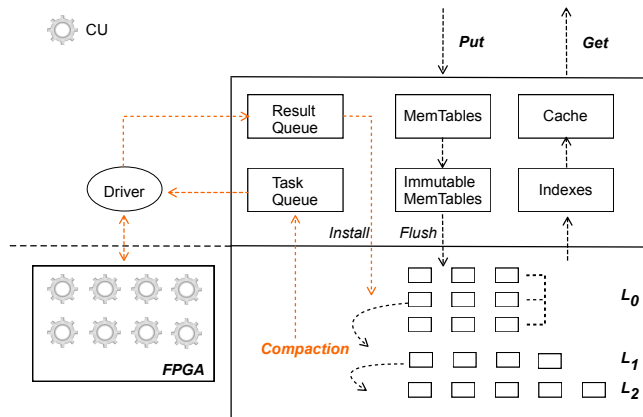
Figure 4: System Overview of the Storage Engine with the FPGA-offloading of Compactions.
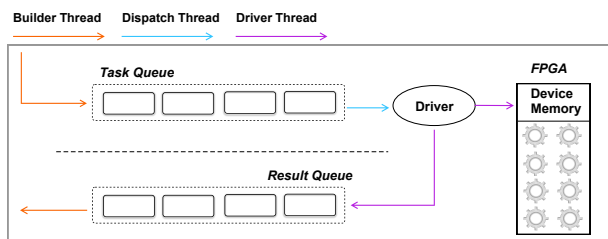

Figure 5: Asynchronous Compaction Scheduler. By implementing a thread pool to distribute FPGA-offloading compaction tasks, the cost of context switch is reduced.

Builder thread For each triggered compaction, the builder thread partitions extents to be merged into multiple groups of similar sizes. Each group then forms a compaction task, with its data loaded into the main memory. An FPGA compaction task is built containing the metadata required including pointers to the task queue, input data, results, a callback function (transferring compacted data blocks from the FPGA to the main memory), return code (indicating whether a task is finished successfully) and other metadata of the compaction task, as shown in Figure 6. This compaction task is pushed to the task queue, waiting to be dispatched to a CU on the FPGA. This builder thread also checks the result queue and installs compacted data blocks back to the storage when the task is successful. In cases when the FPGA failed to complete
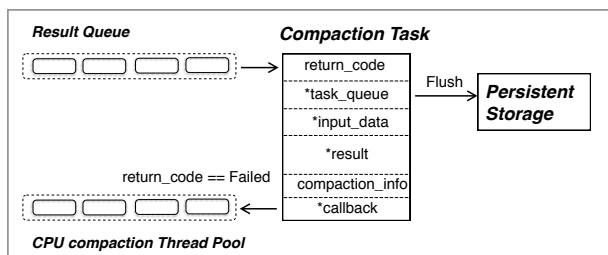

Figure 6: CPU checks the task's status, redoes the task and flushes the data as SSTable if necessary.

a compaction task (e.g., the key-value size exceeds FPGA's capacity), it starts a CPU compaction thread to retry this task. In our practices, we find that only 0.03% tasks offloaded to the FPGA fail on average.

Dispatcher thread The dispatcher consumes the task queue and dispatches tasks to all the CUs on the FPGA in a round-robin manner. Given that compaction tasks have similar sizes, such round-robin dispatching achieves balanced workload distribution among multiple CUs on the FPGA. The dispatcher notifies the driver thread to transfer the data to the device memory on the FPGA.

Driver thread The driver thread transfers input data associated with a compaction task to the device memory on the FPGA and notifies the corresponding CU to start working. When a compaction task is finished, this driver thread is interrupted to execute the callback function, which transfers the compacted data blocks back to the host memory, and pushes the completed task into the result queue.

In such implementation, we tune the size of a compaction task to provide sufficient data for a compaction unit, guarantee load balances among CUs, and limit the overhead for retrying a compaction task upon failures on the FPGA. We also tune the number of threads for the builder, dispatcher, and the installer separately, to achieve balanced throughputs among them.

3.2.2 Instruction and Data Paths

To drive the FPGA for compactions, we need to transfer instructions and data via the PCIe interconnect. To maximize such transfer efficiency, we design an *Instruction Path* and a *Compaction Data Path*. We also design an *Interruption Mechanism* to notify the *Installer Thread* when a CU completes a compaction task, and a *Memory Management Unit (MMU)* managing the device memory on the FPGA. The whole process is depicted in Figure 7.

The detailed description of the submodules are as follows:

Instruction Path The instruction path is designed for small and frequent data transfers like the CU availability check.

Compaction Data Path The data path uses DMA. The data for compaction is transferred via this path. By this way, the CPU is not involved in the data transfer procedure.

Interruption Mechanism When a compaction task finishes, an interrupt will be sent via PCIe, and then the result is written back to the host with the help of the interrupt vector.

Memory Management Unit (MMU) MMU allocates memory on the device memory to store the input data copied from the host.

3.3 Compaction Unit

Compaction Unit (CU) is the logic implementation of the compaction operation on the FPGA. Multiple CUs can be deployed on the same FPGA, the total number of which mainly
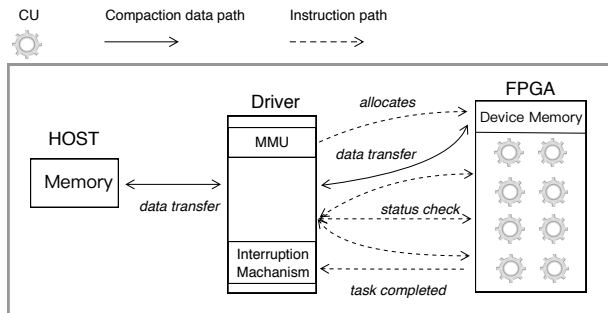
Figure 7: FPGA Driver.

depends on the available hardware resources. Figure 8 illustrates the design of a CU. In this design, a compaction task consists of multiple stages: Decoder, Merger, KV Transfer, and Encoder (details introduced below). We introduce buffers (i.e., KV Ring Buffer, and Key Buffer) between subsequent modules and a Controller to coordinate the execution of each module.
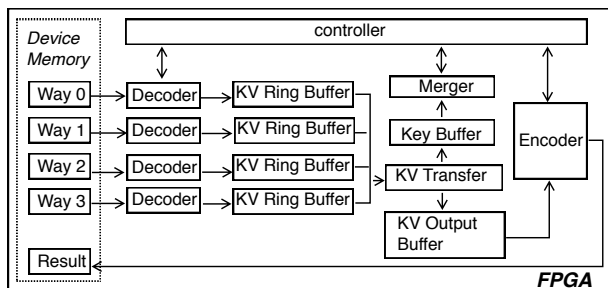

Figure 8: Compaction Unit overview.

Decoder We consider a KV store where keys are prefix-encoded to save spaces. The *Decoder* decodes input KV records. If the tiering compaction policy is adopted, there may exist multiple ways of input data blocks to be merged. For the levelling policy, there are at most two ways of inputs from the two adjacent levels. We find that most tiering compactions involve two to four ways of inputs in our practices. If we place two decoders in a CU with each one decoding one way of inputs, we need to build three two-way compactions to achieve a single four-way compaction. Instead, placing four decoders consume 40% more hardware resources without requiring more tasks for two to four ways of compactions, compared with the two-decoder design. Thus, we chose to place four decoders in each CU. The decoder outputs decoded KV records into the *KV Ring Buffers*.

KV Ring Buffer The *KV Ring Buffer* caches the decoded KV records. We have observed in our practices that KV sizes rarely exceed 6KB. Thus, we configure each KV ring buffer to contain 32 8KB slots. The additional 2KB can be used to store meta-information such as the KV length. We design three status signals for a KV ring buffer: $FLAG\_EMPTY$, $FLAG\_HALF\_FULL$ and $FLAG\_FULL$, indicating whether

the buffer is empty, half full, or fully filled, respectively. Beginning with an empty buffer, the decoder keeps decoding and filling this buffer. After this buffer is half-filled, the downstream *Merger* is allowed to read the filled data and starts merging them. While the merger is working, the decoder continues to fill the rest of the buffer until it is fully filled. We match the speed of the merger and the decoder so that the merge takes the same amount of time to finish its job by filling half of the ring buffer. In this way, the decoder and the merger can be efficiently pipelined. In cases that these two modules do not match, the *Controller* pauses the decoder. We keep a read pointer for the merger to mark from where it should read in KV records, and a write pointer for the decoder similarly.

KV Transfer, Key Buffer and Merger Only keys are transferred to the *Key Buffer*, and then the *Merger* for comparisons. If a key is qualified as an output, the *KV transfer* module transfers the corresponding KV record from the upstream KV ring buffer to the *KV Output Buffer* with the same data structure as the KV ring buffer. The *Controller* gets notified with the comparison result and move the read pointer in the corresponding KV Ring Buffer forward accordingly. For example, as illustrated in Figure 9, if the KV record in *way 2* is the smallest key-value in the current comparison round, the Controller will notify KV Transfer to transfer the KV record where the read pointer points in *way 2*'s KV Ring Buffer and move the read pointer forward to fetch the next entry into the key buffer for the next round.

Encoder This module encodes the merged KV records from the KV output buffer to data blocks and places them into the device memory. Because there is only one way of merged data, we place only one encoder within each CU.

Controller The *Controller* module serves as a coordinator in a CU. It manages the read and write pointers in the KV ring buffer for the merger and decoder, respectively. It also signals each module when to start or pause.
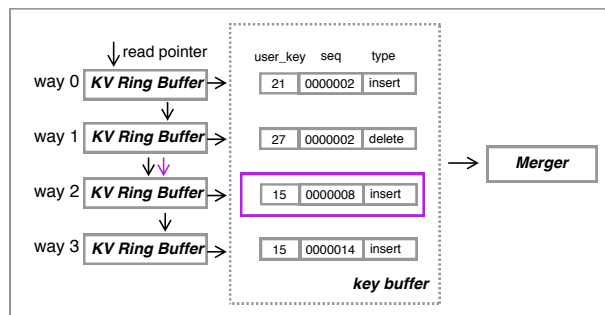

Figure 9: Detailed Design of the Merger.

3.4 Analytical Model for CU

Due to the pipeline design, it is crucial to match the throughput of different modules to avoid over-provisioning and waste of hardware resources. However, it is difficult to derive the

number of resources for each component with so many tuning knobs. Therefore, in this section we propose an analytical model for CU to guide the resource provisioning. The notations and parameters that we use in the model are summarized in Table 1.

Table 1: Summary of Notations used in the Analytical Model.

Parameter	Values/Units	Description
N	Workload-dependent	Total number KVs processed
f_{FPGA}	200 MHz	Clock frequency of the FPGA
W_{bus}	8 Bytes	Width of bus data
W_{key}	$1 \sim 2K$ Bytes	Width of key
W_{value}	$0 \sim 4K$ Bytes	Width of value
$A_{decoder}$	2	Decoder amplification factor[1]
$A_{kv\_transfer}$	1	KV_transfer amplification factor
A_{merger}	5	CPE amplification factor
$A_{encoder}$	2	Encoder amplification factor
$B_{transfer}$	Bytes/second	Host-device transfer bandwidth
$b_{decoder}$	10	Base cycles for Decoder
$b_{kv\_transfer}$	18	Base cycles for KV_transfer
b_{merger}	50	Base cycles for Compaction PE
$b_{encoder}$	46	Base cycles for Encoder
μ	Workload-dependent	Merging selectivity

The throughput of a CU is dominated by the throughput of the slowest stage as shown in Equation 1, with only one exception on the KV Transfer. When the KV Transfer does work to transfer merged KVs, it is executed in serial with its predecessor stage, Merger. In this case, their costs shall be combined. The cost of each stage consists of its computation and memory overhead (i.e., moving data into/from Block RAMs inside the FPGA chip) in addition to a constant number of base cycles consumed when starting this stage. We introduce the details in the following.

$$T_{CU} = min\{T_{decoder}, T_{kv\_transfer}, T_{cpe}, T_{encoder}, T_{mem}\} \quad (1)$$

$$T_{decoder} = \frac{f_{FPGA}}{b_{decoder} + (A_{decoder} \cdot W_{key} + W_{value})/W_{bus}} \quad (2)$$

$$T_{merger} = \frac{f_{FPGA}}{b_{merger} + A_{merger} \cdot W_{key}/W_{bus}} \quad (3)$$

$T_{kv\_transfer}$
$$= \frac{f_{FPGA}}{b_{kv\_transfer} + (A_{kv\_transfer} \cdot W_{key} + W_{value})/W_{bus}} \quad (4)$$

$$T_{encoder} = \frac{f_{FPGA}}{b_{encoder} + (A_{encoder} \cdot W_{key} + W_{value})/W_{bus}} \quad (5)$$

$$T_{mem} = \frac{N \cdot (W_{key} + W_{value}) \cdot (1 + \mu)}{B_{transfer}} \quad (6)$$

Equation 2, 4, 3, and 5 model the cost of processing a single KV record for the *Decoder*, *Merger*, *KV Transfer* and *Encoder* stages, respectively. Equation 6 models the throughput of data transfer between the host and the device, bounded by PCIe bandwidth. Through inspecting the hardware implementation and performance profiling, we initiate these models with data listed in Table 1. As an example, for a KV record with 8-byte key and 32-byte value, the costs for each stage in the term of cycles consumed per KV are 16, 55, 23, and 52 for the Decoder, Merger, KV Transfer, and Encoder stages, respectively. The overall throughput is $\frac{200MHz}{55} = 3.6 \, M \, records/s$, assuming the performance is bounded by the Merger. We valid this model with experiments in Section 4.2.

4 EVALUATION

In this section, we first evaluate how a Compaction Unit (CU) performs on the FPGA using microbenchmarks, in addition to validating our analytical model for a CU. We then evaluate how the proposed FPGA-offloading of compactions contribute to X-Engine using varying workloads. X-Engine is one of the state-of-the-art LSM-tree KV stores [16].

4.1 Experimental Setup

We evaluate our proposal on a server featuring two Intel Xeon Platinum 8163 2.5 GHz 24-core CPUs with two-way hyper-threading, a 768 GB Samsung DDR4-2666 main memory and a RAID 0 consisting of 10 Samsung SSDs, running Linux 4.9.79. We attach a Xilinx Virtex UltraScale+ VU9P FPGA board (running at 200MHz) with a 16 GB device memory to this server through a x16 PCIe Gen 3 interface, to which we offload compactions. Specifications of this FPGA board is summarized in its datasheet [39].

4.2 Evaluating the FPGA-based Compaction

CU Performance Analysis. To evaluate the performance of a CU, we prepare 4,000,000 KV records with varying key and value sizes and manually trigger a compaction to merge them. Figure 10 compares the throughput, in the term of input KV records merged per second, achieved by a single CPU thread and the FPGA-offloading of compactions.

We observe that the throughputs of the FPGA-offloading compactions are 203% to 507% higher than those on the CPU, achieving performance speedups across all KV sizes. For shorter KV sizes (thus smaller compaction tasks), the FPGA-offloading solution suffers from a relatively higher percentage of the offloading overhead. Such costs are diluted when the KV size increases, resulting in higher speedups over the CPU.

---

[1]The amplification factor indicates the times of the number of clock cycles for each module if we use kv_transfer as the baseline.

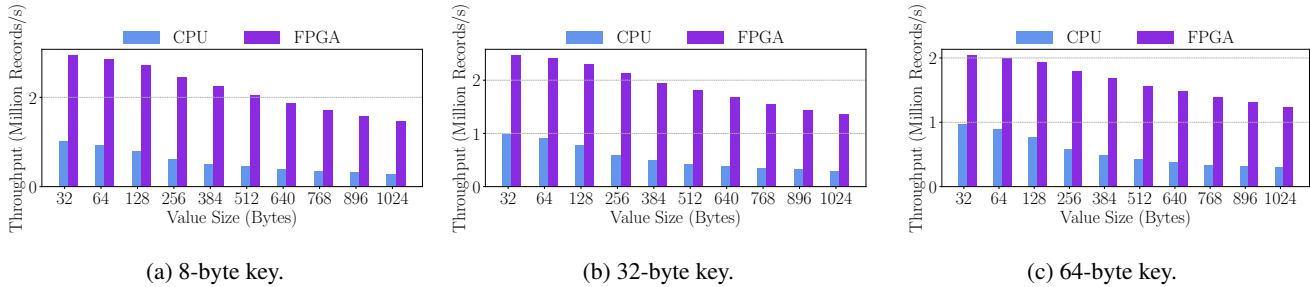| (a) 8-byte key. | (b) 32-byte key. | (c) 64-byte key. |

Figure 10: Throughput of FPGA-based and CPU-based compactions with varying KV settings.

We validate our analytical model for a CU on the FPGA using a high-pressure scenario in which all input KV records are decoded, compared, encoded, and written back to the storage (i.e., 100% selectivity). In this case, in our current implementation, the Merger and the KV Transfer stages are not pipelined. They execute in serial. Figure 11 reports the measured and estimation throughput. For KV records up to 8-byte key and 256-byte values, estimation errors are within only 5%. With larger sizes, the error grows up to 13% for the 1024-byte value. Such estimation inaccuracies are very likely caused by potential pipeline stalls and bus contentions. And, we conclude that the overall throughput is always bounded by that of the Merger and KV Transfer stages combined, neither the accesses to the device memory nor the data transfer over the PCIe.
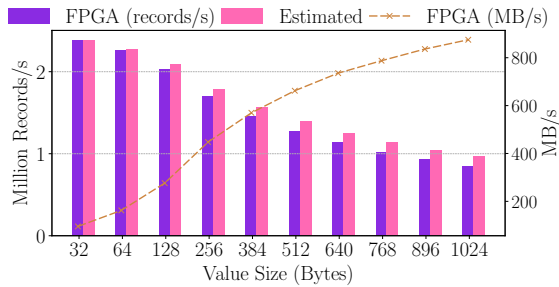

Figure 11: FPGA Compaction Model (key=8 bytes).

Resource Consumption. In Table 2, we report the resource consumption (i.e., LUT, Flip-Flop and memory consumed) of each stage in a CU on the FPGA. A single CU consumes less than 5% of the total resources, with the encoder consuming the largest number of resources because it has the most complicated logics including prefix encoding and communication with other stages. In practice, we can place up to 8 CUs in the FPGA board, utilizing 25.6% of the LUTs, 10.4% of the Flip-Flops, and 38.4% of the device RAM. In addition to the about 15% resources consumed by the shell inside the FPGA chip (including buses, DDR controllers, etc.), our current implementation utilizes about half

of the hardware resources on the FPGA. We recognize that such utilization is constrained by the necessary place and route overhead on the FPGA and our current implementation which can be improved with further optimizations.

Table 2: Hardware resource utilization by a single CU.

	LUT	Flip-Flop	RAM (MB)
Decoder	5783×4	4570×4	8×4
KV Transfer	1076	1006	0
Merger	4119	2555	0
Encoder	6489	4101	0
Others	3819	4841	14
1 CU	38635	30783	46
FPGA total	1182000	2364000	960
Utilization	**3.2%**	**1.3%**	**4.8%**

4.3 Evaluating a KV Store with FPGA-offloading of Compactions

4.3.1 Workloads

To evaluate the impacts of FPGA-offloading of compactions on an LSM-tree KV store using X-Engine [16], we adopt DBBench, a popular opensource benchmark to measure LSM-tree storages such as LevelDB and RocksDB [8, 25, 26]. We consider skewed workloads following a zipf distribution, and use the zipf factor of 1.0 to generate the default workload. We vary multiple parameters in our experiments including the number of CPU threads used for compactions, the read ratio (i.e., the percentage of reads (v.s. writes) in the workload) and others.

We prepare KV records for 32 LSM-tree tables, with each table storing 200,000,000 records in a single X-Engine KV store instance. The total data size is about 278 GB. We tune the cache size in the main memory to 70 GB. By default, we use 8-byte keys and 32-byte values, a common size as in many KV stores deployed in the industry, especially those storing secondary indexes. In all experiments, we warm up the storage for 3,600 seconds and measure performance metrics for another 3,600 seconds, which is long enough for the exposure of performance issues in our setting.

4.3.2 The Impacts of Compactions

Figure 12 shows the overall performance (transactions per second) achieved in correspondence with the size of level L_0, comparing the impacts of increasing numbers of threads for compactions on the CPU-only baseline and the FPGA-offloading solution, given our default WPI workload. By up to 24 CPU threads, added threads yield significant performance improvements by merging overlapping key ranges in the level L_0 at an increasing speed, which reduces the expected number of I/O accesses for a point lookup. In this phase, compactions are slower than the total write throughput of the KV store, so that it is not able to ingest enough newly written data in the LSM-tree. Thus, accelerating compactions using more threads pays back.

From 24 to 32 threads, the throughput barely changes. After 32 threads, the throughput drops with more threads added. The reason is two-fold. Firstly, more threads added introduce more resource contentions, as shown in Figure 1 where the overall CPU utilization almost always 100%. And, we show in Figure 3 that compactions, in this case, are bounded by computations on the CPU. Secondly, added threads eventually saturate the disk I/O as shown in Figure 1, incurring I/O contentions for both the query/transaction processing and compactions.

Figure 12 shows that the FPGA-offloading compaction achieves around 23% higher throughput than the best CPU-only baseline in this workload. Both accelerating compactions and reducing resource contentions at the CPU side contribute to such improvements (shown in Figure 1). Due to the fine-grained compaction task build and distribute, we have not observed any obvious difference in memory consumption between FPGA-offloading compaction and CPU-based compaction (shown in Figure 1). The average memory bandwidth with FPGA-offloading compaction (50.06 GB/s) is 29% lower than the best CPU baseline (70.50 GB/s), showing the CPU-only approach consumes more memory bandwidth.
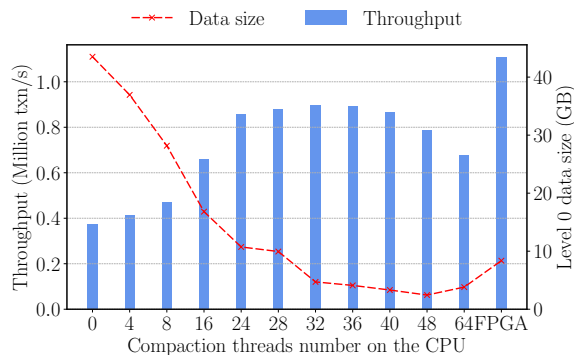

Figure 12: Compaction threads number vs. L_0 data size.

Table 3 summaries the performance metrics and power consumption of X-Engine with FPGA-offloading compaction and CPU-based compaction using 32 threads. With FPGA acceleration, the throughput is improved by 23.3% while the average response time is decreased by 14.7% and 28.1% for read and write operation respectively. The tail request latency is reduced by over 40% as the computation contention is relieved with compaction offloaded to the FPGA. Furthermore, we report the power consumptions. The average machine-wise power consumption is lowered by 7.2% with FPGA-offloading of compactions reducing the CPUs' utilization and power consumptions. Because the FPGA has higher energy efficiency for compactions than CPUs, the overall energy efficiency (number of transactions processed per watt) is improved by 31.7%. With further continuous efforts on optimizing such FPGA-offloading implementations for KV stores, more energy savings and TCO reductions are promising.

4.3.3 The Impacts of Read Ratio

In this section, we investigate the performance benefits of the FPGA-offloading compaction when the read ratio varies. In Figure 13, we gradually reduce the read ratio to 75% which resembles a write-heavy WPI workload, our proposed offloading approach outperforms the baseline with around 10% reduction in CPU consumptions in all cases and consumes slightly more I/Os due to its higher throughput than the CPU.

4.3.4 Macro Benchmarks

Now we compare the CPU-only baseline and our proposal using the DBBench [26] and YCSB benchmarks [4]. Figure 14 reports the DBBench results, including the following tests: **FR** (*fill random*, randomly inserting records), **SRWR** (*seek random while writing*, seeking random individual records with only one thread inserting records), **RWR** (*read while writing*, reading multiple random threads with only one thread inserting records), **UR** (*update random*, updating random records) and **RRWR** (*read random write random*, all threads reading and writing random records for 90% and 10% of the time, respectively).

In these tests, our proposal manages to reduce the CPU utilization and improve the throughput with similar I/O and memory consumptions. FR has the highest performance improvement (54% improvement in total) per CPU utilization reduction (15% utilization reduction in total) because such write-only workloads generate a lof of compactions that benefit from offloading. For the I/O-bounded RWR, our offloading approach improves the throughput by 18% by relieving CPUs from compaction I/Os (20% utilization reduction) without resolving the I/O bottleneck. In other tests, our approach achieves slightly higher I/O utilizations than the CPU-only baselines, resulting in 17% to 29% performance improvements. These results demonstrate multiple benefits of replacing CPUs for the execution of compactions by FPGAs.

Figure 15 reports the YCSB results using the following

Table 3: The value of adding an FPGA to X-Engine.

	Million Txn/s	Avg Get RT (μs)	P99 Get RT (μs)	Avg Put RT (μs)	P99 Put RT (μs)	Power (Watt)	Efficiency (Txn/Watt)
CPU (32 compaction threads)	0.90	139.89	928.15	107.51	864.95	636.54	1416.54
CPU + FPGA	1.11	119.38	537.08	77.30	499.52	590.78	1865.44
Improvement	**+23.3%**	**-14.7%**	**-42.1%**	**-28.1%**	**-42.2%**	**-7.2%**	**+31.7%**

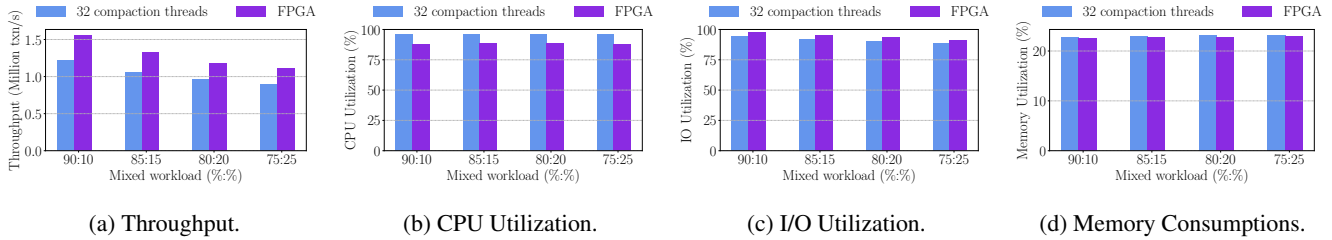| (a) Throughput. | (b) CPU Utilization. | (c) I/O Utilization. | (d) Memory Consumptions. |

Figure 13: Comparing the CPU-only baseline and the FPGA-offloading proposal with varying read ratios (percentages of gets).

workloads: **workload a** (50% read and 50% update), **workload b** (95% read and 5% update), **workload c** (read-only), **workload d** (95% read and 5% insert), **workload e** (95% scan and 5% insert) and **workload f** (50% read-modify-write latest records and 50% random reads). Reads dominate all these workloads, resulting in non-frequent compactions in the underlying KV store. Hence, the throughout improvement is marginal with our proposal. We observe 23.7% and 16.1% gains on throughput for **workload a** and **workload f**, respectively. Both these workloads contain non-trivial writes.

In both the DBBench and the YCSB benchmarks, the proposed FPGA-offloading of compactions contribute more to the KV store when there are significant writes in the workload, compared with read-intensive ones, because compactions in LSM-trees are only triggered upon writes. Acknowledging that both the WPI workload and read-intensive workloads are common in the applications of LSM-tree KV stores, there are efforts in both the literature and the industry that exploit FPGAs to accelerate lookups and queries [31]. Given the flexibility of FPGAs, it is worth exploring of dynamically switching the logics of FPGAs or programming different dedicated accelerator units in a single FPGA to suit various workloads in the production.

5 Related Work

5.1 Software Optimizations of Compactions

To improve the efficiency of compactions, VT-tree [29] uses the stitching technique to avoid unnecessary disk I/Os for sorted and non-overlapping key ranges. However, this method is subject to data distributions and may lead to fragmentations, worsening the performance of range scans and compactions. In bLSM [28] and PE [18], the data distribution is taken into consideration. Both algorithms partition the key range into multiple sub-key ranges and confine compaction in hot data

key ranges, which accelerates the data flow. PCP [40] observed that the compaction procedure can be pipelined. It employs multiple CPUs and storages to fully utilize both CPU and I/O resources to speed up the compaction procedure. Dayan et al. offer richer space-time trade-offs by merging as little as possible to achieve given bounds on lookup cost and space and proposed a hybrid compaction policy (i.e., use levelling for the largest level and tiering for the rest) to reduce write amplifications [6, 25]. Huang et al. propose to split data in the LSM-tree into small data blocks, and reuse data blocks without overlapping key ranges during compactions extensively to reduce the write amplification [16]. All these software optimizations are orthogonal to our work on accelerating the compaction using FPGAs. We believe combined software and hardware optimization will benefit the system performance and power consumption a step further. It seems to be a fruitful area that orchestrates the hardware (e.g., CPU, FPGA, GPU) and exploit the potentials to schedule the tasks to the hardware that best suit their characteristics.

5.2 Hardware Accelerations in Databases

In the past decades, optimizing databases using carefully designed hardware techniques has been a very heated area in both the academia and the industry. Because both analytical and transactional databases manipulate data records through a well-defined set of operators or primitives (e.g., put, get, scan, join), we are able to either implement such operators using fixed hardware logics on a database-specific processor or machine [1, 12, 17, 36, 38], or do so using high-level languages (e.g., C++, OpenCL) on a general-purpose processor with assistances from latest hardware features (e.g., SIMD, high-bandwidth memory) [3, 15]. Comparing these two approaches, database-specific hardware very likely delivers groundbreaking performance and energy-efficiency with a relatively longer time-to-market and a higher engineering and financial cost per processor. And, hardware-aware opti-
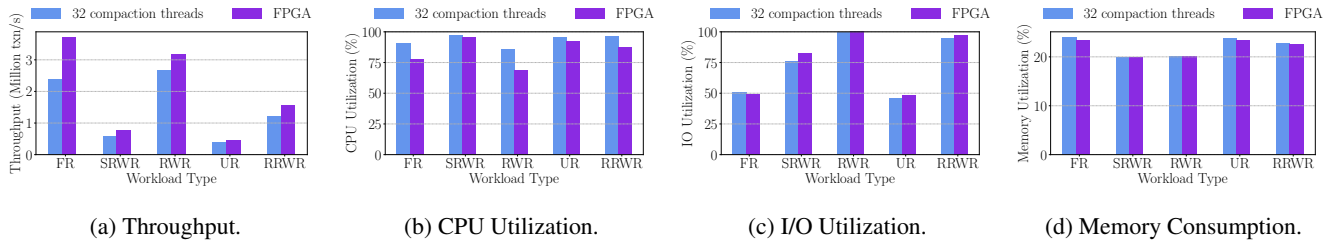
(a) Throughput. (b) CPU Utilization. (c) I/O Utilization. (d) Memory Consumption.

Figure 14: Comparing the CPU-only baseline and the FPGA-offloading proposal using the **DBBench** benchmark.

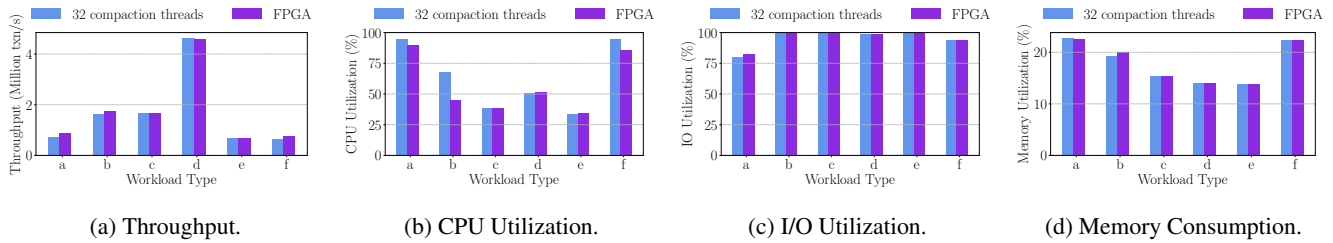(a) Throughput. (b) CPU Utilization. (c) I/O Utilization. (d) Memory Consumption.

Figure 15: Comparing the CPU-only baseline and the FPGA-offloading proposal using the **YCSB** benchmark.

mizations on general-purpose processors are much easier to program and deploy in the market with lower levels of the energy proportionality and performance speedup [32]. This work tries a third approach using FPGAs.

We explore using FPGA, a flexible and programmable hardware accelerator, for database optimizations aiming at both higher performance and higher economic values. In the literature, FPGAs have already been widely studied to accelerate individual operators or algorithms such as data partition [20], hashing [19], algorithmic trading [27], achieving significant performance improvements. For SQL accelerations, Glacier [21] is a compiler that translates SQL queries into VHDL code, targeting at streaming and standing queries. For unpredictable workloads like warehouse scenario, techniques like partial reconfiguration [10] and runtime parameterization [22] has been proposed. IBM's Netezza [14] managed to push simple selection and projection-based filtering to FPGAs. Advanced SQL offloading like `Group By` and `Where` clause has also been addressed [10, 11, 37]. The state-of-art SQL acceleration is capable of dealing with pattern matching and user-defined function [30]. Commercial examples for SQL offloading like Oracle's Exata [34] and IBM's Netezza [14] reveals the interests of the industry in this field.

Instead of stretching the arms on these *foreground* operators or algorithms, we offload a heavy and frequently executed *background* operation, compaction, to FPGAs. We argue that FPGAs can contribute to the overall database performance (a database storage in our case for now) by relieving CPUs from computation-bounded tasks, and performing much more efficiently for the offloaded tasks in a relatively isolated hardware environment.

6 Conclusion

In this paper, we argue that the LSM-tree KV store suffers from slow compactions due to resource contentions, oversized LSM-tree levels and other reasons. With the CPU and I/O consumptions of compactions carefully profiled, we find that compactions can be surprisingly bounded by computation for merging shot KVs. We further propose to offload compactions to a dedicated FPGA for acceleration and integrate our proposal with X-Engine, a state-of-the-art LSM-tree KV store. To facilitate such offloading, we have designed and implemented the pipelined *Compaction Units* on the FPGA and a *Driver* to achieve efficient offloading. With evaluations comparing our proposal with the fine-tuned CPU-only baseline, we show that the proposed FPGA-offloading of compactions can increase the system throughput and energy efficiency by up to 23% and 31.7%, respectively.

Acknowledgments

References

[1] Oliver Arnold, Sebastian Haas, Gerhard Fettweis, Benjamin Schlegel, Thomas Kissinger, and Wolfgang Lehner. An application-specific instruction set for accelerating set-oriented database primitives. In *Proceedings of the 2014 International Conference on Management of Data (SIGMOD)*, pages 767–778. ACM, 2014.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[3] Xuntao Cheng, Bingsheng He, Mian Lu, Chiew Tong Lau, Huynh Phung Huynh, and Rick Siow Mong Goh. Efficient query processing on many-core architectures: A case study with intel xeon phi processor. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2081–2084. ACM, 2016.

[4] 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. ACM, 2010.

[5] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD)*, pages 79–94. ACM, 2017.

[6] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 505–520. ACM, 2018.

[7] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, 2019.

[8] Jeff Dean and Sanjay Ghemawat. Leveldb. https://github.com/google/leveldb, 2020.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, pages 205–220. ACM, 2007.

[10] Christopher Dennl, Daniel Ziener, and Jurgen Teich. On-the-fly composition of fpga-based sql query accelerators using a partially reconfigurable module library. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 45–52. IEEE, 2012.

[11] Christopher Dennl, Daniel Ziener, and Jürgen Teich. Acceleration of sql restrictions and aggregations through fpga-based dynamic partial reconfiguration. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 25–28. IEEE, 2013.

[12] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, H-I Hsiao, and Rick Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and data engineering*, 2(1):44–62, 1990.

[13] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[14] Phil Francisco et al. The netezza data appliance architecture: A platform for high performance data warehousing and analytics. *IBM Redbooks*, 2011.

[15] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

[16] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*, pages 651–665. ACM, 2019.

[17] Balakrishna R Iyer. Hardware assisted sorting in ibm's db2 dbms. In *International Conference on Management of Data, COMAD 2005b*, 2005.

[18] Christopher Jermaine, Edward Omiecinski, and Wai Gen Yee. The partitioned exponential file for database storage management. *The VLDB Journal—The International Journal on Very Large Data Bases*, 16(4):417–437, 2007.

[19] Kaan Kara and Gustavo Alonso. Fast and robust hashing for database operators. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–4. IEEE, 2016.

[20] Kaan Kara, Jana Giceva, and Gustavo Alonso. Fpga-based data partitioning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 433–445. ACM, 2017.

[21] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on wires: a query compiler for fpgas. *Proceedings of the VLDB Endowment*, 2(1):229–240, 2009.

[22] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. Flexible query processor on fpgas. *Proceedings of the VLDB Endowment*, 6(12):1310–1313, 2013.

[23] 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.

[24] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[25] RocksDB. A persistent key-value store for fast storage environments. https://rocksdb.org/, 2020.

[26] Facebook RocksDB. Performance benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks, 2019.

[27] Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proceedings of the VLDB Endowment*, 3(1-2):1525–1528, 2010.

[28] Russell Sears and Raghu Ramakrishnan. blsm: a general purpose log structured merge tree. In *Proceedings of the 2012 International Conference on Management of Data (SIGMOD)*, pages 217–228. ACM, 2012.

[29] Pradeep Shetty, Richard P Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. In *FAST*, pages 17–30, 2013.

[30] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid cpu-fpga architectures. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD)*, pages 403–415. ACM, 2017.

[31] David Sidler, Zsolt István, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. doppiodb: A hardware accelerated database. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD)*, pages 1659–1662. ACM, 2017.

[32] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 International Conference on Management of Data (SIGMOD)*, pages 231–242. ACM, 2010.

[33] Zeke Wang, Kaan Kara, Hantian Zhang, Gustavo Alonso, Onur Mutlu, and Ce Zhang. Accelerating generalized linear models with mlweaving: a one-size-fits-all system for any-precision learning. *Proceedings of the VLDB Endowment*, 12(7):807–821, 2019.

[34] Ronald Weiss. A technical overview of the oracle exadata database machine and exadata storage server. *Oracle White Paper. Oracle Corporation, Redwood Shores*, 2012.

[35] MongoDB WiredTiger. Wiredtiger. https://github.com/wiredtiger/wiredtiger.

[36] De Witt. Direct-a multiprocessor organization for supporting relational database management systems. *IEEE Transactions on Computers*, 100(6):395–406, 1979.

[37] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment*, 7(11):963–974, 2014.

[38] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100: The architecture and design of a database processing unit. *SIGPLAN Not.*, 49(4):255–268, February 2014.

[39] Xilinx. Ultrascale architecture and product data sheet: Overview. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf, 2019.

[40] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. Pipelined compaction for the lsm-tree. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 777–786. IEEE, 2014.

[41] Shijie Zhou, Rajgopal Kannan, Yu Min, and Viktor K Prasanna. Fastcf: Fpga-based accelerator for stochastic-gradient-descent-based collaborative filtering. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 259–268. ACM, 2018.

[42] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K Prasanna. An fpga framework for edge-centric graph processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 69–77. ACM, 2018.

HotRing: A Hotspot-Aware In-Memory Key-Value Store

Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu,
Yuanyuan Sun, Huan Liu, Feifei Li
Alibaba Group

Abstract

In-memory key-value stores (KVSes) are widely used to cache hot data, in order to solve the *hotspot issue* in disk-based storage or distributed systems. The hotspot issue inside in-memory KVSes is however being overlooked. Due to the recent trend that hotspot issue becomes more serious, the lack of hotspot-awareness in existing KVSes make them poorly performed and unreliable on highly skewed workloads.

In this paper, we explore hotspot-aware designs for in-memory index structures in KVSes. We first analyze the potential benefits from ideal hotspot-aware indexes, and discuss challenges (i.e., *hotspot shift* and *concurrent access* issues) in effectively leveraging hotspot-awareness. Based on these insights, we propose a novel hotspot-aware KVS, named HotRing[1], that is optimized for massively concurrent accesses to a small portion of items. HotRing is based on an ordered-ring hash index structure, which provides fast access to hot items by moving head pointers closer to them. It also applies a lightweight strategy to detect hotspot shifts at run-time. HotRing comprehensively adopts lock-free structures in its design, for both common operations (i.e., read, update) and HotRing-specific operations (i.e., hotspot shift detection, head pointer movement and ordered-ring rehash), so that massively concurrent requests can better leverage multi-core architectures. The extensive experiments show that our approach is able to achieve 2.58× improvement compared to other in-memory KVSes on highly skewed workloads.

1 Introduction

The in-memory key-value store (KVS) is an essential component in storage infrastructures (e.g. databases, file systems) that caches frequently accessed data in memory for faster access. KVSes help to improve the performance and scalability of these systems, where billions of requests need be processed in each single second. Many state-of-the-art KVSes, e.g., Memcached [44], Redis [31] and their variants [8, 15, 17, 33],

(a) Daily distribution   (b) Extreme distribution

Figure 1: Access ratio of different keys.

are widely developed and deployed in production environments of enterprises, such as Facebook [42], Amazon [3], Twitter [49] and LinkedIn [12].

The *hotspot issue* (i.e., a small portion of items that are frequently accessed in a highly-skewed workload) is a common problem in real-world scenarios, and has been studied extensively in literature [4, 10, 20, 23, 27]. There are many solutions to address cluster-wide hotspots, such as consistent hash [29], data migration [9, 11, 46] and front-end data caching [16, 26, 32, 36]. Besides, the single-node hotspot issue is also well addressed. For example, computer architecture leverages hierarchical storage layout (e.g., disk, RAM, CPU cache) to cache frequently accessed data blocks in low-latency storage medium. Many storage systems, e.g., LevelDB [18] and RocksDB [14], use in-memory KVSes to manage hot items.

However, the hotspot issue inside an in-memory KVS is usually being overlooked. We have collected access distributions in in-memory KVSes from Alibaba's production environments, as illustrated in Figure 1. We observe that 50% (daily cases) to 90% (extreme cases) of accesses only touch 1% of total items, which shows that the hotspot issue becomes unprecedentedly serious in the Internet era. There are several reasons behind this phenomenon. First, the population of active users in online applications keeps growing. A real-time event (e.g., online sales promotions, breaking news) is able to attract billions of accesses to a few items in a short period of time, where fast access to these hotspots is critical. It has been reported that every 0.1s of loading delay would

---

[1] HotRing is a subcomponent of *Tair* — a NoSQL product extensively used in Alibaba Group and publicly available on Alibaba Cloud.

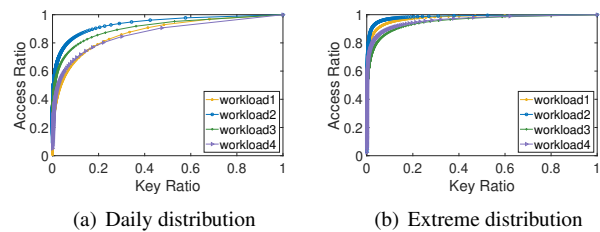cost Amazon 1% in sales, and every 0.5s of additional load delay for Google search results would lead to a 20% drop in traffic [35]. Second, infrastructures beneath such applications become complex. It is common that a minor error (e.g., due to software bugs or configuration mistakes) somewhere in the pipeline may lead to (unpredictably) repeated accesses to an item (e.g., read and return an error message endlessly). It is desired that such unpredicted hotspots shall not crash or block the entire system. Hence, keeping a KVS performant and reliable in the existence of hotspots is of great importance.

Many index structures can be used to implement a KVS, such as skip list [14, 18], balanced/trie trees (e.g., Masstree [37]), and hashes (e.g., Memcached [44], MemC3 [15], MICA [33], FASTER [8]), where hashes are the most popular due to faster lookups. However, we observe that most approaches are not aware of hotspots, i.e., they indistinguishably manage all items via a same policy. In such case, reading a hot item involves the same number of memory accesses compared to other items. From a theoretical analysis (detailed in § 2.2), we find that looking up hotspots in current hash indexes requires much more effort than from an ideal strategy. Though there exist mechanisms to reduce memory accesses, they only render limited efficacy. For example, CPU cache helps to speedup hotspot accesses, but has only 32MB of capacity. Rehash operation helps to reduce the length of each collision chain, but significantly increases the memory footprint. This situation provides us opportunities to further optimize hotspot accesses in highly-skewed workloads.

In this paper, we propose HotRing, a hotspot-aware in-memory KVS that leverages a hash index optimized for massively concurrent accesses to a small portion of items, i.e., hotspots. The initial idea is to make memory accesses required for looking up an item (negatively) correlated to its hotness, i.e., the hotter items shall be read faster. To achieve this goal, two challenges have to be addressed: *hotspot shift* - the set of hot items keeps shifting, and we need to detect and adapt to such shifts in a timely manner; *concurrent access* - hotspots are inherently accessed by massively concurrent requests, and we need to sustain high concurrency for them. For the hotspot shift issue, we replace the collision chain in the hash index with an *ordered-ring* structure, such that bucket headers can directly re-point to hot items as hotspots shift, without compromising correctness. In addition, we use a lightweight mechanism to detect hotspot shift at run-time. For the concurrent access issue, we adopt a lock-free design inspired by existing lock-free structures [19, 50], and extend it to support all operations required by HotRing, including hotspot shift detection, head pointer movement and ordered-ring rehash.

We have conducted extensive experimental evaluations on benchmarks that simulate real workload, and have compared HotRing with lock-free chain-based hashes and other baselines. The results show that, in extremely skewed workloads, HotRing processes up to 565M read requests per second, providing $2.58\times$ improvement over other systems. It

also achieves $2.17\times$ and $1.32\times$ improvement for in-place-updates and read-copy-updates respectively. This verifies that HotRing is an effective structure to improve the capability of hotspot processing on each single node, making it a performant and reliable in-memory KVS.

Our main contributions are summarized as follows:

- We identify the hotspot issue in existing in-memory indexes, and demonstrate that hotspot-aware designs have great potential to improve performance for hot items.

- We propose HotRing, an ordered-ring hash structure, as the first effort to leverage hotspot-aware designs. It provides fast access to hot items by moving head pointers closer to them. It also adopts a lightweight strategy to detect hotspot shifts at run-time.

- We make HotRing lock-free to support massively concurrent accesses. In particular, we design from scratch HotRing-specific operations, including hotspot shift detection, head pointer movement and ordered-ring rehash.

- We evaluate our approach on real-workload-based benchmarks. The results show that HotRing significantly outperforms other KVSes when the accesses are highly-skewed.

The rest of this paper is organized as follows. §2 introduces hash indexes and hotspot issues, and discusses opportunities and challenges for hotspot-aware hashes. §3 elaborates the detailed design of HotRing, and §4 evaluates its performance. Lastly, §5 reviews related work and §6 concludes the paper.

2 Background & Motivation

In this section, we first introduce the hash index and hotspot issues in existing KVSes. We then show potential benefits from ideal hotspot-aware hashes theoretically. At last, we discuss challenges to effectively leverage hotspot-awareness in practical indexes, as well as our design principles.

2.1 Hash Indexes and Hotspot Issues

Hash index is the most popular in-memory structure used in KVSes, especially when range queries are not needed by upper applications. Figure 2 illustrates the typical structure of a hash index, which contains a global *hash table* and one *collision chain* for each entry in the table. To access a key, we first calculate its hash value h to locate the corresponding entry head, and then check items in the collision chain until that key is found or the end of chain is reached (i.e., key not exists). A n-bit hash value can be further divided into a hash table part (e.g., k-bit) and a *tag* part (e.g., $(n-k)$-bit). The tag can be included in each item to avoid comparing long keys [8, 33]. As can be seen in Figure 2, hash indexes are not aware of hotspots, i.e., hot items might be distributed evenly
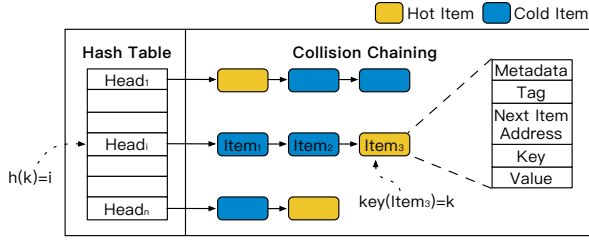
Figure 2: The conventional hash index structure.

in collision chains, For a hot item placed close to the tail of the collision chain (e.g., $Item_3$ in the figure), it requires more memory accesses than other items in front. However, in highly skewed workloads, slight increase of hot-item access cost may result in severe decline of overall performance.

There are several ways to reduce hot-item access cost, however, with only limited effects. First, CPU caches can speedup accesses to hot data blocks (i.e., in the unit of a 64-byte cacheline). However, for most commodity servers, the capacity of CPU cache is around 32 MB, while the entire memory volume exceeds 256 GB. Only 0.012% of memory can be cached, far less than observed hotspot ratios in Figure 1. To better utilize CPU cache, many cache-friendly index structures [8, 33] are proposed. Second, the hash table can be enlarged (i.e., via *rehash*) to reduce lengths of collision chains, so that locating a hot item needs fewer memory accesses. However, rehash is no longer advised when the hash table is already huge in size. For example, for two successive rehash operations, the second one requires two times the memory space, but only brings in half of the efficacy (in terms of the chain length reduction). In summary, all existing approaches only mitigate the hotspot issue to a small extent.

2.2 Potential Benefits of Hotspot-Awareness

As hotspot issue is getting serious (shown in Figure 1), it renders a rising opportunity to the design of hotspot-aware hash indexes. First of all, it is interesting to have a rough estimation and analysis on how much potential benefits we can obtain from leveraging hotspot-aware designs.

In conventional chain-based hash indexes, hot items are randomly placed in the collision chain, so that hot items and cold items are equivalent in terms of access cost. Suppose that we have N items (i.e., key-value pairs) stored in a hash table with B buckets, the average length of each bucket chain is $L = N/B$. The number of expected memory accesses to retrieve an item in the chain E_{chain} is

$$E_{chain} = 1 + \frac{L}{2} = 1 + \frac{N}{2 \cdot B} \qquad (1)$$

where the leading 1 represents the lookup in the hash table.

In an ideal hotspot-aware hash index, memory accesses required to retrieve an item should be (negatively) correlated to this hotness, e.g., the hottest item needs the fewest memory
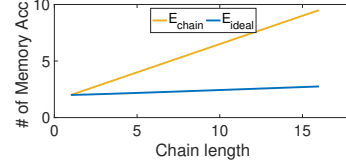

Figure 3: Expected memory accesses for an index lookup (total items $N = 2.5 \cdot 10^8$).

accesses to retrieve. We model item hotness in a Zipfian distribution, where the access frequency f of the x-th hottest item is expressed as:

$$f(x) = \frac{\frac{1}{x^\theta}}{\sum_{n=1}^{N} \frac{1}{n^\theta}} \qquad (2)$$

where θ is the skewness factor. To simplify the analysis, we assume that hotspots are evenly distributed in B buckets, i.e., each bucket contains exactly one from the top B hottest items, one from the top $B + 1$ to $2B$ hottest items, and so on. In this case, if we can sort all items in a chain by their access frequencies (in descending order), the number of expected memory accesses to retrieve an item E_{ideal} is

$$\begin{aligned} E_{ideal} &= 1 + \sum_{k=1}^{L} F(k) \cdot k \\ &= 1 + \sum_{k=1}^{\frac{N}{B}} \left[\sum_{i=(k-1) \cdot B+1}^{k \cdot B} f(i) \right] \cdot k \end{aligned} \qquad (3)$$

where $F(k)$ represents the accumulated access frequencies of the k-th item on each chain.

To estimate the potential benefits from hotspot-aware designs, we calculate the expected number of memory accesses for both traditional hash and ideal hotspot-aware hash, as shown in Figure 3. We can observe that, as collision chain length keeps growing, hotspot-aware hash significantly improves the access efficiency. This result confirms that the consideration of hotspot-awareness in a hash index is a promising direction for performance improvement.

2.3 Challenges and Design Principles

We have shown that making an index hotspot-aware is beneficial. However, there remains several challenges before we can leverage this insight in practical designs:

- **Hotspot Shift.** In real applications, access patterns keep changing over time. It is prohibitive to order all items ideally by their latest hotness. Hence, we need a lightweight approach to track the shift of hotness.

- **Concurrent Access.** Each hotspot is being inherently accessed by massively concurrent requests. Therefore, it is critical to support high concurrency for both read/write operations, in order to sustain satisfactory performance.

For the hotspot shift problem, our design principle is to avoid re-order items in the chain, but to move head pointers
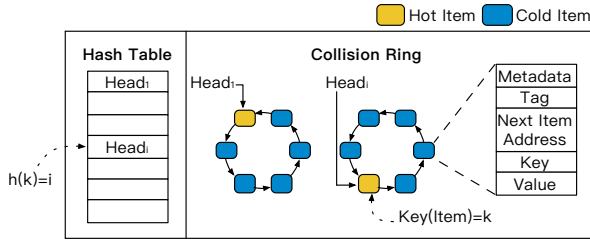
Figure 4: The index structure of HotRing.

instead, e.g., point to the hottest item or a globally better position. To ensure all items in a bucket are always accessible regardless of head pointer movement, we replace the collision chain with a *ordered-ring* structure, called HotRing (§3.1). Though this design cannot achieve optimal hotspot-awareness discussed in §2.2, we observe in experiments that it is sufficiently effective and fast. Besides, we apply two lightweight strategies to detect hotspot shift at run-time (§3.2).

For the concurrent access problem, lock-free structures are canonical solutions, with which expensive lock and synchronization operations are eliminated. Many works have demonstrated that lock-free designs can significantly improve system throughput [2, 5, 21]. Examples include read-copy-update (RCU) [13] and Hazard Pointers [40], based on atomic Compare-And-Swap (CAS). In our work, the lock-free design adopts the existing work [19, 50], which ingeniously solves the concurrency problem of deletions and insertions (§3.3). We extend this design to support all basic operations required by HotRing, including hotspot shift detection, head pointer movement and rehash (§3.4).

3 Design of HotRing

In this section, we elaborate detailed designs in HotRing that adopt hotspot-awareness, including the index structure, hotspot-shift detection strategies, and lock-free operations (i.e., read/write, insert/delete, head pointer movement, and rehash).

3.1 Ordered-Ring Hash Index

Figure 4 depicts the index structure of HotRing, which refines the structure of collision chains in conventional hash indexes. In our design, the last item in the chain is linked to the first item, forming a *collision ring*. In this manner, a head pointer in the hash table can point to any items in the corresponding ring, rather than being fixed to the front item in the chain. The design of collision ring makes it possible for HotRing to move the head pointer according to the data hotness, and scan the entire ring from any starting position. Note that if there is a single item in the ring, its next-item pointer just points to itself.

However, due to the ring-based design, there exists a serious problem: if the target item is not found, it may lead to infinite traverses in the ring. It is important to figure out when we can
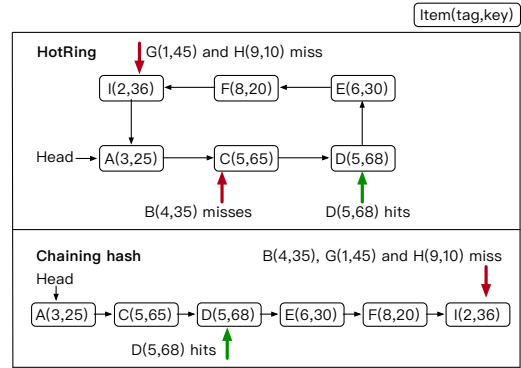

Figure 5: Find operation in HotRing.

safely terminate the lookup process. Note that it is insufficient to mark the first item pointed by the head pointer as the stop signal, because it can be modified by concurrent requests (e.g., the marked item is deleted). Hence, we propose an ordered-ring structure to help determine the termination of lookup processes. Intuitively, we can sort items in the ring by their keys. In this case, the occurrence of item-not-found can be determined if we have already encountered two successive items that are respectively smaller and larger than the target item. Furthermore, since comparing two long keys might be costly, we utilize the tag field (introduced in §2.1) first. That is to say, an item k is ranked by the pair of its tag and key fields, i.e., $order_k = (tag_k, key_k)$.

During a lookup process for item k, suppose that the item i is being accessed, we can immediately terminate if one of the following conditions satisfies.

Condition for Item Found (*Hit*) :

$$order_i = order_k \tag{4}$$

Conditions for Item Not Found (*Miss*) :

$$\begin{cases} order_{i-1} < order_k < order_i \\ \text{or } order_k < order_i < order_{i-1} \\ \text{or } order_i < order_{i-1} < order_k \end{cases} \tag{5}$$

Figure 5 illustrates all possible situations of looking up an item in HotRing. We show the dictionary order (tag, key) of each item in the figure. For example, the item C is behind item A due to $tag_A < tag_C$; and the item D is behind item C (with the same tag), because of $key_C < key_D$. The item B is confirmed to be a miss when compared to the item C, because of $tag_A < tag_B < tag_C$; the items G and H are misses when compared to the item I, because of $tag_G < tag_I < tag_F$ and $tag_I < tag_F < tag_H$ respectively. Unlike the traditional chain-based hashes, not all items in the ring have to be accessed before a miss is concluded. Assume that a ring contains n items, we only need to compare with $(n/2) + 1$ items in average for a lookup.

3.2 Hotspot Shift Identification

In ordered-ring hash index, the lookup process can easily determine whether there is a hit or miss. The remaining problem is how to identify hotspots and adjust head pointer when hotspot shift occurs.

Hotspot items are evenly distributed in all buckets, due to strongly uniformed distribution of hash values. Here, we focus on hotspot identification in each bucket independently. In practice, the number of collision items in each bucket is relatively small (e.g., 5 to 10 items), so that there is usually one hotspot in each collision ring (under 10% - 20% hotspot ratio). We can improve the hotspot access by pointing the head pointers to the only hotspot, which avoids re-organizing data and reduces memory overhead. To obtain good performance, two metrics have to be concerned, i.e., identification accuracy and reaction delay. The accuracy of hotspot identification is measured by the proportion of identified hotspots. The reaction delay is the time span between the time a new hotspot occurs and the time we successfully detect it. Considering both metrics, we first introduce a *random movement* strategy that identifies hotspots with extremely low reaction delay. We then propose a *statistical sampling* strategy that provides much higher identification accuracy with relatively high reaction delay.

First of all, we define several terms used throughout this section. The first item pointed by the head pointer is called the *hot item*, and the rest items are *cold items*. Their accesses to them are defined as *hot access* and *cold access*, respectively.

3.2.1 Random Movement Strategy

Here we introduce a straightforward random movement strategy, which retains less reaction delay but achieves relatively low accuracy. The basic idea is that the head pointer is periodically moved to a potential hotspot from an instant decision, without recording any historical metadata. In particular, each thread is assigned a thread-local parameter to record the number of requests it executes. After every R requests, the thread determines whether to perform a head pointer movement operation. If the R-th access is a *hot access*, the position of head pointer remains unaffected. Otherwise, the pointer is moved to the item accessed by this *cold access*, which becomes the new *hot item*. The parameter R affects the reaction delay and identification accuracy. If a small R is used, the reaction delay to achieve stable performance will be low. However, this may also adversely cause frequent and ineffective head pointer movement. In our scenarios, data accesses are highly skewed and hence the head pointer movement tends to be infrequent. The parameter R is empirically set to 5 by default, which has been demonstrated to provide low reaction delay and negligible performance impact (as shown in Figure 15(b)).

Note that if the workload skewness is not that obvious, the random movement strategy will become inefficient. More importantly, this strategy is unable to handle multiple hotspots
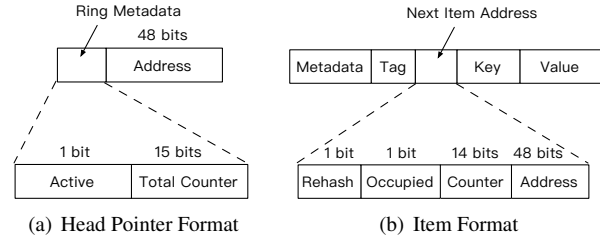

Figure 6: HotRing Index Format.

in a collision ring. In this case, the head pointer tends to move frequently, which does not help to speedup hotspot accesses but adversely affects normal operations.

3.2.2 Statistical Sampling Strategy

In order to achieve higher performance, we design a statistical sampling strategy that aims to provide more accurate hotspot identification with slightly higher reaction delay. We first introduce the detailed formats of items and pointer in HotRing and show how to take advantage of existing formats to maintain statistics without additional space overhead. Then, we elaborate the sampling strategy to estimate access frequencies. Finally, we propose a way to derive the optimal head pointer movement when hotspot shifts, considering that multiple hotspots may exist in a ring.

Index Format. We plan to record access frequencies for all items in each collision ring. Since the physical address of modern machines occupies only 48 bits (but can be updated with 64-bit atomic compare-and-swap operations), we can utilize the remaining 16 bits to record metadata. In HotRing, each head pointer consists of three parts (as shown in Figure 6(a)): an *Active* bit, a *Total Counter* (15 bits), and the address (48 bits). The *Active* bit is a flag used to control statistical sampling for hotspot identification. The *Total Counter* records the number of accesses to the corresponding ring. Besides, the structure of an item is shown in Figure 6(b). *Rehash* is a flag to control rehash process (discussed in §3.4). *Occupied* is used to ensure concurrency correctness (discussed later in this section). HotRing uses the remaining 14 bits in the *Next Item Address* to record access counts for each item. Based on statistics maintained at both ring level and item level, the calculation of access frequencies is straightforward.

Statistical Sampling. How to dynamically identify hotspots with low overhead is a challenging problem. The hash table is usually large, e.g., contains $2^{27} \sim 2^{30}$ buckets. The simultaneous and continuous updates of statistics on massive rings will cause severe performance degradation. It is critical to minimize the overhead while retain the accuracy, which is achieved by periodical sampling in HotRing. In particular, each thread maintains a thread-local counter for processed requests. After every R requests are finished, we determine whether to launch a new round of sampling (by turning on the *Active* flag in Figure 6(a)). If the R-th access is a *hot access*, it means that the current hotspot identification is still accurate,

and sampling needs not be triggered. Otherwise, it means the hotspot has shifted and we start the sampling. The parameter R is set to 5, following similar considerations as in §3.2.1. When the *Active* bit is set, the subsequent accesses to the ring are to be recorded in both *Total Counter* and corresponding items' counters. This sampling process requires additional CAS operations, and results in temporary access deficiency. To shorten this period, we set the number of samples the same as the number of items in each ring, which we believe already provides enough information to derive new hotspots.

Hotspot Adjustment. Based on collected statistics, we are able to determine new *hot item* and move the head pointer according to the access frequencies of items. After sampling process is done, the last accessing thread is responsible for frequency calculation and hotspot adjustment. First, the thread atomically resets the *Active* bit using a CAS primitive, which ensures that only one thread will perform subsequent tasks. Then, this thread calculates the access frequency of each item in the ring. The access frequency of item k is n_k/N, where N is *Total Counter* of the ring and n_k is the counter of the k-th item. Next, we calculate the income of the head pointer to each item. When the item t $(0 < t < k)$ is pointed by the head pointer, the corresponding income W_t is calculated by the following formula:

$$W_t = \sum_{i=1}^{k} \frac{n_i}{N} * [(i-t) mod\ k] \tag{6}$$

The income W_t measures the average number of memory accesses for the ring when item t is selected to be pointed by the head pointer. Therefore, selecting the item with the $min(W_t)$ as the *hot item* ensures that hotspots can be accessed faster. If the calculated position is different from the previous head, the head pointer should be moved using a CAS primitive. Note that the strategy not only deals with single hotspot, but also works for multiple hotspots. It helps to figure out the optimal position (e.g., may not necessarily be the hottest item) that avoids frequent movement between hotspots. After the hotspot adjustment is done, the responsible thread resets all counters to prepare for the next round of sampling in future.

Write-Intensive Hotspot with RCU. For update operations, HotRing provides an in-place update method for those values less than 8 bytes (i.e., modern machines support atomic operations for up to 8 bytes). In this case, reading and updating an item is treated the same in terms of hotness. However, the situation is completely different for larger values, as shown in Figure 7. The read-copy-update (RCU) protocol has to be applied for high performance. In this case, the preceding item's pointer needs to be modified to point to the new item during an update. If the write-intensive hotspot in the head is modified, the entire collision ring has to be traversed to reach its preceding item. That is to say, a write-intensive hot item also makes its preceding item hot. Taking this insight, we modify the statistical sampling strategy slightly. For a RCU update, the counter of its preceding item is incremented instead. This
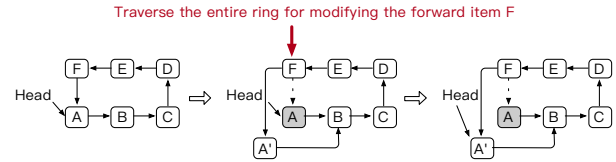

Figure 7: Updating *hot item* A with RCU makes item F hot.

helps to point the head to the precedent of a write-intensive hotspot, making the entire RCU update operation fast.

3.2.3 Hotspot Inheritance

When performing RCU update or deletion on the head item, we need to move the head pointer to another item. However, if the head pointer is moved randomly, it may point to a *cold item* with a high probability, which will cause the hotspot identification strategies to be triggered frequently. Furthermore, the performance of the system will be seriously degrades due to frequent triggering of identification strategies.

First of all, if the collision ring has only one item (i.e., the *Next Item Address* has the same position as the head pointer), the head pointer is modified by CAS to complete the update or deletion. If there are multiple items, HotRing uses existing hotspot information (i.e., head pointer position) to inherit the hotness. We design different head pointer movement strategies for both RCU update and delete operations to ensure the validity of hotspot adjustment: For the RCU update of the head item, the most recently updated item has a high probability of being accessed immediately due to the temporal locality of accesses. Hence, the head pointer is moved to the new version of the head item. For the deletion of the head item, the head pointer is simply moved to the next item, which is a straightforward and effective solution.

3.3 Concurrent Operations

The head pointer movement makes the lock-free design more complicated. This is mainly reflected in the following aspects: On one hand, the head pointer movement may be concurrent with other threads. Hence, we need to consider the concurrency of head pointer movement and other modification operations, preventing the pointer from pointing to invalid items. On the other hand, when we delete or update an item, we need to check if the head pointer is pointing to the item. If so, we need to move the head pointer correctly and smartly. In this section, we mainly introduce the control method of concurrent access to solve the concurrency problem in HotRing. In order to achieve high access concurrency and ensure high throughput, we have implemented a complete set of lock-free designs, which has been rigorously introduced by previous work [19, 50]. The atomic CAS operation is used to ensure that two threads will not modify the same *Next Item Address* simultaneously. If multiple threads are trying to update the same *Next Item Address*, only one thread succeeds and others fail. Failed threads have to re-execute their operations.
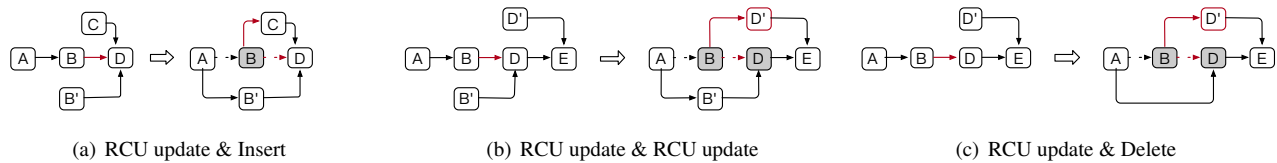
| (a) RCU update & Insert | (b) RCU update & RCU update | (c) RCU update & Delete |

Figure 8: Different concurrency issues that involve RCU operations.

Read. HotRing scans the collision ring to search for an item with the target key as described in Sec. 3.1. No additional operations are required to ensure correctness of read operations. Therefore, the read operations are completely lock-free.

Insertion. The insertion create a new item (e.g., item C in Figure 8(a)), and modify the preceding item's *Next Item Address* . Two concurrent insertions may compete for the same *Next Item Address*. The CAS ensures that only one succeeds and the other has to retry.

Update. We design two update strategies for different value sizes. The in-place-update operation (for 8-byte values) does not affect other operations, which is guaranteed through CAS. However, the RCU operation (for longer values) needs to create a new item, which challenges the concurrency of other operations. Taking the RCU update & Insert as an example in Figure 8(a): one thread is trying to insert the item C by modifying the *Next Item Address* of the item B, and another thread is trying to update the B with B′ concurrently. Operations of both threads will succeed since they modify different pointers with CAS. However, since item B is not visible to the ring, even though the insert operation for item C has succeeded, it cannot be accessed subsequently and lead to incorrectness. The same problem exists in Figure 8(b). In order to solve the problem, HotRing uses the *Occupied* bit (as shown in Figure 6(b)) to ensure correctness. We perform the update operation in two steps. For example, in the case of Update & Insert: Firstly, the *Next Item Address* of item B that to be updated is atomically set as occupied. Once the *Occupied* bit is set, the insertion of Item C will fail and have to retry. Secondly, the *Next Item Address* of item A is atomically changed to item B′ and the *Occupied* bit for B′ is reset.

Deletion. The deletion is achieved by modifying the pointer to the deleted item to its next item. Therefore, it must be ensured that *Next Item Address* of the deleted item is not changed during the operation. Similarly, we utilize the *Occupied* bit to ensure correctness of concurrent operations. For the case of RCU update & Delete as shown in Figure 8(c), the update for item D is processed by updating the forward item B's pointer, while item B is currently being deleted. The updated item D′ cannot be traversed correctly, resulting in data miss. If the *Occupied* bit of item B is set for deletion, the update for item D will fail to modify item B's *Next Item Address* and have to retry. Once the deletion of item B is completed, the update operation can be successfully executed.

Head Pointer Movement. The movement of the head pointer is a special action in HotRing. In order to ensure the correctness of the head pointer movement with other oper-

ations (especially for update and deletion), we need additional management. There are two major problems that need to be addressed: (1) how to handle the concurrency of normal operations and the head pointer movement caused by identification strategies? (2) how to handle the head pointer movement, caused by updating or deleting of the head item?

For the head pointer movement caused by the identification strategies, we also use the *Occupied* bit to ensure correctness. When moving the head pointer to a new item, we set its *Occupied* bit to ensure that the item will not be updated or deleted during the movement. For head item update, HotRing moves the head pointer to new version of this item. Before moving the head pointer, we need to ensure that the new version item will not be changed (i.e., updated or deleted) by other threads. Therefore, when updating the item, HotRing sets the *Occupied* bit of the new version item first, until the movement is completed. For head item deletion, HotRing needs to occupy not only the item that is ready to be deleted, but also its next item. Because if the next item is not occupied during the deletion operation, the next node may have been changed, which makes the head pointer points to an invalid item.

3.4 Lock-free Rehash

As new data arrives from insertions, the number of collision items in a ring continues to increase, resulting in traversing more items per access. In this case, the performance of KVSes will be seriously degraded. We propose in HotRing a lock-free rehash strategy that allows for flexible rehash as data volumes increase. The conventional rehash strategy is triggered by the load factor (i.e., average length of chain) of the hash table. However, this fails to consider the effect of hotspots, and hence is unsuitable for HotRing. In order to make the index adapt to the growth of hotspot items, HotRing uses the access overhead (i.e., average number of memory accesses to retrieve an item) to trigger the rehash. Our lock-free rehash strategy includes three steps:

Initialization. First of all, HotRing creates a backend rehash thread. The thread initializes the new hash table that is twice the size of the old one, by sharing the highest bit of the tag. As shown in Figure 9(a), There is an old head pointer in an *Old Table*'s bucket, and there are two new head pointers in the *New Table*' correspondingly. The number of bits required for hash is expanded from k to $k + 1$. HotRing divides data based on the tag range. Assuming that the hash value has n bits, and the tag range is $[0, T)$ ($T = 2^{(n-k)}$), two new head pointers manage items from $[0, T/2)$ and $[T/2, T)$

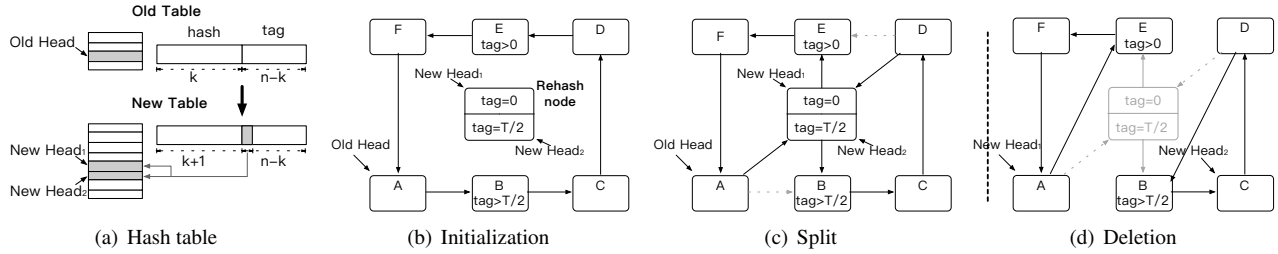| (a) Hash table | (b) Initialization | (c) Split | (d) Deletion |

Figure 9: The lock-free rehash strategy (The dotted line between (c) and (d) represents a transition period before deletion).

respectively. Meanwhile, the rehash thread creates a *rehash node* consisting of two child rehash items, which correspond to two new head pointers respectively. Each rehash item has the same format as data item, except that no valid KV pair is stored. HotRing identifies rehash items by the *Rehash* bit in each item. In the initialization phase, the tags of two child rehash items are set differently. As shown in Figure 9(b), the corresponding rehash items set the tags to 0 and T/2, respectively.

Split. In the split phase, The rehash thread splits the ring by inserting two rehash items into it. As shown in Figure 9(c), rehash items are inserted before item B and item E respectively, becoming the boundaries of the tag range to divide the ring. When two insertion operations complete, the *New Table* is made active. After that, subsequent accesses (from *New Table*) need to select the corresponding head pointer by comparing the tags, while previous accesses (from *Old Table*) proceed by identifying the *rehash node*. All data can be accessed correctly without effecting concurrent reads and writes. Until now, accesses to items are logically divided into two paths. When we look up a target item, at most half of the ring needs to be scanned. For example, the traversal path for accessing item F is $Head_1 \rightarrow E \rightarrow F$.

Deletion. In this phase, the rehash thread delete the *rehash nodes* (as shown in Figure 9(d)). Before that, the rehash thread have to maintain a transition period to ensure that all accesses initiated from the *Old Table* have finished, such as the grace period for read-copy-update synchronization primitive [13]. When all accesses end, the rehash thread can safely delete the *Old Table* and then *rehash nodes*. Note that the transition period only blocks the rehash thread, but not access threads.

4 Evaluation

In this section, we evaluate the performance of HotRing using real-workload-based benchmarks. In particular, we compare the throughput and scalability of HotRing with lock-free chain-based hash and other baseline systems. We also provide detailed evaluations to demonstrate the effectiveness of major designs adopted by HotRing.

4.1 Experimental Setup

Environment. We run experiments on a machine consisting of two Intel(R) Xeon(R) CPU E5-2682 v4 with 2.50GHz

Table 1: Experimental environment.

CPU	2.50GHz Intel Xeon(R) E5-2682 v4 * 2
L2 cache	256KB (512 * 8 way)
L3 cache	40MB (32768 * 20 way)
Cache Alignment	64B
Main Memory	32GB 2133MHz DDR4 DRAM * 8

Table 2: Hotspot access ratio with different hotspot definition (i.e., top α of hottest items) and zipfian parameter (θ).

θ \ α	1%	10%	20%	30%	40%	50%
0.5	9.9%	31.6%	44.7%	57.7%	63.2%	70.7%
0.7	24.9%	50.0%	61.6%	71.9%	75.9%	81.2%
0.9	57.3%	76.2%	82.2%	86.9%	89.9%	92.2%
0.99	75.1%	87.4%	91.2%	93.4%	94.9%	96.2%
1.11	91.7%	96.4%	97.6%	98.2%	98.7%	99.0%
1.22	97.8%	99.2%	99.5%	99.6%	99.7%	99.8%

processors. They have 2 sockets, each with 16 cores (64 hyperthreads in total). The machine has 256GB RAM capacity, and runs CentOS 7.4 OS with Linux 3.10 kernel. To achieve better performance, we bind each thread to the corresponding core. Table 1 summarizes the detailed hardware configuration of the machine.

Workloads. We conduct experiments using the YCSB core workloads [10], except workload E that involves scan operations. For each item (i.e., key-value pair), we set the key size to be 8 bytes, and the value size to be 8 and 100 bytes for in-place-update and read-copy-update (RCU) respectively. In each test, the number of loaded keys is fixed to 250 millions, and the key-bucket ratio (i.e., the number of keys divided by the number of buckets) varies to control average length of collision chains. In addition, we tune the zipfian distribution parameter θ in YCSB to generate workloads that simulate daily and extreme hotspot scenarios. Table 2 shows the ratio of hotspot accesses with different hotspot definitions and skewness. Recall that Figure 1 shows the workload distributions in our production environments. We observe that θ falls in [0.9, 0.99] for daily scenarios, and in [1, 1.22] for extreme cases. Hence, we choose 0.99 and 1.22 for θ as representa-
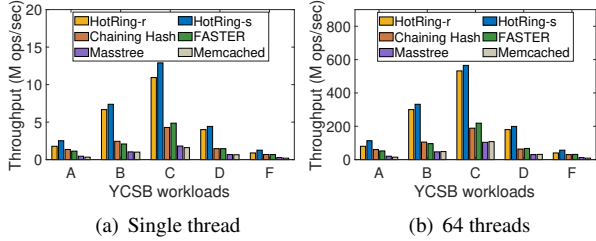
(a) Single thread   (b) 64 threads

Figure 10: Throughput of HotRing and other systems.

(a) Impact of chain length   (b) Impact of θ

Figure 11: Performance impact of collision chain length and access skewness.

tives.

Baselines. In order to better demonstrate the advantage of hotspot-aware designs in HotRing, we implement a lock-free chain-based hash index as a baseline (Chaining Hash). It is modified from the hash structure in Memcached, and uses the CAS primitive to insert new items to the head of collision chains. We also compare to other KVS systems: the C++ version of FASTER [7], in which we ensure all data resides in memory; Masstree [30], a high-performance in-memory range index that is a representative KVS with non-hash indexes. In addition, the lock-based Memcached [44] is also included as a reference.

Note that the memory footprint of an index structure greatly affects system performance. In order to have fair comparisons, we strictly make the memory consumption of indexes the same for all approaches. In each test, if not otherwise specified, we use following default settings: 64 threads, 8-byte value payloads, workload B of YCSB, θ set to 1.22, and key-bucket ratio set to 8 (for HotRing).

4.2 Comparison to Existing Systems

We evaluate HotRing against four baselines introduced above, i.e., Chaining Hash, FASTER, Masstree and Memcached, which are all high-performance KVS implementations.

Overall performance. Figure 10 shows the overall system throughput of all approaches on different YCSB workloads. We run two HotRing variants with distinct hotspot identification strategies: HotRing-r adopts random movement strategy, and HotRing-s adopts sampling statistics strategy. Compared to other systems, HotRing achieves better performance in throughput under all workloads, especially for workloads B and C. HotRing-s outperforms other approaches by 2.10× - 7.75×. It achieves 12.90M ops/sec with a single thread and 565.36M ops/sec with 64 threads, which implies promising scalability. Besides, HotRing also keeps advantages for insertion operations. For workloads D and F of YCSB (with massive insertions), HotRing-s outperforms other approaches by 1.81× - 6.46×. This is because the ordered-ring structure speeds up item location by early termination, while the tag field reduces the cost of sorting. Though the hotspot identification of HotRing-r is less accurate than HotRing-s (about 7% worse), its overall performance is still significantly improved compared to other systems.

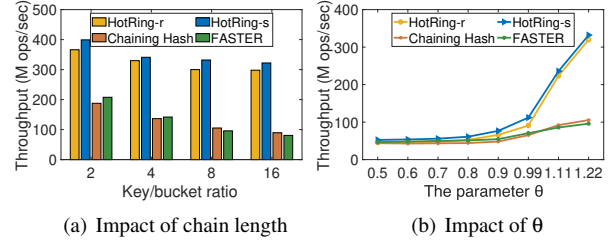**Collision chaining length.** Figure 11(a) shows the

throughput of different approaches when we vary the length of collision chains. We tune the key-bucket ratio from 2 to 16, which means that the conflicts in the hash table become more intense. We can see that Chaining Hash and FASTER have good performance when key-bucket ratio is 2. This is because when the collision chain is short, the memory access overhead for hotspot items is relatively minor, making the effect of cache more significant (especially for FASTER). However, as the length of the collision chain increases, frequent accesses to hotspot items drop the performance of Chaining Hash and FASTER seriously. In contrast, HotRing retains satisfactory performance even for long chains. In particular, when the key-bucket ratio is 2, its read throughput is 1.93× and 1.31× compared to that of Chaining Hash and FASTER. When the ratio becomes 16, the performance gap increases to 4.02× and 3.91× respectively. This is because HotRing puts hot items close to the head, so that less memory accesses are required. This design is more cache-friendly, where only head pointers and hotspot items need be cached, rendering higher performance. Therefore, we conclude that HotRing has better performance and scalability due to its hotspot-aware designs.

Access skewness. Figure 11(b) shows the throughput of different approaches when the zipfian parameter θ varies. We tune θ from 0.5 to 1.22, which means that the hotspot issue in workloads become more severe. As can be seen, the performance improvement in both Chaining Hash and FASTER is not obvious as θ increases, since they lack hot-aware considerations. In contrast, the performance of HotRing significantly improves as θ increases, especially when θ is greater than 0.8. Even when θ is in [0.5, 0.8] range, where hotspot issue is minor, HotRing-s still achieves better performance than others. This is because HotRing-s is able to handle the case of multiple hotspots in the collision ring. When there are multiple items with similar access frequencies, HotRing-s can find the best head pointer position to achieve optimal performance (§3.2.2). However in this case, HotRing-r fails to choose the optimal head pointer position, leading to frequently-triggered pointer movements.

RCU operation. In order to show the performance of RCU more prominently, we use YCSB to generate write-intensive workloads with 100-byte value payloads (both 50% write and write only). Figure 12 shows the throughput of different approaches when RCU operations are involved. In this test, we
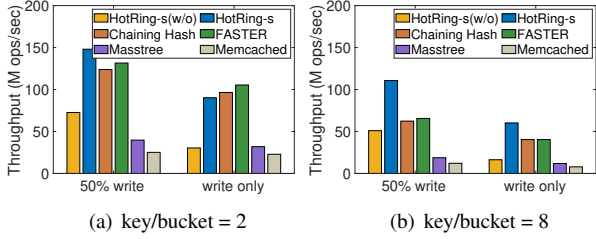
(a) key/bucket = 2   (b) key/bucket = 8

Figure 12: Performance of RCU operations.

(a) θ = 1.22   (b) θ = 0.99

Figure 13: Break-down cost.

(a) θ = 1.22   (b) θ = 0.99

Figure 14: Reaction delay.

(a) 100% read miss   (b) Impact of *R*

Figure 15: Performance impact of read miss and *R*.
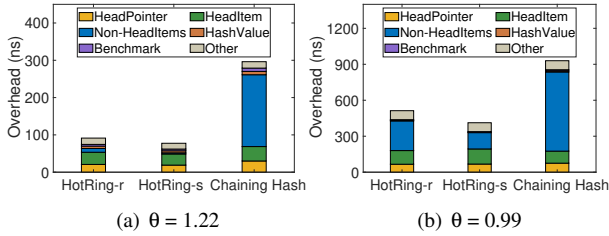
demonstrate the need for special processing of RCU operation in HotRing (§3.2.2). In particular, HotRing-s indicates that the sampling statistics strategy will increase the forward item counter instead when an item is updated by RCU. HotRing-s(w/o) represents the strategy without distinguishing RCU operations. Firstly, HotRing-s(w/o) has poor performance in all cases, even worse than Chaining Hash and FASTER. This is because HotRing-s(w/o) needs to traverse the entire collision ring for completing an RCU operation on the hot item. However in HotRing-s, the optimized hotspot counting strategy significantly improves RCU performance. Note that when key-bucket ratio equals 2, the performance of HotRing-s is slightly slower than that of FASTER. This is because the hotspot item requires RCU operation at the second slot pointed by the header point, where one additional memory access is needed. Furthermore, it involves one extra CAS operation (on *Occupied* bit) to complete the RCU operation. As the number of collision items keeps increasing, above issues will be greatly mitigated. For example, when key-bucket ratio reaches 8, the throughput of HotRing-s is 1.32× better than Chaining Hash and FASTER.

4.3 Investigation of Detailed Designs

In this section, we compare HotRing with the conventional chaining hash, in order to investigate the advantages of hotspot-aware designs.

Break-down cost. We collects the break-down cost of different functions involved during workload execution. Figure 13 shows the average break-down cost for a single read access in HotRing and Chaining Hash (where key-bucket ratio is 8). In this figure, *HeadPointer* is the cost to locate the head pointer of the corresponding collision ring or chain; *HeadItem* is the cost to access the head item; *Non-HeadItem* is the cost to access other items; *HashValue* is the cost due to the hash calculation; *Benchmark* is the cost to read and
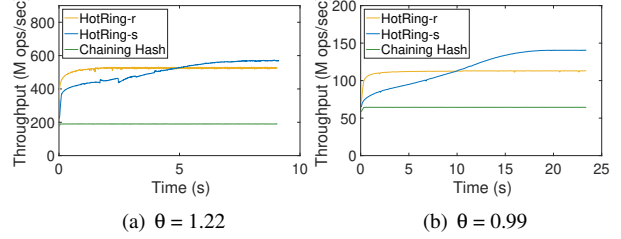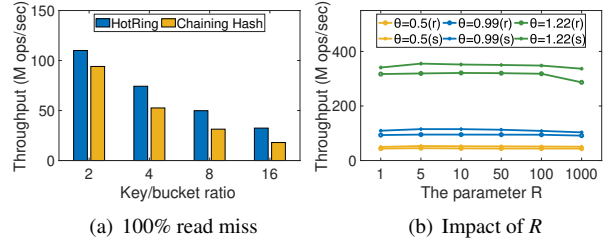
interpret the workload command; and *Other* is the cost from the system kernel. As can be observed, the cost in Chaining Hash is mainly dominated by *Non-HeadItem* accesses, which is about 193ns/660ns when θ = 1.22/0.99. This indicates that hotspot items in chain-based hash tend to be evenly distributed in the chain, increasing the access cost significantly. In contrast, HotRing-r and HotRing-s significantly reduce the *Non-HeadItem* cost. Especially for HotRing-s, the *Non-HeadItem* cost is about 10ns/136ns when θ = 1.22/0.99. Since the proportion of *Non-HeadItem* is negatively related to the hotspot identification accuracy, this implies that HotRing-s has higher hotspot identification accuracy where more hotspot items are detected and placed at the head.

Reaction delay. Reaction delay is one of important metrics for measuring hotspot identification strategies. Figure 14 shows the throughput trends over time after hotspots have shifted (workload C). We can observe that HotRing-r has faster reaction than HotRing-s, which only takes less than 2 seconds to reach stable state. However, its peak throughput is much lower due to its inaccurate hotspot detection. The throughput of Chaining hash is not affected since it lacks hotspot-awareness.

Read miss. We evaluate the throughput of both approaches for handling read misses as shown in Figure 15(a). As can be seen, the performance gap between HotRing and Chaining Hash expands with the increase of chain length. In particular, HotRing achieves 1.17× improvement when the key-bucket ratio is 2, and 1.80× when the ratio reaches 16. This is because that HotRing only needs to compare with half of items in average for a lookup (as shown in Figure 5), while Chaining Hash accesses all items in the chain.

Parameter *R*. Recall that the choice of parameter *R* affects the frequency of head pointer movement (§3.2). When *R* is small, the hotspot identification has less reaction delay, but results in more frequent (and invalid) head pointer movements.
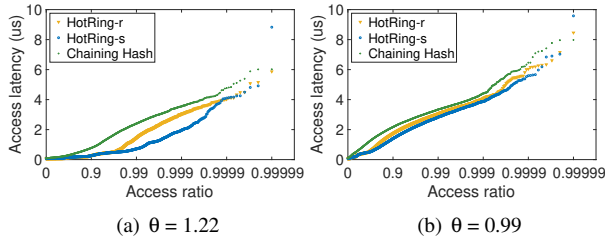
(a) θ = 1.22   (b) θ = 0.99

Figure 16: Tail latency.

Figure 17: Rehash performance.

Figure 15(b) shows the impact of R on overall throughput in different scenarios. It can be observed that the performance get slightly worse when R is either too small (due to overheads from hotspot identification) or too large (due to delayed handles of hotspot shifts). In practice, we set R to 5 for balanced consideration and better throughput.

Tail latency. HotRing-s requires statistical sampling and the last thread during a sampling process needs to calculate access frequencies to find the best position for the head pointer. Hence, there might exist long-tail accesses due to such additional computations. Figure 16 shows the latency distribution of 100 thousands accesses. When θ = 1.22, the 99-percentile response time is about $2\mu s$, but there are long-tail accesses requiring $8.8\mu s$. It is similar when θ = 0.99, where 99-percentile response time is $3\mu s$ and long-tail access time is $9.6\mu s$. Note that the long-tail access is partially related to the simplification of our implementation choices, and can be further mitigated by moving additional computations to dedicated backend threads.

Lock-free rehash. Rehash is an important mechanism to ensure stable performance of growing hash tables. We construct following scenario to evaluate our lock-free rehash operation: in the initial state, the number of loaded keys is 250 millions and the key-bucket ratio is 8; then, we use a YCSB workload with 50% read (θ =1.22) and 50% insertion to simulate the continuous growth of hash tables. Figure 17 shows HotRing' performance over time when rehashes are conducted. In particular, I,T, and S represent the initialization, transition, and splitting phases of rehash, respectively. It can be observed that two consecutive rehash operations help to retain the throughput as data volume continuously grows. The short-term drops during rehash are attributed to the temporary lack of hotspot awareness when the new hash table starts to work.

5   Related Work

Many existing works focus on the design of index structures for key-value stores. Memcached [17] is a widely-used distributed key-value store, which is used by a large number of companies. However, its multi-threading performance is unsatisfactory, due to frequent competition of locks. Based on Memcached, there is plenty of work with outstanding contributions in the literature. By implementing lock-free designs and cache-friendly optimizations, they achieve higher con-
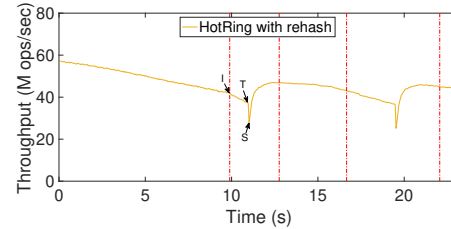
currency and throughput [8, 15, 17, 24, 33, 43]. In particular, FASTER [8] is one of the state-of-the-art implementations with lock-free designs. For hotspot awareness, Splay trees [47] is an inspiring work that adapts its structure to optimize for recently accessed items. However, its lock-based design makes it unsuitable for highly concurrent scenarios.

Besides, there are many works on the better integration of system designs with emerging hardwares, such as FP-GAs [6,34], RDMA-enabled NICs [28,41,45], GPU [22], low-overhead user-level implementations for TCP [25], and InfiniBand with hardware-level reliable datagrams [41]. Meanwhile, in order to provide fast memory allocation (for insertion and deletion), many protocols also leverage lock-free memory management methods [38, 39], which can be used to prevent the ABA problem [48]. Note that these optimization for hardware and memory management are orthogonal to the design of index structures, and we can also adopt these ideas to further improve HotRing's performance.

6   Conclusion and Future Work

In real-world deployment of KVSes, the hotspot issue is common and becomes more serious recently. For example, in order to provide a highly-available service, Alibaba's NoSQL product *Tair* [1] has to allocate more machines than necessary to handle sudden occurrences of hotspots. Hence, we explore opportunities and challenges for designing hotspot-aware in-memory KVS. Based on discovered insights, we propose a hash index called HotRing that is optimized for massively concurrent accesses to a small portion of items. It dynamically adapts to the shift of hotspots by pointing bucket heads to frequently accessed items. In most cases, hot items can be retrieved within two memory accesses. HotRing comprehensively adopts lock-free structures in its design, for both common hash operations and HotRing-specific operations. The extensive experiments show that our approach is able to achieve $2.58\times$ throughput improvement compared to other in-memory KVSes on highly skewed workloads. Now HotRing has become a subcomponent of *Tair*, extensively used in Alibaba Group.

At present, HotRing-r is designed for single hotspot on each chain, while HotRing-s also handles multiple hotspots. In most cases, we can mitigate the multiple hotspot issue by reducing chaining length via rehash. For some extreme cases where it fails to handle, we leave the exploration of a suitable solution as future work.

References

[1] Tair - NoSQL product of Alibaba Group. https://help.aliyun.com/document_detail/145957.html, 2019.

[2] Juan Alemany and Edward W Felten. Performance Issues in Non-blocking Synchronization on Shared-memory Multiprocessors. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing (PODC 1992)*, pages 125–134. ACM, 1992.

[3] Amazon. Amazon ElastiCache. https://aws.amazon.com/cn/elasticache, 2014.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. 40(1):53–64, 2012.

[5] Brian N Bershad. Practical Considerations for Lock-free Concurrent Objects. 1991.

[6] Michaela Blott, Kimon Karras, Ling Liu, Kees A Vissers, Jeremia Bär, and Zsolt István. Achieving 10Gbps Line-rate Key-value Stores with FPGAs. In *HotCloud*, 2013.

[7] Badrish Chandramouli. microsoft FASTER. https://github.com/microsoft/FASTER, 2018.

[8] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD 2018)*, pages 275–290. ACM, 2018.

[9] Yue Cheng, Aayush Gupta, and Ali R Butt. An In-memory Object Caching Framework with Adaptive Load Balancing. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys 2015)*, page 4. ACM, 2015.

[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)*, pages 143–154. ACM, 2010.

[11] Frank Dabek, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP 2001)*, volume 35, pages 202–215. ACM, 2001.

[12] Dhananjay Ragade. David Raccah. Linkedin Memcached. https://www.oracle.com/technetwork/server-storage/ts-4696-159286.pdf, 2014.

[13] Mathieu Desnoyers, Paul E McKenney, Alan S Stern, Michel R Dagenais, and Jonathan Walpole. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[14] Facebook. RocksDB. https://rocksdb.org.

[15] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013)*, volume 13, pages 371–384, 2013.

[16] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC 2011)*, page 23. ACM, 2011.

[17] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux journal*, 2004(124):5, 2004.

[18] Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb, 2011.

[19] Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC 2001)*, pages 300–314. Springer, 2001.

[20] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of HDFS Under HBase: A Book Messages Case Study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 199–212, 2014.

[21] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture (ISCA 1993)*, volume 21, pages 289–300. ACM, 1993.

[22] Tayler H Hetherington, Mike O'Connor, and Tor M Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC 2015)*, pages 43–57. ACM, 2015.

[23] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. Characterizing Load Imbalance in Real-world Networked Caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets 2014)*, page 8. ACM, 2014.

[24] Intel. Intel Threading Building Blocks. https://www.threadingbuildingblocks.org, 2017.

[25] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*, volume 14, pages 489–502, 2014.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing Key-Value Stores with Fast In-Network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017)*, pages 121–136. ACM, 2017.

[27] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proceedings of the 11th international conference on World Wide Web (WWW 2002)*, pages 293–304. ACM, 2002.

[28] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA Efficiently for Key-Value Services. *ACM SIGCOMM 2014 Conference (SIGCOMM 2014)*, 44(4):295–306, 2015.

[29] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on The World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing (STOC 1997)*, volume 97, pages 654–663, 1997.

[30] Eddie Kohler. Masstree. https://github.com/kohler/masstree-beta, 2013.

[31] Redis lab. Redis. https://https://redis.io/, 2017.

[32] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be fast, Cheap and in Control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, 2016.

[33] Hyeontaek Lim, Donsu Han, David G Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*, pages 429–444, 2014.

[34] Kevin Lim, David Meisner, Ali G Saidi, Parthasarathy Ranganathan, and Thomas F Wenisch. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA 2013)*, volume 41, pages 36–47. ACM, 2013.

[35] Greg Linden. Akamai Online Retail Performance Report: Milliseconds are Critical. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html, 2006.

[36] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[37] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the Seventh EuroSys Conference on Computer Systems, (EuroSys 2012)*, pages 183–196. ACM, 2012.

[38] Maged M Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*, pages 73–82. ACM, 2002.

[39] Maged M Michael. Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 21–30. ACM, 2002.

[40] Maged M Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[41] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC 2013)*, pages 103–114, 2013.

[42] 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 USENIX Symposium on Networked Systems Design and Implementation (NSDI 2013)*, volume 13, pages 385–398, 2013.

[43] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*, pages 29–41. ACM, 2011.

[44] Matthew Shafer. Memcached. https://github.com/memcached/memcached, 2012.

[45] Patrick Stuedi, Animesh Trivedi, and Bernard Metzler. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC 2012)*, pages 347–353, 2012.

[46] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[47] Robert Endre Tarjan. Sequential Access in Splay Trees Takes Linear Time. *Combinatorica*, 5(4):367–378, 1985.

[48] R Kent Treiber. *Systems Programming: Coping With Parallelism*. New York: International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[49] Twitter. Twitter Memcached. https://github.com/twitter/twemcache, 2014.

[50] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC 1995)*, volume 95, pages 214–222. Citeseer, 1995.

BCW: Buffer-Controlled Writes to HDDs for SSD-HDD Hybrid Storage Server

Shucheng Wang[1], Ziyi Lu[1], Qiang Cao[1*], Hong Jiang[3],

Jie Yao[2], Yuanyuan Dong[4] and Puyuan Yang[4]

[1]Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System,

[2]School of Computer Science and Technology, Huazhong University of Science and Technology,

[3]Department of Computer Science and Engineering, University of Texas at Arlington,

[4]Alibaba Group

Abstract

Hybrid Storage servers combining high-speed SSDs and high-capacity HDDs are designed for high cost-effectiveness and provide μs-level responsiveness for applications. Observations from the production hybrid cloud storage system Pangu suggest that HDDs are often severely underutilized while SSDs are overused, especially for writes that dominate the hybrid storage. This lopsided utilization between HDDs and SSDs leads to not only fast wear-out in the latter but also very high tail latency due to frequent garbage collections induced by intensive writes to the latter. On the other hand, our extensive experimental study reveals that a series of sequential and continuous writes to HDDs exhibit a periodic, staircase shaped pattern of write latency, i.e., low (e.g., 35μs), middle (e.g., 55μs), and high latency (e.g., 12ms), resulting from buffered writes in HDD's controller. This suggests that HDDs can potentially provide μs-level write IO delay (for appropriately scheduled writes), which is close to SSDs' write performance. These observations inspire us to effectively exploit this performance potential of HDDs to absorb as many writes as possible to avoid SSD overuse without performance degradation.

To achieve this goal, we first characterize performance behaviors of hybrid storage in general and its HDDs in particular. Based on the findings on sequential and continuous writes, we propose a prediction model to accurately determine next write latency state (i.e., fast, middle and slow). With this model, a Buffer-Controlled Write approach, BCW, is proposed to proactively and effectively control buffered writes so that low- and mid-latency periods in HDDs are scheduled with application write data and high-latency periods are filled with padded data. Based on BCW, we design a mixed IO scheduler (MIOS) to adaptively steer incoming data to SSDs and HDDs according to write patterns, runtime queue lengths, and disk status. We perform extensive evaluations under production workloads and benchmarks. The results show that MIOS removes up to 93% amount of data written to SSDs, reduces
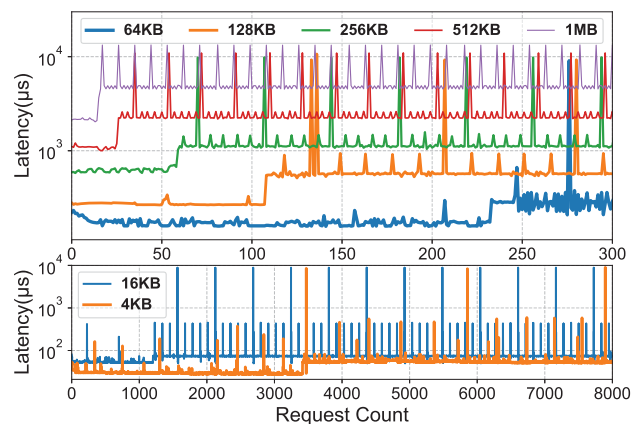
Figure 1: Sequential writing in a 10TB Western Digital HDD.

average and 99th-percentile latencies of the hybrid server by 65% and 85% respectively.

1 Introduction

Storage clouds have prevalently deployed hybrid storage servers integrating solid-state drives (SSDs) and hard-drive disks (HDDs) in their underlying uniform storage infrastructure, such as Alibaba Pangu [9], Amazon [38], Facebook [35], Google [20], Microsoft Azure [8]. Such hybrid storage servers employ an SSD-HDD tiered architecture to reap the benefits of both SSDs and HDDs for their superior IO performance and large capacity respectively, thus achieving high cost-effectiveness. Incoming writes are quickly persisted in the SSD tier and acknowledged, and then flushed to the HDD tier at a later time.

Our observations from real-world production workloads of hybrid storage servers in Pangu indicate that, SSDs are generally over-used while HDDs are less than 10% utilized on average, missing the opportunity to exploit HDDs' performance and capacity potentials. However, writes are known to be unfriendly to SSDs for two reasons. First, SSDs have limited P/E (Program/Erase) cycles [6, 36] that are directly

---

related to the amount of writes. Second, SSDs suffer from un-predictable, severe performance degradations resulting from garbage collections [26, 51]. To guarantee stable write per-formance of storage servers in write-heavy workloads, cloud providers have to deploy more and/or larger SSDs, signifi-cantly increasing their total investment capital.

Our extensive experimental study on HDD write behav-iors, conducted on various HDD products and with results shown in Figure 1, suggests that a series of continuous and sequential small HDD writes (e.g., 4KB) exhibit low latency (e.g., 35μs) for about 60 ms, and then a sharply elevated high latency (e.g., 12ms), which is followed by medium latency (e.g., 55μs) for about 40ms. The three states of write behav-iors, or *write states* in short, are referred to in this paper as *fast, mid*, and *slow* writes, respectively. The former two types of writes can provide μs-level responsiveness, because incom-ing writes are considered complete and acknowledged (to the host) once their data have been written into the built-in buffer in the controller. However, when the write buffer is full, host writes have to be blocked until the buffered data are flushed into the disk, causing slow writes. This finding inspires us to fully exploit performance potentials offered by buffered writes of HDD, improving the performance while mitigating write-penalty on SSDs. Our goal is to enable hybrid storage servers to achieve higher performance and reliability without introducing extra hardware.

However, the key challenge for adopting buffered writes in HDDs to take advantage of the fast and mid writes is the dif-ficulty in predicting precisely when these write states would occur. The internal buffer and other components of HDDs are completely hidden from the host. Host can only identify the current write state according to its own delay, but not future write states. To address this issue, we build a prediction model for sequential and continuous write patterns that predicts the next HDD write state. The insights is that, the write states of continuous and sequential HDD write requests is periodical. The prediction of next write state can be achieved with the information of buffered-write period and current write state. Then, we propose a Buffer-Controlled Write (BCW) approach. BCW can proactively and effectively control the buffer write behavior according to the predictor and runtime IO monitor-ing. Besides, BCW also actively "skip" slow writes by filling padded data during HDD slow writes.

We further propose a mixed IO scheduler (MIOS) for SSD-HDD hybrid storage by leveraging the BCW approach. MIOS adaptively redirects incoming writes to SSDs or HDDs de-pending on write states, runtime queue length, and disk status. Under high IO intensity, MIOS can be triggered to reduce IO pressure, the amount of data written and write penalty on SSDs while improving both average and tail latency.

The main contributions of this paper are as follows.

- Through extensive experimental studies on HDD write behaviors, we discover that there exist a periodic staircase-shaped write latency pattern consisting of μs-

level write latency (low and mid write states) followed by ms-level write latency (slow write state) upon con-tinuous and sequential writes, because of the buffered write feature in HDDs. To facilitate the full exploita-tion of this write latency pattern, we build a predictor to pre-determine what the next write state is.

- We propose a buffer-controlled write (BCW) approach, which proactively activates continuous and sequential write patterns, as well as effectively controls the IO be-havior, according to the predictor and runtime IO mon-itoring. BCW also employs data padding to actively avoid, or skips, slow writes for the host.

- We design an SSD-HDD mixed IO scheduler (MIOS) to improve the overall performance of SSD-HDD hybrid storage servers, while substantially reducing write traffic to SSDs.

- We prototype and evaluate MIOS under a variety of pro-duction workloads. The results demonstrate that MIOS reduces average and tail latency significantly with dra-matic decrease in the amount of data written to SSDs.

The rest of the paper is organized as follows. Section 2 provides the necessary background for the proposed BCW approach. Section 3 analyzes the behaviors of HDD buffered writes. Section 4 describes design and implementation of BCW and MIOS. We evaluate the effectiveness of MIOS in Section 5. Finally, Section 6 describes related works and Section 7 concludes the paper.

2 Background and Motivation

2.1 Primary Storage

Nowadays, primary storage involves popular solid-state driver (SSD), and traditional hard disk drive (HDD). SSDs have become a mainstream storage media due to its superior performance to and lower power consumption than HDDs [1, 49]. However, the limited write endurance has become a critical design issue in SSD-based storage systems [34]. Fur-thermore, SSDs suffer from performance-degrading garbage collections (GCs), which recycle the invalid pages by moving valid parts to new blocks and then erasing old blocks [27, 37]. GCs with ms-level delays can block incoming user requests, thus leading to long tail latency [17]. On the other hand, both large IO blocks and high IO intensity can lead to sudden increase in SSD queue, resulting in high tail latency [26]. Therefore, recent studies [50] indicate that SSDs do not al-ways exhibit their ideal performance in practical.

HDDs have large capacity at low cost without the wear-out problem. However, HDDs have relatively low performance compared to SSDs. A random HDD IO has 2∼3 orders of magnitude higher latency than an SSD IO. This is primarily because of the ms-level mechanical seeking of disk head.

2.2 SSD-HDD Hybrid Storage

To accommodate exponentially increasing storage require-
ment while achieving overall cost-effectiveness, SSD-HDD
hybrid storage has emerged to be an inevitable choice for
cloud providers [40,47]. Most providers, such as Google [20],
Amazon [38], Facebook [35], and Microsoft's online ser-
vices [8], expect larger storage capacity and better perfor-
mance but at lower cost. To meet this demand, they increas-
ingly embrace storage heterogeneity, by deploying variable
types and numbers of SSDs, which offer lower IO latency [14],
as the primary tier and HDDs, which provide larger capacity
at low cost as the secondary tier. The fast SSD tier generally
plays the role of a write buffer to quickly persist incoming
write data, which are eventually flushed to the slower but
larger HDD tier. As a result, the SSD tier absorbs most of the
write traffic from foreground applications.

2.3 Write Behavior of Hybrid Storage

Write-intensive workloads widely exist in many production
environments, such as enterprise applications, supercomput-
ing, and clouds. Enterprise servers are expected to rapidly
persist production data in time, such as business databases.
Burst buffer [3,28] in supercomputing systems also deploy
high-performance SSDs to temporarily store instantaneous
highly-intensive write data.

More commonly, many backend storage servers in cloud
must accommodate write-dominated workloads, as observed
in Alibaba Pangu [9]. Pangu is a distributed large-scale stor-
age platform and provides cost-effective and unified storage
services for Alibaba Cloud [22,30] and Ant Financial [9]. As
such, Pangu needs to minimize the total cost of ownership
while meeting strict QoS requirements like tail latency [5,15].

As an observation made through our analysis of produc-
tion trace data from Pangu in Table 1, some storage nodes
(servers) in Pangu rarely serve reads from the frontend and
instead must handle amounts of highly-intensive writes. For
Alibaba Cloud, because the upper-level latency-critical online
services generally build their own application-aware caches to
ensure service responsiveness and reserve local fast-storage
to cache hot data for user reads, the backend storage nodes
are thus required to persist new and updated data from fron-
tend nodes as soon as possible. To better understand this
write-dominated workload behavior, we analyze four typical
workloads on Pangu storage nodes A (Cloud Computing), B
(Cloud Storage), C and D (Structured Storage). We count the
workloads from one SSD and one HDD in each node because
the workload behavior of all storage devices is basically the
same in one node. Observations are drawn as follows. A com-
prehensive workload analysis of Pangu can be found in the
previous study [31].

- Most requests are writes. As shown in Table 1, more than
  77% and up to 95% of requests are writes in these nodes,
  and the amount of data written is 1-2 orders of magnitude

Table 1: The workload characteristics of Pangu traces
recorded from one SSD and one HDD in four different nodes,
A B C and D, that support online services.

Node Type	Duration (min)	Writes (GB)	Reads (GB)	Avg. Req. Size(KB)	Peak KRPS	Avg. KRPS	Avg. HDD IO Uti.(%)	Avg. SSD IO Uti.(%)
A	45	18.5	1.4	56.0	3.4	0.23	7.6	11.9
B	30	74.4	2	17.7	9.3	2.5	9.8	28.5
C	30	10.7	2.1	4.2	9.6	2.7	4.1	24.6
D	26	10.1	1.7	4.1	11.1	3	4.8	25

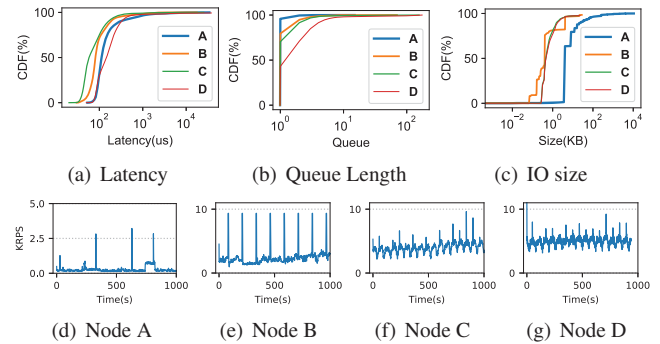| (a) Latency	(b) Queue Length	(c) IO size
(d) Node A	(e) Node B	(f) Node C	(g) Node D

Figure 2: Behaviors of production workloads on four repre-
sentative hybrid storage nodes in Pangu in terms of latency,
queue length, request size and IO intensity.

larger than that of data read from them. Actually, nearly 3
TB data are written to every SSD each day, which is close
to DWPD (Drive Writes Per Day) that strictly limits the
amount of SSD data written daily for reliability.

- The IO intensity distribution has bursty patterns. As
  shown in Figure 2(d) through Figure 2(g), SSDs experi-
  ence bursty intensive write workloads (e.g., 11K request
  per second in workload D).

- The amount of data written to SSDs and HDDs differ
  dramatically. For instance, the average SSD IO utiliza-
  tion is up to 28.5% in load B while it is less than 10% in
  HDD. Even so, most of the HDD utilization is used in
  dumping SSD data, rarely servicing user requests.

- There exists long tail latency. As shown in Figure 2(a),
  SSDs with high IOPS suffer from heavy-tail IO latency
  (e.g., the 99th percentile latency is 10ms) due to queue
  blocking in Figure 2(b). This is caused in part by (1)
  large writes (e.g., 1MB), and (2) frequent garbage col-
  lections induced by high write intensity.

- Small size IOs account for a large proportion of all IOs.
  As shown in Figure 2(c), more than 75% of write re-
  quests are of 10KB or smaller, and the average request
  size is nearly 4KB in C and D.

2.4 Challenge

To relieve the SSD pressure from write-dominate work-
loads, a simple solution is to increase the number of SSDs
in the hybrid nodes. However, this is a costly solution as it
increases the total cost of ownership. An alternative is to ex-
ploit the severely underutilized HDD IO capacity in hybrid
storage nodes when SSDs are overused. The state-of-art so-
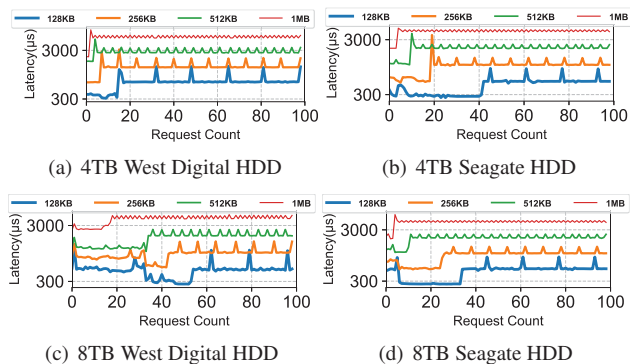lution SSD-Write-Redirect (SWR) [31] redirects large SSD

Figure 3: Sequential writing on four types of HDDs.

writes to idle HDDs. This approach can alleviate the SSD queue blocking issue to some extent. However, the IO delays experienced by requests redirected to HDDs are shown to be 3-12 times higher than those experienced on SSDs. This is clearly undesirable, if not unacceptable, for most small writes that demand μs-level latency. The key challenge is how to reduce HDD IO delay to the μs-level that is close to SSDs, a seemingly impossible task at a first glance. Fortunately, as we look closer into the write behaviors of HDDs, this is indeed a possible task, which we elaborate next.

3 HDD Write Behaviors

To have a comprehensive understanding HDD write behaviors, so as to assess the possibility of achieving μs-level write latency on HDDs, we perform continuous and sequential writes, which is the most friendly write pattern for HDDs.

3.1 Buffered Writes

We conduct a "*Profiling*" process to observe detailed HDD behaviors, a series of continuous and sequential writes with the same IO size are written to an HDD. We select five representative HDD products: West Digital 10TB (WD100EFAX [13]), 8TB (WD8004FRYZ [12]), 4TB (WD40EZRZ [13]), Seagate 8TB (ST8000DM0004 [44]), 4TB (ST4000DM004 [45]). The 4TB Seagate HDD is SMR (Shingled Magnetic Recording) and the other four HDDs are PMR (Perpendicular Magnetic Recording). We draw three interestng observations from the profiling results shown in Figure 1 and Figure 3.

- For each tested HDD, the series of continuous sequential write requests experience a similar sequence of three-level write latency, i.e., low, mid, and high latencies, forming a staircase-shaped time series. For example, in the 10TB HDD, the three levels of write latency of 16KB writes are about 66μs, 135μs, and 12ms respectively.

- The observed HDD write behavior is periodic. At the beginning (right after the buffer becomes empty) low-latency writes last for a period (e.g., 60ms in 10TB), which is followed by a spike (high-latency writes) and then mid-latency writes. If the write pattern is contin-
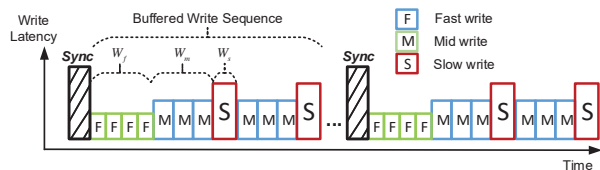

Figure 4: The HDD Buffered-Write Model with two complete Buffered Write Sequences.

uous, high-latency writes and mid-latency writes will appear alternately.

- The number of low-latency continuous writes in a sequence relies on their I/O size. Smaller write size leads to a larger number of writes. For example, the number of 16KB and 64KB writes is about 1200 and 240 on the 10TB HDD, respectively.

The reasons behind these observed HDD write behaviors are as follows. Modern HDDs deploy a built-in DRAM (e.g., 256MB for the 10TB and 8TB HDDs, and 64MB for the two 4TB HDDs). However, only a part of the DRAM (e.g., 16MB for 10TB WD and 8TB Seagate HDD, 4MB for 8TB WD HDD and 4TB Seagate HDD, 2MB for 4TB WD HDD) can be used to buffer incoming write IOs based on external observation. The remaining capacity of the HDD built-in DRAM can be used as read-ahead cache, ECC buffer [10], sector remapping buffer, or prefetching buffer [18,43]. However, the exact mechanism by which this built-in DRAM in HDD is used, which varies with the HDD model, is generally proprietary to the HDD manufactures only. Fortunately, the actual size of write buffer can be measured externally by profiling.

Upon successful buffering of a write, HDD immediately informs the host of request completion. When the buffered data exceed a threshold, the HDD must force a flushing of the buffered data into their locations in the disk media. During this period, incoming writes must be blocked until the buffer is freed up again. It is worth noting that, after an idle period, the HDD buffer may become empty implicitly as a course of flushing data to the disk. However, to explicitly empty the buffer, we can actively invoke *sync*() to force flushing.

3.2 An HDD Buffered-Write Model

To formally characterize the HDD write behavior, we build an HDD buffered-write model. Figure 4 illustrates the schematic diagram of the HDD buffered-write model in the time dimension (runtime). The x axis represents the time sequences of transitions among the three write levels, with each sequence being started by "Sync" event.

A Buffered-Write Sequence consists of three aforementioned types of HDD buffered writes, i.e., Fast (low-latency), Mid (mid-latency) and Slow (high-latency) writes, which denotes as F, M, and S, respectively. In the model, F, M, and S can be thought of as the states a write request can be in (i.e., experiencing the fast, mid or slow write process). As described in Table 2, these IO delays are denoted as L_f, L_m and L_s, respectively. The F state means that an incoming write

Table 2: The list of descriptions about all the parameters in the HDD Buffered-Write Model.

Parameters	Description
$L_{f/m/s}$	The IO delays of write requests in the F/M/S write states
$W_{f/m/s}$	The cumulative amount of written data for the Fast/Mid/Slow Stages
$T_{f/m/s}$	The time duration of the Fast/Mid/Slow Stages
s_i	The IO size of write request i

request w_i with the size of s_i can be buffered completely in the built-in DRAM buffer of HDD. The M state means that the write buffer is close to be full. The S state means that the write buffer is full and any incoming write request is blocked.

A Buffered Write Sequence lasts a Fast stage, followed by one or more Slow-and-Mid stage-pairs. The sequence begins when there is sufficient buffer available for Fast stage (e.g., close to empty). It ends when current series of continuous writes ends. The Fast, Mid, and Slow Stage last for T_f, T_m, and T_s respectively, which are determined by the cumulative amount of written data W_f, W_m, and W_s in the respective states. Actually, $W_f = T_f * s_i / L_f$ and it is applied to W_m.

We can profile the HDDs to identify such key parameters. For example, in the 10TB HDD with 64KB write requests shown in Figure 1, the value of L_f is 180μs, L_m is 280μs and L_s is 12ms. The value of T_f is 60 ms, T_m is 37ms and T_s is 12ms. W_f is 16MB and W_m is 8MB. W_s depends on the IO size s_i. According to the HDD buffered-write model, the Fast and Mid writes of HDD have 100-μs-level latency, which can approach the write latency of SSDs. This motivates us to design a controllable buffer write strategy for HDDs to reduce writes on SSDs in hybrid storage systems without sacrificing the overall performance.

Note that the buffering and flushing mechanisms are completely hidden from the host and heavily depend on the specific HDD models. Fortunately, we can measure the buffered write feature of an HDD externally and experimentally based on the aforementioned experiments.

4 Design

To fully exploit HDD buffered writes, two critical challenges must be addressed. The first is how to determine which write state that a write request will be Fast (F), Mid (M), or Slow (S), in order to properly schedule the write request. The second is how to steer an incoming write request to HDD without performance degradation.

For the first problem, we build a *Write-state Predictor* to pre-determine the next write state based on the current write state and buffer state. The ability to determine the subsequent write state of HDD is critical to scheduling incoming write requests. Based on that, we propose *Buffer-Controlled Writes*, shortly for **BCW**, a writing approach to proactively activate continuous and sequential write patterns that the predictor relied on, as well as effectively controls the IO behavior according to the predictor and runtime IO monitoring. To avoid performance degradation caused by S writes, we propose a proactive padding-write approach to "skip" the S state by
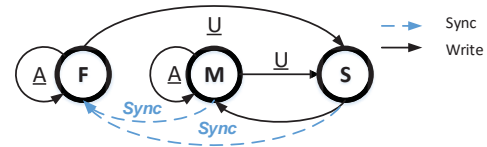

Figure 5: The State Predication Diagram. Each write request can only be one of the three write states, F, M, and S. Letter "A" means that the current data written in the F and M states are less than the W_f and W_m values, respectively. Otherwise, the write buffer is "U". The *Sync* operation takes the next write state back to F.

executing slow writes with padded non-user data.

To overcome the second issue, we propose the SSD-HDD Mixed IO scheduler (**MIOS**) that adaptively controls queue lengths of SSDs and HDDs in hybrid storage nodes, and determines where to steer a user write request.

4.1 Write-state Predictor

The next write state could be predicted according to write buffer's free space and the write state of the current request. In HDD buffered write, each write request state should be one of F, M, and S state. The HDD write buffer state is considered by buffered-write model to be in either A (available) or U (unavailable). The "A" state means that the current Accumulative Data Written (**ADW**) in the F and M states are less than W_f and W_m, respectively. Otherwise, the write buffer is in the "U" state. Figure 5 shows how the next write state is determined based on the current buffer state and write state in a State Predication Diagram, which is described as follows:

- F/A : The current write state is F and the buffer is available. Next write state is most likely to be F.
- F/U : Although the current write state is F, the buffer is unavailable. Next write state is likely to change to S.
- M/A : The current write state is M and the buffer is available. Next write state is most likely to remain M.
- M/U : Although the current write state is M, the buffer is unavailable. Next write state should be S.
- S : The current write state is S. Next write state will be M with a high probability.
- The *Sync* operation will force the next write state and buffer state back to be F/A in all cases.

Based on that, we design a Write-state Predictor described in Algorithm 1. It identifies what the current write state is, F, M or S, by monitoring the IO request size and latency, and calculating the free space in the write buffer. That is, the ADW in the current write state (F or M) is recorded and compared with W_f or W_m, for predicting the next write state.

Next, we assess the prediction accuracy of the write-state predictor. We write 100GB data with an IO size of 128KB to the 10TB WD HDD and invoke *sync()* after each 1GB data written. The results show that the predictor correctly identifies 99.5% of the F state, 98.1% of the M state and 60.3% of the S state. The low prediction accuracy for the S write-state is

Algorithm 1 The algorithm of Write-state Predictor

Input: Current write request size: *size*; The last write state: *state*; Current accumulative amount of data written: *ADW*;
The amounts of data written in the *F* state and *M* state: W_F and W_M
Output: Write-state prediction for the next request (*F*, *M* or *S*)

```
1:  function Predictor()
2:  if state == F then
3:      if (ADW + size) < W_f then : return F
4:      else: return S
5:      end if
6:  else if  state == M then
7:      if (ADW + size) < W_m then : return M
8:      else: return S
9:      end if
10: else: return M
11: end if
```

Algorithm 2 The algorithm of Buffer-Controlled Write

Input: The max loop of Buffered Write Sequence: $Loop_{max}$
Request R_i size: $size_i$; Current written amount: *ADW*;
The state of last write: *state*;
Active padded writes and their size: *PS*,*PF* and $size_{PS}$,$size_{PF}$

```
1:  sync()
2:  while loop < Loop_max do
3:      if  request R_i in the HDD write queue  then
4:          write R_i to HDD, update ADW and state
5:      else
6:          if Predictor() == S then
7:              flag_HDD = False    // Stop receiving
8:              while state == S do
9:                  write PS to HDD, update ADW and state
10:             end while
11:             flag_HDD = True    // Start receiving
12:             reset ADW; loop++
13:         end if
14:         if Predictor() == M then
15:             write PF to HDD; update ADW and state
16:         end if
17:     end if
18: end while
```

due to the prediction policy that tends to favor the *S* write-state to reduce the performance degradation, when an actual *S* write-state is mis-predicted as a different state.

4.2 Buffer-Controlled Writes

Buffer-Controlled Writes (BCW) is an HDD writing approach that ensures user writes using *F* or *M* write state, and avoids allocating Slow writes. The key idea of BCW is to make buffered write controllable. Based on the Write-state predictor, we design BCW, as described in Algorithm 2.

Upon activating BCW, a *sync()* operation is invoked to force synchronization to empty the buffer actively. BCW dispatches sequential user writes to HDD if it is predicted to be in *F* or *M* state, otherwise pads non-user data to HDD, until it reach to the max setting loop (or unlimited) of Buffered Write Sequence. If there are user requests in the queue, BCW writes them serially. After a write is completed, BCW adds its write size to ADW, and updates the write-state accordingly.

During light or idle workload periods with sparse requests, the HDD request queue will be empty from time to time, making the write stream discontinuous. To ensure the stability and certainty of buffered writes in a sequential and continuous pattern, BCW will proactively pad non-user data to write to the disk. The padding data are of two types, *PF* and *PS*. The former is used to fill the *F* and *M* states with 4KB non-user data; the latter is to fill the *S* state with larger block size, e.g., 64KB of non-user data. A small *PF* can minimize the waiting time of user requests. A large *PS* helps trigger Slow write quickly. Note that even for each padded write, BCW still executes the write-state predictor algorithm.

More specifically, BCW continuously calculates ADW in the current state (*F* or *M*). When ADW is close to W_f or W_m, it means that the HDD buffered write is at the tail of the Fast or Mid stage. The *S* write state may occur after several writes. At this point, BCW notifies the scheduler and proactively triggers the Slow write with *PS*.

To avoid long latency for user writes, at this period, all incoming user requests have to be steered to other storage devices, such as SSDs. When an *S* write completed, the next

write will be *M* according to the write-state predictor. Then BCW resets the ADW and accepts user requests again.

We also find it unnecessary to proactively fill padded writes in the Fast state before ADW exceeds W_f. When ADW does not reach W_f, the physical disk operation is not yet triggered and the buffer can absorb user requests for this period. When ADW exceeds W_f in a short time of period, it means that the buffer will begin to flush the data to the disk and the next write state will be changed to *S*. On the other hand, when ADW is less than W_f for a long time of period, the disk can flush the buffered data automatically so that the next write state may be *F*. However, it does not affect performance. We apply this observation to the scheduler design in the next section.

In most cases, the sequential and continuous write pattern induced by BCW is reasonably stable. However, this pattern can be broken, e.g., HDD reads. Besides, slow writes may be triggered in advance by user requests or *PF* writes. To regain the control of buffered writes, BCW continuously executes *PS* until a *S* write state is detected. As a result, the write-state predictor will be recalibrated. The cost of this strategy is that BCW wastes IO and storage of HDD to perform *PS* writes. In addition, we also can issue *sync()* to reset the buffered write state in BCW. Howerver, a *sync()* can take several hundred milliseconds, during which the HDD cannot accept any writes. Fortunately, in the experiment, we found that the BCW interrupted cases are rare.

BCW stores incoming data to HDD in log-append manner. This differs with traditional logging mode in existing file systems like ext4 [33]. The latter allocates writes to the tail logical address of the log, ensuring address continuity. However, it doesn't ensure IO continuity and does not determine the next write state. In contrast, BCW maintains both address and IO continuity, make the buffer writing be controllable.
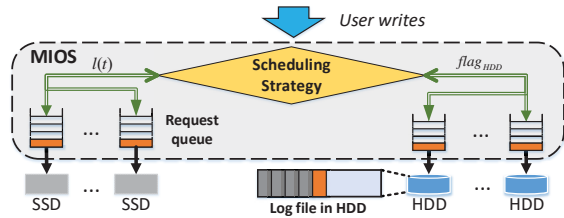
Figure 6: Architecture of the Mixed IO scheduler. It monitors all request queues of SSDs and HDDs. The user writes meeting the conditions are redirected to appropriate HDDs.

4.3 Mixed IO scheduler

BCW provides a proactive and controllable buffer writing approach. In this section, we further propose a *Mixed IO scheduler* (**MIOS**) for SSD-HDD hybrid storage to leverage BCW effectively. The scheduler decides whether or not to steer user writes to a HDD request queue according to the results of the Write-state Predictor and current queue status.

Architecture The architecture of MIOS is shown in Figure 6. MIOS monitors all request queues of SSDs and HDDs at runtime, judiciously triggers the BCW process, and determines whether a user write should be directed to a selected HDD or SSD. MIOS creates a device file in each HDD in the configuration process. The device file stores BCW writes in an append-only manner. Before MIOS scheduling, a Profiling is performed to determine the key parameters (W_f, W_m, etc.) for the write-state predictor.

Scheduling Strategy In algorithm 3, the SSD request queue length $l(t)$ at time t is a key parameter in MIOS. When $l(t)$ is larger than a predefined threshold L, the scheduler steers user writes to an HDD with the prediction of it being F or M write state. The threshold L is pre-determined according to the actual performance measurements on SSD. Specifically, we measure the write latency under different SSD queue lengths. If the request with queue length l has latency larger than the IO delay of HDD in the M state, we simply set the threshold L to the minimum l. The rationale is that when the SSD queue length is larger than L, the SSD writes' latencies will be at the same level as their latencies on an HDD in the F or M write state with BCW. L can be determined and adjusted experimentally according to workload behaviors and storage device configurations at runtime. This strategy mitigates, though not avoids, the long-tail latency upon workload bursts or heavy Garbage Collections on SSD [37, 48]. In these cases, the SSD request queue length can be 8-10 times longer than its average. Therefore, redirected HDD writes not only relieve SSD pressure imposed by bursty requests and heavy GCs, curbing the long-tail latency, but also lower the average latency.

Additionally, when the queue length of SSD is less than L, triggering BCW is optional. Enabling or disabling BCW in this case is denoted as **MIOS_E** or **MIOS_D**, respectively. In other words, *MIOS_E* strategy allows redirection with BCW when the queue length of SSD is lower than L. *MIOS_D* strat-

---

Algorithm 3 The algorithm of Mixed IO Scheduler

Input: SSD queue length at time t: $l(t)$;
Queue length threshold: L; HDD available flag: $flag_{HDD}$;
Schedule Strategy: *MIOS_D* or *MIOS_E*

1: **if** ($flag_{HDD}$ == True) **then**
2:     **if** $l(t) > L$ && $Predictor()$ == F or M **then**
3:         Send to HDD queue
4:     **else if** *MIOS_E* && $Predictor()$ == F **then**
5:         Send to HDD queue
6:     **else:** Send to SSD queue
7:     **end if**
8: **else:** Send to SSD queue
9: **end if**

---

egy, by contrast, disables redirection when the SSD queue length is lower than L. Note that the write latency of an HDD in the M write state is still much higher than that of an SSD. The request latency after redirection may be increased. Therefore, when the $l(t)$ is lower than L, we only redirect user requests to leverage the F write state of HDD in *MIOS_E*. We will experimentally analyze the positive and negative effects of *MIOS_D* and *MIOS_E* in Section 5.

Generally, a typical hybrid storage node contains multiple SSDs and HDDs. We divide all disks into independent SSD/HDD pairs, each of which contains an SSD and one or more HDDs. Each SSD/HDD pair is managed by an independent MIOS scheduler instance.

Finally, MIOS requires the complete control over HDDs. It means that the HDDs in BCW cannot be interfered by other IO operations. When an HDD is executing BCW and a read request arrives, **MIOS** immediately suspends BCW and serves this read. It will try to redirect all writes to other idle disks at this time. For read-dominated workloads, BCW can be disabled to avoid interfering with reads.

4.3.1 Implementation

MIOS can be implemented in either file-level or volume-level to jointly manage SSDs and HDDs in a hybrid storage. In this work, MIOS provides a simple yet flexible file-level request scheduling scheme atop of existing file systems to leverage their mature file-to-storage mapping mechanism. A user request is identified with the corresponding filename and file internal offset. To reduce overhead of the underlying file system, MIOS employs direct IO mode to access the log by calling Linux kernel functions such as *open*, *close*, *read*, and *write*. Data of each write request is stored as a chunk in the log. We design a metadata structure that records and tracks chunks in the log. We choose a hash table and use the file ID field of a request as the hash key.

When an HDD is idle, all user data stored in the log will be written to their own original files, after which the log will be recycled, called as HDD Garbage Collection. HDD GC should be triggered when the log size exceeds a predefined threshold (e.g., 70% capacity of HDD). HDD GC first sequentially and continuously reads user data chunks that are interspersed with

Table 3: The amount of redirected writes data and requests with the *MIOS_D* and the *MIOS_E* strategies.

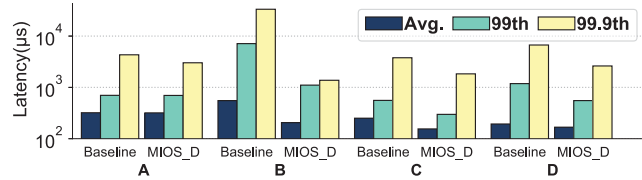| Workload Type	A	B	C	D
Writing Method	Baseline / MIOS_D / MIOS_E			
SSD Writes (GB)	14.7 / 13.9 / 1.2	61.2 / 57.1 / 48.1	7.2 / 6.1 / 2.1	7.5 / 6.3 / 2.1
HDD Writes (GB)	- / 4.1 / 61.6	- / 18.4 / 56	- / 4.5 / 22.3	- / 4.4 / 25.6
SSD Requests (millions)	0.43 / 0.36 / 0.04	4.4 / 3.7 / 1.3	4.8 / 3.7 / 1.6	4.7 / 3.8 / 1.3

Figure 7: The average, 99th and 99.9^{th}-percentile latency under four Pangu production workloads, comparing *Baseline* with *MIOS_D* (a logscale is used for the y-axis).

padded data in the log to reduce seeks. And then it extracts and merges the user data to update their correspond files. These file updates can be performed in batch [53].

5 Evaluation

5.1 Experiment Setup

We run experiments for performance evaluation on a server with two Intel Xeon E5-2696 v4 processors (2.20 GHz, 22 CPUs) and 128 GB of DDR3 DRAM. To understand the impact of different storage configurations on the performance, we choose two types of SSDs, a 256GB Intel 660p SSD [11] and a 256GB Samsung 960EVO SSD [16]. Their peak write throughputs are 0.6 GB/s and 1.5GB/s, respectively. Three types of HDDs are WD 10TB, WD 4TB, and Seagate 4TB, as described in Section 3.1.

The 10TB WD HDD has a W_f of 16MB and W_m of 8MB. Using the process to pre-determine the queue length threshold L explained in Section 4, we set L to 1 for workload of node A, 3 for node B, 2 for node C and D, where the workloads of nodes A, B, C and D are described in Table 1 of Section 2. As discussed earlier, MIOS has two schemes, **MIOS_D** and **MIOS_E**. When the SSD queue length is less than L, the former conservatively disables request redirection; the latter allows request redirection but only redirects user write requests to the F write state. The **Baseline** for the evaluation is writing all user data into the SSDs. In addition, a complete BCW sequence consists a series of 1 Fast stage and 10 Mid/Slow stage-pairs (Figure 4).

5.2 MIOS under Production Workloads

We first evaluate the effectiveness of *MIOS_D* under four Pangu production workloads on the WD 10TB HDD.

Write Performance Figure 7 shows that the average and tail latency (99th and 99.9^{th}) of all four workloads are significantly reduced by *MIOS_D*. Among four workloads, B gains the most benefit. Its average, 99th and 99.9^{th}-precentile
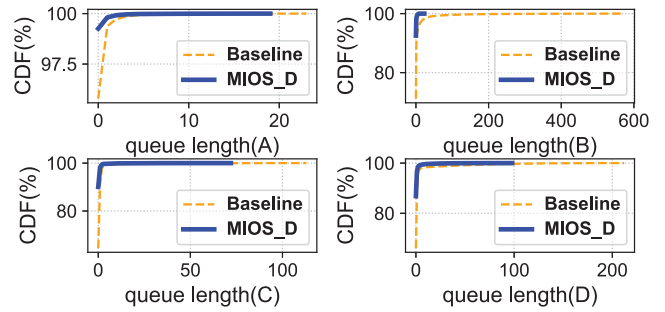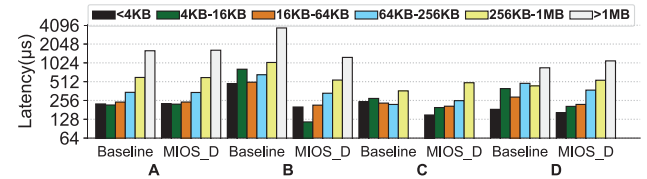

Figure 8: The CDF of SSD queue length.

Figure 9: The average request latency in six request-size groups that are classified by IO size with *MIOS_D*.

latencies are reduced by 65%, 85%, and 95% respectively. On the contrary, these three latencies in A are only reduced by about 2%, 3.5% and 30%, respectively, which is far less than the other workloads. The reason is that the redirection in *MIOS_D* is only triggered when the queue length is high, but A has the least intensity and thus the least queue blocking, which renders MIOS much less useful.

To better understand the root causes for the above experimental results, the cumulative distribution functions (CDFs) of SSD queue lengths for four workloads are shown in Figure 8. *MIOS_D* significantly shorten queue lengths compared to *Baseline*. B and A have the maximum (95%) and minimum (15%) reduction in their queue lengths. Therefore, *MIOS_D* reduces the overall queueing delay significantly.

Request size To deeply understand impact of write size in *MIOS_D* and BCW, we break down all redirected requests into six groups with different ranges of IO sizes, and measure the average latency in each group.

Figure 9 shows that, in all four workloads, *MIOS_D* reduces the average write latency of size below 64KB. The B workload benefits the most. The average latencies of three groups of small-sized requests (<4KB; 4KB-16KB; 16KB-64KB) are reduced by 61%, 85%, and 59%, respectively. The other three workloads also reduce their latencies differently. In *Baseline*, small and intensive requests result in queue blocking more frequently (Figure 2) than in *MIOS_D*. Therefore, *MIOS_D* is the most effective in reducing latency in such cases.

However, in groups of requests larger than 256KB, the average latency is increased in all workloads except B. The latency is increased by up to 31.7% in the >1MB group, and 12.1% in the 256KB-1MB group for the D workload. The average latency of the 256KB-1MB group in C is increased by 20.1%. The reason is twofold. First, large SSD writes under light load have better performance than HDDs because
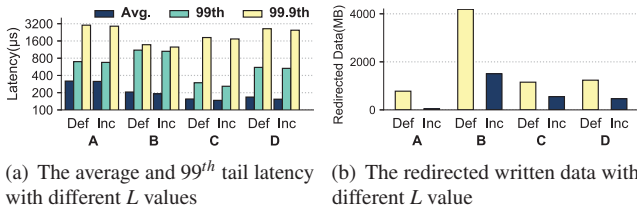
(a) The average and 99th tail latency with different L values

(b) The redirected written data with different L value

Figure 10: *MIOS_D* with different queue length threshold *L*.

(a) The average and tail latency with *MIOS_D* and *MIOS_E*

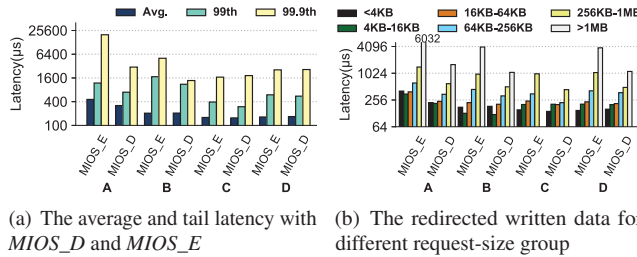(b) The redirected written data for different request-size group

Figure 11: *MIOS_D* vs *MIOS_E*.

SSDs have high internal-parallelism that favors large requests. Second, large writes are relatively sparse and not easy to be completely blocked. For example, the average latency of the >256KB request-size groups in *Baseline* is very close to the raw SSD write performance.

Queue Length Threshold *L* To evaluate the effect of *L* selection, we compare the pre-defined *L* value (**Def**), determined by the process described Section 4.2, with $L+1$ (**Inc**). Note that the process for pre-defining the queue length threshold is designed to tradeoff between reducing the write latency and reducing the write traffic to SSD.

Figure 10(a) shows that, *Inc* slightly reduces average, 99th and 99.9^{th}-percentile latency compared to *Def*. Among the four workloads, the maximum reduction in average latency is less than 10%. This is because the higher queue length is, the longer waiting delay a request experiences. Therefore, *Inc* can acquire more latency gains by redirection than *Def*. However, the choice of *L* value can greatly affect the amount of redirected data. In Figure 10(b), the number of redirected requests is much smaller in *Inc* than in *Def*. The amount of redirected data for workloads A∼D are decreased by 94%, 64%, 52% and 62%, respectively. These results are consistent with the implications of Figure 8 that longer queue length in SSD triggers much fewer SSD overuse alerts, significantly reducing chances for request redirecting to HDD.

MIOS_D vs MIOS_E We compare *MIOS_D* with *MIOS_E* in terms of the amount of data written to SSD and HDD, and the number of redirected write requests, as shown in Table 3. Workload A has the highest percentage of data and requests redirected with *MIOS_E*, reducing the SSD written data by up to 93.3% compared with *Baseline*, which is significantly higher than *MIOS_D*. Since workload A has lower IO intensity, *MIOS_E* has more chances to redirect even when the queue length is low. Note that we also counted the padded data in BCW as the amount of data written in HDD. In such a case the total amount of data written can vary
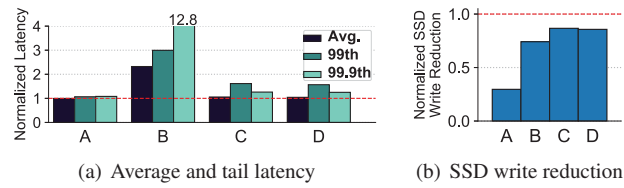

(a) Average and tail latency

(b) SSD write reduction

Figure 12: Latency and SSD written data reduction with only *F* write-state by actively issuing sync() (Normalized to *MIOS_D*).

Table 4: Amount of data written to and number of requests processed in SSD with different HDDs under workload B.

	Baseline	WD-10TB	WD-4TB	SE-4TB
SSD written data (GB)	61.2	4.1	4.2	4.4
SSD write requests (thousands)	4453	720	724	769

a great deal. Workload B has the lowest percentage of redirection with *MIOS_E*, which reduces SSD written data by 30%. Nevertheless, the absolute amount of redirected data is very large because the SSD written data in *Baseline* is larger than any of the other three workloads. Compared with *MIOS_D*, *MIOS_E* can greatly decrease the amount of data written to SSD. Therefore, it is beneficial to alleviate SSD wear-out.

However, the negative effect of *MIOS_E* is the increase of average and tail latency. In Figure 11(a), *MIOS_E* leads to generally higher average latency than *MIOS_D* by up to 40% under workload A. Although for the other three workloads, the average latency remains basically unchanged. This is because for workload A much more writes (i.e., >90%) are redirected by *MIOS_E* than by *MIOS_D*, and in HDD requests experience longer latency than in SSD. Moreover, the 99.9^{th}-percentile latency of *MIOS_E* is increased by 70% in A, 55% in B, 31% in C, and 8% in D compared to *MIOS_D*. The results can be explained by Figure 11(b). *MIOS_E* increases the average latency for nearly all the IO size groups, especially for the groups with requests of size larger than 256KB.

Moreover, we only use the *F* write state by proactively issuing $sync()$ when the ADW reaches W_f. In Figure 12, we measure the average, 99th, 99.9^{th}-percentile latency and the SSD written data reduction with this strategy. We take the MIOS_D as the baseline and present the performance normalized to MIOS_D. The 99.9^{th}-percentile latency is increased by 12.8x over MIOS_D in the B node. The 99th-percentile latency in the B, C and D nodes also be increased by 3x, 1.65x and 1.56x, respectively. This means that this strategy is less efficiency for reducing tail latency when the workload becomes heavier. This is because it redirects less SSD write data than MIOS_D when SSD suffers queue blockage. As mentioned in Section 4.2, $sync()$ is a high cost operation (e.g., several hundreds of milliseconds) to flush the HDD buffer and cannot serve any requests during the operation.

Experiment with other HDDs We use the 4TB WD, 4TB Seagate and the 10TB WD HDD to replay workload B, comparing *MIOS_D* (with the default *L* value) with the *Baseline* in terms of the amount of data written to and the number of write requests processed in SSD. Workload B is chosen for
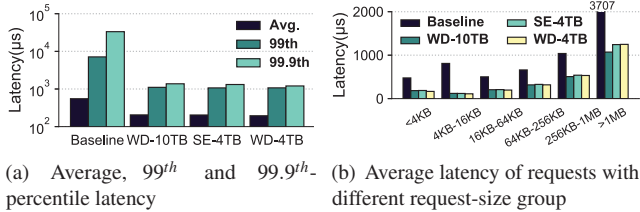
(a) Average, 99th and 99.9^{th}-percentile latency

(b) Average latency of requests with different request-size group

Figure 13: *MIOS_D* performance with three different types of HDDs under workload B.

(a) Queue length CDF
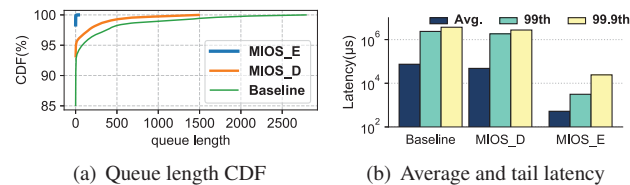
(b) Average and tail latency

Figure 14: Queue length CDF and latency under Pangu workload A, with the 660p SSD for three scheduling strategies.

this experiment, since it has the most SSD written data and the most severe SSD queue blockage, clearly reflecting the effect of IO scheduling.

Figure 13 shows that different types of HDDs do not have a significant impact on the effect of *MIOS_D*. First, the average and tail latencies for all the three HDDs are virtually identical, with a maximum difference of less than 3%. In addition, of the six request-size groups, only the >1MB group exhibits a large difference among the different HDDs. The average latency of 10TB HDD is 14% lower than that of the other two 4TB HDDs. This is because the native write performance gap between the HDDs. It can be found from Table 4 that different types of HDDs do not notably affect the amount of data redirected, with little difference of less than 5%.

Experiment with lower-performing 660p SSD Next, to further explore the effect of MIOS with different SSDs, we deploy the lower-performance 660p SSD. We replay the same workload A that has the lowest pressure on SSD, and employ the *MIOS_D* and *MIOS_E* strategies, respectively.

From the latency CDF Figure 14(a), when using SSDs with the low performance SSD, more than 7% of the requests are severely affected by long queuing delay and the maximum queue length reaches up to 2700. It surpass the experiment result with 960EVO SSD (e.g., 23 shown in Figure 8(B)). This is because when the IO intensity exceeds the ability of 660p SSD to accommodate, the SSD queue length builds up quickly. As a result in Figure 14(b), the average and tail latencies in *Baseline* rise sharply compared with 960EVO SSD shown in previous Figure 7. The average latency in *Baseline* is 90ms and the 99th-percentile latency exceeds 5 second.

With such a high workload pressure on a lower-performance SSD, MIOS can help reduce some of the pressure on SSD by redirecting some of the queued requests to HDDs. As seen from Figure 14, *MIOS_D* decreases the queue blockage with a maximum 45% queue length reduction. At

Table 5: The HDD utilization with *MIOS_E* and *MIOS_D*.

Node Type	Duration (s)	Baseline	Net Util. MIOS_D	Net Util. MIOS_E	Gross Util. MIOS_E
A	2700	7.6%	7.9%	11.9%	27.9%
B	1800	9.8%	18.2%	26.8%	56.9%
C	1800	4.1%	10.7%	16.2%	35.8%
D	1560	4.8%	12.3%	17.3%	39.5%

same time, the average latency in *MIOS_E* returns to μs-level (e.g., 521μs), and the 99th and 99.9^{th}-percentile latencies are reduced to an acceptable range of 2.4ms and 87ms, respectively. Because *MIOS_E* redirects much more SSD requests with low queue length, it prevents queue blockage in SSD, particularly for a lower-performance one. By comparing this experiment on a lower-performance SSD with an earlier one on a high-performance SSD, we believe that when the performance of SSD in hybrid storage cannot support the intensity of a write-dominated workload, MIOS and BCW can provide an effective way to improve the overall IO capacity by offloading much of the workload pressure on SSD to HDD.

In addition, we compare BCW to a system that simply adds an extra SSD. We equally distribute the workloads to two SSDs. The system can achieve the same or even better latency than *MIOS_E*, but at a significantly increased hardware cost.

5.3 BCW

We further analyze and evaluate the wasted storage-space of BCW, due to the padded data to help keep continuous HDD written pattern and skip the *S* write state.

Amount of padded data We first analyze the amount of padded data written to HDD. In Table 3, we measure the data amount with *MIOS_D* and *MIOS_E* when a BCW sequence contains one Fast state and 10 Mid/Slow stage-pairs. The stats in the table clearly indicates that *MIOS_E* generates substantially more HDD write data than *MIOS_D*. When more requests are redirected, the amount of padded data increases proportionally. For example, the padded data with *MIOS_E* is 15x that with *MIOS_D* in workload A, 3x in workload B, 4x in workloads C and D. Frequently triggering BCW increases the occurrences and thus amount of padded data. Furthermore, when the amount of redirected data increases, the Fast stage without padded data will be used up faster and more Mid/Slow stage-pairs with padded data will be executed.

HDD utilization The original Pangu traces exhibit low HDD utilization, which is defined by the percentage of time an HDD is actually working on processing IO requests. More specifically, Table 1 and 5 shows that HDDs are generally keeps very low utilization (e.g., <10%) in all four workloads.

Using MIOS, the HDD utilization has been increased with different degrees. The gross utilization is defined to be the percentage of the total execution time when the HDD is working on IO requests (including *sync*() operation), which is the real usage of the disk. The highest gross utilization is 56.9% under workload B. This means that the disk still has enough free time for HDD garbage collection. To analyze the amount of
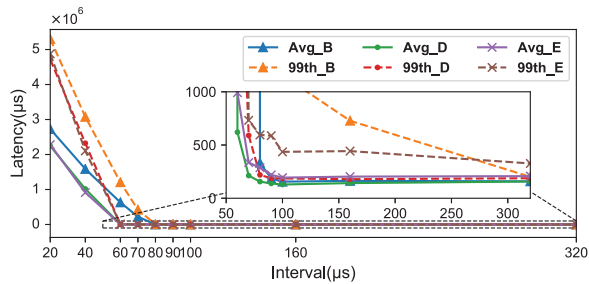
Figure 15: FIO benchmark to experiment with three strategies. The IO transmission interval is set to 20-320us.

time HDD is effectively working for user requests, we define *net utilization* as the percentage of the total execution time that the HDD spends exclusively serving user requests, excluding the time HDD spends on padding data in BCW. Thus, the net utilization is positively correlated to the amount of redirected data. The net utilization of HDD in *MIOS_E* is higher than that in *MIOS_D*. Under workload B and *MIOS_E*, HDD has the highest net utilization improvement over *Baseline*, by 2.7x, while the same is enhanced to 1.8x under *MIOS_D*.

5.4 Write Intensity

The effectiveness of BCW heavily depends on the write intensity. To better understand this relationship, we test the average and tail latency of three scheduling strategies as a function of write intensity (in terms of IO transmission interval), *Baseline* (_**B**), *MIOS_D* (_**D**) and *MIOS_E* (_**E**). We initialize the IO size to 32KB, and continuously issue write requests using FIO [4]. Since FIO cannot adjust IO intensity, we set the generated IOs with a fixed transmission interval from 20 to 320μs. We use 960EVO SSD and 10TB HDD, and set the L value to 1.

Figure 15 shows that, when the interval is between 20-60μs, the requests written to SSD are severely blocked and the 99th tail latency reaches as high as 5.2 second. In this case, both *MIOS_D* and *MIOS_E* can significantly reduce the request latency. And *MIOS_E* is slightly better than *MIOS_D* because the former handles burst writes better. When the interval is 60-80μs, *Baseline* still exhibits very high latency in SSD. However, the latency has returned to an acceptable μs-level after scheduling by MIOS. When the interval exceeds 100μs, the average and 99th-percentile latencies are stable, because there is very little SSD queue blockage with this level of request intensity. In this case, *MIOS_D* and *Baseline* have the lowest average latency and remain the same as the interval grows. However, the average and tail latency of *MIOS_E* is higher than others. This is because even if there is no queue in SSD, *MIOS_E* will still redirect requests, and the performance gap between SSD and HDD can lead to high latency.

6 Related Works

IO scheduler The IO scheduling on HDD had been adequately studied as CFQ, Anticipatory, Deadline [7], and NCQ [54]. With wide adoption of SSDs, more recent re-

searches address flash IO characteristics as read/write performance asymmetry and internal parallelism. FIOS [39] employs a fair IO timeslice management to attains fairness and high efficiency of SSD. HIOS [23] gives GC-aware and QoS-aware scheduler in host. PIQ [19] and ParDispatcher [46] minimize access conflicts between IO requests. A large body of research further offer finer scheduling inside of SSD to reduce interference between IO flows [42], write amplification [27], and GC overhead [17,21,24]. SWAN [26] partitions SSDs into multiple zones to separately serve write requests and perform GC. These works focus on homogeneous-device block-level scheduling. In contrast, MIOS schedules writes upon SSD-HDD hybrid storage.

Hybrid storage For SSD-HDD hybrid storage, most works use SSDs as a read cache or/and write buffer [2,25,41], and HDDs as the secondary or backup storage [29], due to the large performance gap between SSD and HDD. Prior works also employ HDDs as a write cache for SSDs to reduce the amount of data written to the latter [41,52]. Besides, SSD-HDD mixed RAID [32] also has been studied to complement their disadvantages with advantages. Ziggurat [55] as a tiered file system across NVMM and disks steers larger asynchronous writes into disks. SWR [31] merely redirects synchronous large writes to HDDs at highly queueing. BCW further exploits HDD buffer to redirect synchronous small writes while avoiding performance degradation.

7 Conclusion

Some hybrid storage servers serve write-dominate workloads, which leads to SSD overuse and long-tail latency while HDDs are underutilized. However, our extensive experimental study reveals that HDDs are capable of μs-level write IO latency with appropriate buffered writes. This motivated us to use HDDs to offload write requests from overused SSDs by request redirection. To this end, we present a Buffer-Controlled Write approach to proactively control buffered writes, by selecting fast writes for user requests and padding non-user data for slow writes. Then, we proposed a mixed IO scheduler to automatically steer incoming data to SSDs or HDDs based on runtime monitoring of request queues. Our extensive evaluation of MIOS and BCW, driven by real-world production workloads and benchmarks, demonstrated their efficacy.

Acknowledgments

References

[1] David G Andersen and Steven Swanson. Rethinking flash in the data center. *IEEE micro*, 30(4):52–54, 2010.

[2] Manos Athanassoulis, Shimin Chen, Anastasia Aila-maki, Phillip B. Gibbons, and Radu Stoica. Masm: efficient online updates in data warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 865–876, 2011.

[3] Guillaume Aupy, Olivier Beaumont, and Lionel Eyraud-Dubois. What size should your buffers to disks be? In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 660–669. IEEE, 2018.

[4] AXBOE. Fio: Flexible i/o tester. https://github.com/axboe/fio.

[5] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Di-dona. SILK: Preventing latency spikes in log-structured merge key-value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, Renton, WA, July 2019. USENIX Association.

[6] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *FAST*, pages 115–128, 2010.

[7] Daniel P Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

[8] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakan-tan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shash-wat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Win-dows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[9] Alibaba Clouder. Pangu – the high performance distributed file system by alibaba cloud. 2018. https://www.alibabacloud.com/blog/pangu_the_high_performance_distributed_file_system_by_alibaba_cloud_594059.

[10] Intel Corporation. Enterprise-class versus desktop-class hard drives. pages 6–7, 2016. https://www.intel.com/content/dam/support/us/en/documents/server-products/Enterprise_vs_Desktop_HDDs_2.0.pdf.

[11] Intel Corporation. Product brief of intel 660p se-ries. pages 2–2, 2019. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/660p-series-brief.pdf.

[12] Western Digital Corporation. Product brief: Wd gold enterprise class sata hdd. pages 2–3, 2019. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/internal-drives/wd-gold/product-brief-wd-gold-2579-810192.pdf.

[13] Western Digital Corporation. Wd red nas hard drives data sheet. pages 2–3, 2019. http://products.wdc.com/library/SpecSheet/ENG/2879-800002.pdf.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[15] Diego Didona and Willy Zwaenepoel. Size-aware shard-ing for improving tail latencies in in-memory key-value stores. In *NSDI*, pages 79–94, 2019.

[16] Samsung Electronics. Samsung ssd 960 evo m.2 data sheet. pages 2–4, 2017. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/660p-series-brief.pdf.

[17] Nima Elyasi, Mohammad Arjomand, Anand Sivasubra-maniam, Mahmut T Kandemir, Chita R Das, and My-oungsoo Jung. Exploiting intra-request slack to improve ssd performance. *ACM SIGARCH Computer Architec-ture News*, 45(1):375–388, 2017.

[18] FUJITSU. Mbc2073rc mbc2036rc hard disk drives product manual. pages 60–62, 2007. https://www.fujitsu.com/downloads/COMP/fel/support/disk/manuals/c141-e266-01en.pdf.

[19] Congming Gao, Liang Shi, Mengying Zhao, Chun Ja-son Xue, Kaijie Wu, and Edwin H-M Sha. Exploiting parallelism in i/o scheduling for access conflict mini-mization in flash-based solid state drives. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2014.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Le-ung. The google file system. 2003.

[21] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. vol-ume 44. ACM, 2009.

[22] Congfeng Jiang, Guangjie Han, Jiangbin Lin, Gangyong Jia, Weisong Shi, and Jian Wan. Characteristics of co-allocated online services and batch jobs in internet data centers: A case study from alibaba cloud. *IEEE Access*, 7:22495–22508, 2019.

[23] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T Kandemir. Hios: a host interface i/o scheduler for solid state disks. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 289–300. IEEE Press, 2014.

[24] 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.

[25] Taeho Kgil and Trevor Mudge. Flashcache: a nand flash memory file cache for low power web servers. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 103–112. ACM, 2006.

[26] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H Noh. Alleviating garbage collection interference through spatial separation in all flash arrays. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 799–812, 2019.

[27] Jaeho Kim, Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Disk schedulers for solid state drivers. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 295–304. ACM, 2009.

[28] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. Hermes: a heterogeneous-aware multi-tiered distributed i/o buffering system. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 219–230. ACM, 2018.

[29] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 15. ACM, 2019.

[30] Qixiao Liu and Zhibin Yu. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from alibaba trace. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 347–360. ACM, 2018.

[31] Shuyang Liu, Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. Analysis of and optimization for write-dominated hybrid storage nodes in cloud. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 403–415, 2019.

[32] Bo Mao, Hong Jiang, Suzhen Wu, Lei Tian, Dan Feng, Jianxi Chen, and Lingfang Zeng. Hpda: A hybrid parity-based disk array for enhanced performance and reliability. *ACM Transactions on Storage (TOS)*, 8(1):4, 2012.

[33] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[34] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. Sfs: random write considered harmful in solid state drives. In *FAST*, volume 12, pages 1–16, 2012.

[35] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebook's warm {BLOB} storage system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 383–398, 2014.

[36] Muthukumar Murugan and David HC Du. Rejuvenator: A static wear leveling algorithm for nand flash memory with minimized overhead. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2011.

[37] J. Ou, J. Shu, Y. Lu, L. Yi, and W. Wang. Edm: An endurance-aware data migration scheme for load balancing in ssd storage clusters. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 787–796, May 2014.

[38] Mayur R Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids: a viable solution? In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64. ACM, 2008.

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

[40] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. Azure data lake store: a hyperscale distributed file service for

big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 51–63. ACM, 2017.

[41] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *FAST*, volume 10, pages 101–114, 2010.

[42] 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.

[43] Seagate Technology. Enhanced caching advantage—turboboost and advanced write caching. pages 2–3, 2016. https://www.seagate.com/files/www-content/product-content/enterprise-performance-savvio-fam/enterprise-performance-15k-hdd/_cross-product/_shared/doc/enchanced-cache-advantage-tp691.1-1610us.pdf.

[44] Seagate Technology. Barracuda pro compute sata hdd data sheet. pages 2–3, 2018. https://www.seagate.com/www-content/datasheets/pdfs/barracuda-pro-14-tb-DS1901-9-1810US-en_US.pdf.

[45] Seagate Technology. Barracuda compute sata product manual. pages 7–8, 2019. https://www.seagate.com/www-content/product-content/desktop-hdd-fam/en-us/docs/100799391e.pdf.

[46] Hua Wang, Ping Huang, Shuang He, Ke Zhou, Chunhua Li, and Xubin He. A novel i/o scheduler for ssd with improved performance and lifetime. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, 2013.

[47] Hui Wang and Peter Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, pages 229–242, 2014.

[48] Yeong-Jae Woo and Jin-Soo Kim. Diversifying wear index for mlc nand flash memory to extend the lifetime of ssds. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, page 6. IEEE Press, 2013.

[49] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and actions: What we learned from 10k ssd-related storage system failures. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 961–976, 2019.

[50] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Transactions on Storage (TOS)*, 13(3):22, 2017.

[51] 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.

[52] Puyuan Yang, Peiquan Jin, Shouhong Wan, and Lihua Yue. Hb-storage: Optimizing ssds with a HDD write buffer. In *Web-Age Information Management - WAIM 2013 International Workshops: HardBD, MDSP, BigEM, TMSN, LQPM, BDMS, Beidaihe, China, June 14-16, 2013. Proceedings*, pages 28–39, 2013.

[53] Yang Yang, Qiang Cao, Hong Jiang, Li Yang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. Bfo: Batch-file operations on massive files for consistent performance improvement. In *35th International Conference on Massive Storage Systems and Technology (MSST'19)*, 2019.

[54] Young Jin Yu, Dong In Shin, Hyeonsang Eom, and Heon Young Yeom. Ncq vs. i/o scheduler: Preventing unexpected misbehaviors. *ACM Transactions on Storage (TOS)*, 6(1):2, 2010.

[55] 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.

INFINICACHE: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache

Ao Wang[1]*, Jingyuan Zhang[1]*, Xiaolong Ma[2], Ali Anwar[3], Lukas Rupprecht[3], Dimitrios Skourtis[3],
Vasily Tarasov[3], Feng Yan[2], Yue Cheng[1]

[1]*George Mason University*   [2]*University of Neveda, Reno*   [3]*IBM Research–Almaden*

Abstract

Internet-scale web applications are becoming increasingly storage-intensive and rely heavily on in-memory object caching to attain required I/O performance. We argue that the emerging serverless computing paradigm provides a well-suited, cost-effective platform for object caching. We present INFINICACHE, a *first-of-its-kind* in-memory object caching system that is completely built and deployed atop ephemeral serverless functions. INFINICACHE exploits and orchestrates serverless functions' memory resources to enable elastic pay-per-use caching. INFINICACHE's design combines erasure coding, intelligent billed duration control, and an efficient data backup mechanism to maximize data availability and cost effectiveness while balancing the risk of losing cached state and performance. We implement INFINICACHE on AWS Lambda and show that it: (1) achieves $31 - 96\times$ tenant-side cost savings compared to AWS ElastiCache for a large-object-only production workload, (2) can effectively provide 95.4% data availability for each one hour window, and (3) enables comparative performance seen in a typical in-memory cache.

1 Introduction

Internet-scale web applications are becoming increasingly important as they offer many useful services to the end users. Examples range from social networks [22] that serve billions of photo and video files every day to hosted container image repositories such as Docker Hub [5]. These web applications typically require a large storage capacity for the massive amount of data they must store. For instance, Docker Hub hosts over 2.6 million container images, and Facebook generates 4 PB of data daily [6].

Cloud object stores (e.g., Amazon S3, Google Cloud Storage, OpenStack Swift, etc.) have become the first choice for serving the simple object GET/PUT requests of these storage-intensive web applications. To improve request latencies for better user experience, cloud object stores are typically being used in combination with networked, lookaside In-Memory Object Caches (IMOCs) such as Redis [10] and Memcached [9]. Serving requests from an IMOC is much faster than serving them directly from a backing object store. However, due to the high cost of main memory, IMOCs are largely used only as a small cache for buffering small-sized objects that range in size from a few bytes to a few KBs [19].

Caching large objects (i.e., objects with sizes of MBs–GBs) is believed to be relatively inefficient in an IMOC as large objects consume significant memory capacity and network bandwidth. This either causes cache churn with evictions of many small objects that would be reused soon if the cache is too small, or incurs high cost for larger cache sizes.

Large object caching has been demonstrated to be effective and beneficial in cluster computing [16, 38, 47, 58]. To verify that these benefits also apply to web applications, we analyzed production traces from an IBM Docker registry [17] and identified two key properties for large objects: (1) large objects are heavily reused with strong data locality and are accessed less frequently than small ones, and (2) achieving a fast access speed for large objects is critical for system performance though it does not require as stringent a service level objective (SLO) as that for small objects, the latter of which demands sub-millisecond latencies. These properties suggest that web applications can benefit from large object caching, which state-of-the-art IMOCs currently do not provide.

The emerging serverless computing paradigm (cloud function services, or Function-as-a-Service (FaaS)) [36] introduces a new way of building and deploying applications, in which the service providers take care of resource scaling and management. Developers can thus focus on developing the function logic without managing servers. Popular uses of serverless computing today are event-driven and stateless applications such as web/API serving and batch ETL (extract, transform, and load) [1]. However, we find that serverless computing can also provide a potential cost-effective solution for resolving the tension between small and large objects in memory caching.

We demonstrate how to build an IMOC as a serverless application. A serverless application is structured as a collection of cloud functions. A function has memory that can be used to store objects that are needed during its execution. We use this memory to store cached objects. Functions are executed on demand. In our serverless IMOC, the functions are invoked by the tenant to access the cached objects. FaaS providers cache invoked functions and their state so in-memory objects are retained between function invocations. This provides a sufficient lifetime for cached objects. Providers only charge tenants when a function is invoked, in our case, when a cached object is accessed. Thus the memory capacity used to cache an object is billed only when there is a request hitting that object. *Our serverless IMOC reduces the tenants' monetary cost*

---

*These authors contributed equally to this work.

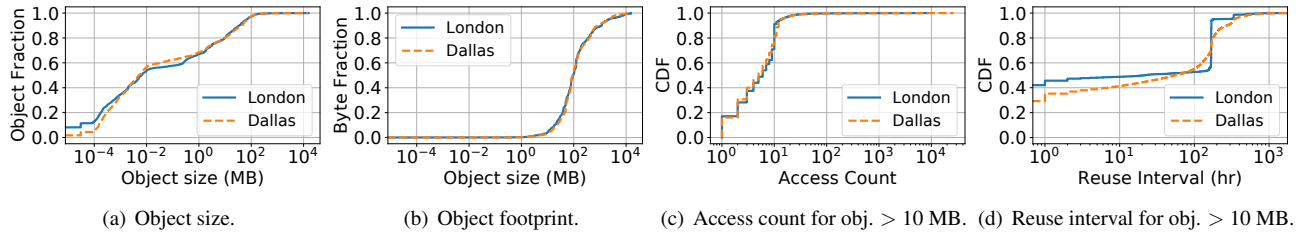|(a) Object size.|(b) Object footprint.|(c) Access count for obj. > 10 MB.|(d) Reuse interval for obj. > 10 MB.|

Figure 1: Characteristics of object sizes and access patterns in the IBM Docker registry production traces.

of memory capacity compared to other IMOCs that charge for memory capacity on an hourly basis whether the cached objects are accessed or not.

Utilizing the memory of cloud functions for object caching introduces non-trivial challenges due to the limitations and constraints of serverless computing platforms: Cloud functions have limited resource capacity (e.g., 1 CPU, up to several GB memory, and limited network bandwidth) with strict network communication constraints (e.g., no inbound TCP connection); providers may reclaim a function and its memory at any time, creating a risk of loss of the cached data.

We present INFINICACHE, a cost-effective in-memory object cache that exploits and orchestrates serverless cloud functions. INFINICACHE synthesizes a series of techniques into a holistic design to overcome the aforementioned challenges and to achieve high performance, cost effectiveness, scalability, and fault tolerance. INFINICACHE leverages erasure coding to: (1) provide fault tolerance against data loss due to function reclamation by the service provider; (2) improve performance by utilizing the aggregated network bandwidth of multiple cloud functions in parallel; and (3) use redundancy to handle tail latencies caused by straggling functions. IN-FINICACHE implements function orchestration policies that improve reliability while lowering cost. Specifically, INFINI-CACHE implements a lightweight data backup mechanism in which a cloud function periodically performs delta synchronization (delta-sync) with a *clone* of itself so as to minimize the chances that a reclaimed function causes a data loss.

In summary, this paper makes the following contributions:

- Identify the opportunities and challenges of serverless function-based object caching by performing a long-term analysis of the internal mechanisms of a popular serverless computing platform (AWS Lambda [2]).

- Design and implement INFINICACHE, the very first in-memory object caching system powered by ephemeral and "stateless" cloud functions.

- Provide an analytical model of INFINICACHE's fault tolerance mechanism built using erasure coding and periodic delta-sync techniques.

- Perform an extensive evaluation using both microbenchmark and production workloads. Experimental results show that INFINICACHE achieves performance comparable to ElastiCache for large objects and improves the cost effectiveness of cloud IMOCs by $31 - 96\times$.

2 Background and Motivation

Large-scale web applications have increasingly complex storage workload characteristics. Many modern web applications utilize a microservice architecture, which consists of hundreds to thousands of microservice modules [33]. Different modules exhibit different object size distributions and request patterns. For example, a Docker image registry service uses Redis to store small-sized container metadata (i.e., manifests), and an object store to store large-sized container images [17, 40]. While in-memory caching has been extensively studied in the context of large-scale web applications focusing on small objects, cloud cache management for large objects remains poorly explored and poses further challenges.

2.1 Large Object Caching

To obtain a better understanding of large object caching, we analyze production traces from an IBM Docker registry collected in 2017 from two datacenters (one in London, UK, and the other in Dallas, US) [17]. The goal is to reveal patterns that enable us to make realistic assumptions for the design of INFINICACHE.

Extreme Variability in Object Size. We first analyze the object size distributions. As shown in Figure 1(a), we find that object sizes span over nine orders of magnitude, and that more than 20% of objects are larger than 10 MB in size. This observation highlights the extreme variability and heterogeneity of real-world object store workloads, which further increases the complexity of cloud IMOC management.

Tension between Small and Large Objects. Efficiently managing both small and large objects in an IMOC is challenging due to two performance-cost tradeoffs. First, with limited cache capacity, large objects occupy a large amount of memory and would cause evictions of many small objects that might be reused in the near future, thus hurting performance. This is evidenced by Figure 1(b), where large objects (with size larger than 10 MB) occupy more than 95% of the total storage footprint. Second, large object requests typically consume significant network bandwidth resources, which may inevitably affect the latencies of small objects.

On one end, to prevent large objects from consuming too much memory and starving small object requests, an object size threshold is defined to not admit objects larger than the threshold [13, 23]. On the other end, system administrators can simply provision more memory (and thus more servers)

to increase the capacity of the cache. However, this would increase the total cost of ownership (TCO) with reduced resource utilization. In fact, according to our analysis of the production Docker registry workloads, for the busiest deployment among seven datacenters, the average throughput of requests with object sizes greater than 10MB is below $3,500$ `GET`s per hour.

Caching Large Objects Matters. While large object caching is challenging, it can provide significant benefit as large object workloads exhibit strong data locality. Figure 1(c) plots the access frequency distribution for all objects larger than 10 MB. About 30% of large objects are accessed at least 10 times, and the object popularity shows a long-tail distribution, with the most popular objects absorbing more than 10^4 accesses. Figure 1(d) shows the temporal reuse patterns of the large object workloads. Around 37%–46% large objects are reused within 1 hour since the last time they were accessed. The strong temporal locality patterns underscore the benefit for caching large objects for web applications.

2.2 Building a Memory Cache on Cloud Functions: Opportunities and Challenges

The above observations lead to an important question to the storage system designers and cluster administrators: *can we build a new cloud caching model that relieves the tension between performance and cost while serving large objects in a cost-effective manner?* We argue that what is missing is a truly elastic cloud storage service model that charges tenants in a request driven mode instead of capacity usage, which the emerging serverless computing naturally enables, with the following desirable properties:

Pay-Per-Use Pricing: FaaS providers (including AWS Lambda [2], Google Cloud Functions [7], Microsoft Azure Functions [4], and IBM Cloud Functions [8]) charge users at a fine granularity – for example, AWS Lambda bills on a per-invocation basis ($0.02 per 1 million invocations) and charges (CPU and memory bundle) resource usage by rounding up the function's execution time to the nearest 100 milliseconds with a rate of $0.0000166667 per second for each GB of RAM. Note the function startup cost is not billed, and does not count for its execution time. Large object IMOC workloads can take advantage of this fine-grained pay-as-you-go pricing model to keep the tenant's monetary costs low.

Short-Term Caching: More importantly, FaaS providers keep functions "warm" by caching their state in memory for a short period of time to mitigate the "cold-start" penalty[1] [15, 43, 54]. Functions that are not invoked for a while can be reclaimed by the provider, and the state stored in the functions is lost. The duration of the "warm" period may vary (ranging from tens of minutes to longer than 6 hours as observed in §4.1) for AWS Lambda, and largely depends on how frequently the Lambda function gets invoked.

Ideally, a cloud tenant can leverage the above properties naturally enabled by a FaaS provider to build an opportunistic IMOC on a serverless platform. As such, a naive design would simply invoke a cloud function and store objects into the function's memory until the function is reclaimed by the provider, and then re-insert the objects into a new function.

This approach is appealing for several reasons. First and foremost, it inherently redefines the pay-as-you-go pricing model in the context of storage (in our case memory cache storage) by realizing a new form of memory elasticity — the memory capacity used to cache an object is billed only when there is a request hitting that object. This significantly differentiates the proposed cache model against conventional cloud storage or cache services, which start charging tenants for capacity usage whenever the capacity has been committed in use. Second, it offers a virtually infinite (yet cheap) short-term capacity, which is advantageous for large object caching, since the tenants can invoke many cloud functions but have the provider pay the cost of function caching[2].

However, FaaS providers place limits on the use of cloud resources to simplify resource management, which introduces challenges in building a stateful cache service atop stateless cloud functions. Take AWS Lambda for example — each Lambda function comes with a limited CPU and memory capacity; tenants can choose a memory amount between 128MB and 3008MB in 64MB increments. Lambda allocates CPU power linearly in proportion to the amount of memory configured, capped by 1.7 cores. Each Lambda function can run at most 900 seconds (15 minutes) and will be forcibly returned when the function times out. In addition, Lambda only allows outbound TCP network connections and bans inbound connections and UDP traffic, meaning a Lambda function cannot be used to implement a server, which is necessary for stateful applications such as IMOC. However, once an outbound TCP connection is established, it can be used to issue (multiple) requests to the function. Another limitation that plagues the performance of serverless applications is the lack of quality-of-service (QoS) control. As a result, functions suffer from straggler issues [45]. *Therefore, an ideal IMOC built atop cloud functions must provide effective workaround solutions to all the above challenges.*

3 INFINICACHE Design

INFINICACHE has three components: an INFINICACHE client library, a proxy, and a Lambda function runtime used to implement cache nodes[3]. As shown in Figure 2, an INFINICACHE deployment consists of a cluster of Lambda cache nodes, which are logically partitioned and managed by multiple proxies. Each proxy orchestrates a *Lambda cache pool*. Applications interact with INFINICACHE via a client library

---

[1]"Cold start" refers to the first-ever invocation of a function instance.

[2]FaaS providers essentially pay for the cost of storing the objects, while the tenants pay for the function invocations and function duration.

[3]We use Lambda cache node and Lambda function (runtime) interchangeably in different contexts.
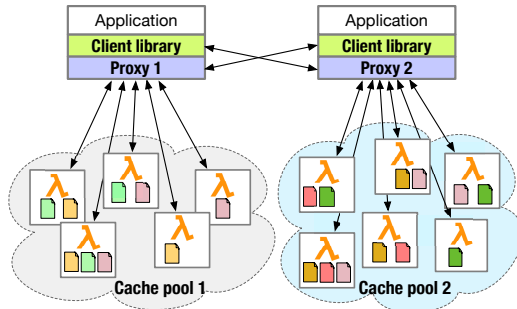
Figure 2: INFINICACHE architecture overview. Icon ▤ denotes EC-encoded object chunks. Chunks with same color belong to the same object.

that is responsible for cache invalidation upon an overwrite and cache insertion upon a read miss assuming a read-only, write-through cache; the client library encodes and decodes the objects using erasure coding (EC) and interfaces with a proxy serving as a rendezvous that streams the EC-encoded object chunks between a client library and the Lambda nodes.

INFINICACHE introduces a proxy primarily because a Lambda node cannot run in server mode due to banned inbound connections. Thus a client library has to rely on an intermediate server (the proxy) for accepting connection requests from Lambda nodes. In INFINICACHE, the client library and proxy are logically separated as they have clearly partitioned functionality, but in deployment they can be physically co-located on the same machine. To enable data sharing across different Lambda cache pools, a client can communicate with any proxy (see Figure 2).

3.1 Client Library

INFINICACHE's client library exposes to the application a clean set of `GET(key)` and `PUT(key, value)` APIs (see Figure 3). The client library is responsible for: (1) transparently handling object encoding/decoding using an embedded EC module, (2) load balancing the requests across a distributed set of proxies, and (3) determining where EC-encoded chunks are placed on a cluster of Lambda nodes.
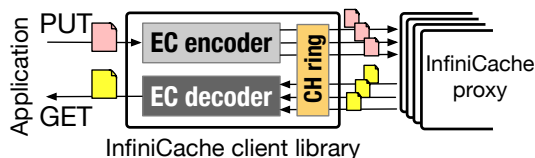

Figure 3: INFINICACHE client library (CH: consistent hashing).

Erasure Coding Processing. In our initial design, we observed that adding EC processing to the proxy would stall the chunk streaming pipeline (§3.2) and significantly impact the overall data transfer performance. Hence we made a design choice to move the computation-heavy EC part from the proxy to the client library.

The `PUT` Path. Assume that we have a multi-proxy deployment in which each proxy manages a separate Lambda node pool with shared access among clients. For a `PUT` request,
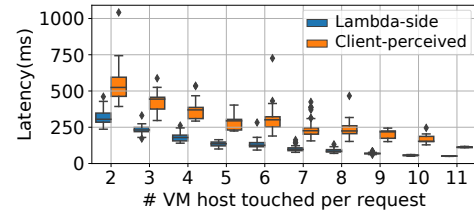

Figure 4: The box-and-whisker plot of latencies as a function of the number of VM hosts touched per request.

INFINICACHE's client library first determines the destination proxy (and therefore its backing Lambda pool) by using a consistent hashing-based load balancing approach. The client library then encodes the object with a pre-configured EC code ($(d + p)$ using a Reed-Solomon (RS) code) and produces a number of object chunks, each with a unique identifier ID_{obj_chunk} (computed as a concatenation of the object key and the chunk's sequence number). To handle extremely large objects, INFINICACHE can encode them with more aggressive EC code (e.g., $(20+4)$). Next, the client decides which Lambda nodes to store the chunks on by randomly generating a vector of non-repetitive ID_{λ}. Each encoded chunk with its piggybacked $<ID_{obj\_chunk}, ID_{\lambda}>$ is sent to the destination proxy, which streams the data to the destination Lambda nodes and remembers the locations in the Lambda pool where the chunks are cached.

The `GET` Path. A `GET` request is first sent to the proxy by using consistent hashing; the proxy then consults its mapping table, which records the chunk to Lambda node association and fetches the object chunks from the associated Lambda nodes (see §3.2). Once the chunks arrive at the client, the client library decodes the chunks, reconstructs the original object, and returns the object to the application.

Eliminating Lambda Contention. Lambda functions are hosted by EC2 Virtual Machines (VMs). A single VM can host one or more functions. AWS seems to provision Lambda functions on the smallest possible number of VMs using a greedy binpacking heuristic [54]. This could cause severe network bandwidth contention if multiple network-intensive Lambda functions get allocated on the same host VM.

We conduct an empirical study to verify this. In our study setup, each Lambda function has 256 MB memory. We use an RS code of $(10+1)$ to split a 100 MB object into 10 data chunks and 1 parity chunk, and place each chunk on a Lambda node randomly selected from a fixed sized Lambda node pool. We measure the latency of `GET` requests by scaling-up the pool from 20 to 200 Lambda nodes. As a result, the number of host VMs that the 11-chunk object spans varies proportionally as the Lambda node pool scales up and down[4]. Figure 4 shows the latency distribution as a function of the number of underlying host VM touched per request. With a larger Lambda node pool (where the request is more likely to be spread across more host VMs), we observe a decreasing

---

[4]We run command `uname` in Lambda to get the underlying host VM's IP.

trend in the latency on the Lambda-side (the time that each Lambda node spends serving the chunk request) as well as the client-perceived (end-to-end) latencies.

These results stress the need to minimize resource contention among multiple Lambda functions sharing the same VM host. While over-provisioning a large Lambda node pool with many small Lambda functions would help to statistically reduce the chances of Lambda co-location, we find that using relatively bigger Lambda functions largely eliminates Lambda co-location. Lambda's VM hosts have approximately 3 GB memory. As such, if we use Lambda functions with ≥ 1.5 GB memory, every VM host is occupied exclusively by a single Lambda function, assuming INFINICACHE's cache pool consists of Lambda functions with the same configuration[5].

3.2 Proxy

Each INFINICACHE proxy (Figure 5) is responsible for: (1) managing a pool of Lambda nodes, and (2) streaming data between clients and the Lambda nodes. Each Lambda node proactively establishes a persistent TCP connection with its managing proxy.
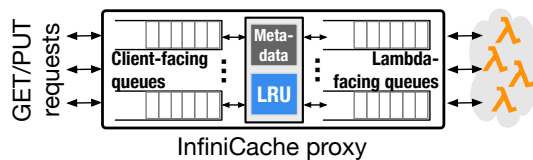

Figure 5: INFINICACHE proxy.

Pool Management.   Each proxy manages a pool of Lambda nodes, and also maintains the metadata to record the mapping between object chunks and Lambda nodes. To achieve fault tolerance, the proxy also serves as a coordinator to coordinate data migration and delta sync (see detail in §4). Each proxy tracks the memory usage of every Lambda node in the pool. The proxy starts to evict objects as long as there is not enough free memory in the Lambda pool using a CLOCK based [30] LRU policy. The LRU module operates at the object granularity at the proxy. After the eviction process, the proxy updates the mapping metadata, and inserts the new data.

First-d based Parallel I/O.   The proxy sends and receives object chunks in parallel by utilizing I/O parallelism to maximize network bandwidth utilization. To mitigate the Lambda straggler problem, the proxy directly streams the first d out of $(d + p)$ encoded object chunks to the client. Though accepting the first-d arrived chunks may likely result in an EC decoding process at the client library, as we show in §5.1, the performance benefit of the optimization outweights the EC decoding overhead with reduced tail latency for GET requests.

3.3 Lambda Function Runtime

The Lambda function runtime executes inside each Lambda instance and is designed to manage the cached object chunks in the function's memory. Our Lambda runtime uses several techniques to work around the inherent limitations of AWS Lambda. These techniques, as described below, ensure that caching is robust and cost-effective with negligible overhead.

Memory and Connection Management.   The Lambda runtime tracks cached key-value pairs that are sorted with a CLOCK-based priority queue[6] for facilitating the ordered chunk backup process described in §4.2. Since AWS Lambda does not allow inbound TCP or UDP connections, each Lambda runtime establishes a TCP connection with its designated proxy server, the first time it is invoked. A Lambda node gets its proxy's connection information via its invocation parameters. The Lambda runtime then keeps the TCP connection established until reclaimed by the provider.

Anticipatory Billed Duration Control.   AWS charges Lambda usage per 100 ms (which we call a *billing cycle*). To maximize the use of each billing cycle and to avoid the overhead of restarting Lambdas, INFINICACHE's Lambda runtime uses a timeout scheme to control how long a Lambda function runs. When a Lambda node is invoked by a chunk request, a timer is triggered to limit the function's execution time. The timeout is initially set to expire within the first billing cycle. The runtime employs a simple heuristic to decide whether to extend the timeout window. If no further chunk request arrives within the first billing cycle, the timer expires and returns 2–10 ms (a short time buffer) before the 100 ms window ends. This avoids accidentally executing into the next billing cycle. The buffer time is configurable, and is empirically decided based on the Lambda function's memory capacity. If more than one request can be served within the current billing cycle, the heuristic extends the timeout by one more billing cycle, anticipating more incoming requests.

Preflight Message.   While the proxy knows whether a Lambda node is running or has already returned, it does not know when a Lambda node will expire and return. Because of the billed duration control design that was just described, a Lambda node may return at any time. For example, right after the proxy has sent a request but before the request arrives at the Lambda, the Lambda function may expire, resulting in a denial of the request. The proxy could maintain global knowledge about the Lambda node's real-time states by periodically polling the Lambda node. However, this is costly especially if the Lambda pool size scales up to several thousand nodes.

To eliminate such overhead, the proxy issues a preflight message (PING) each time a chunk request is forwarded to the Lambda node. Upon receiving the preflight message, the Lambda runtime responds with a PONG message, delays the timeout (by extending the timer long enough to serve the incoming request), and when the request has been served, adjusts the timer to align it with the ending of the current billing cycle. To further reduce overhead, the proxy can attach the PING message as a parameter of a Lambda function

---

[5]AWS does not allow sharing Lambda-hosting VMs across tenants [20].

[6]Note that CLOCK is being leveraged for two unrelated purposes: per-proxy for object eviction (§3.2), and per-node for chunk backup ordering.
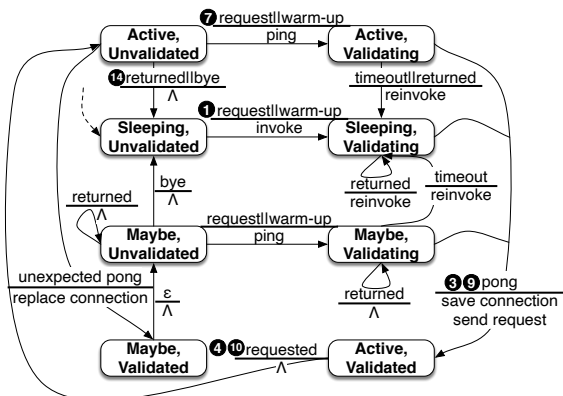
Figure 6: Lambda connection validation process in a proxy.

Figure 7: State transitions of a Lambda function runtime.

invocation request, if the Lambda node is in sleep mode (i.e., not running but cached by AWS). Once awoken, the Lambda runtime sends a `PONG` response back to the proxy.

3.4 Reliable Lambda Connections

To maintain reliable network connections between Lambda nodes and their proxy, each proxy lazily validates the status of a Lambda node every time there is a request to send. A proxy maintains three states for each Lambda connection: 1) A `Sleeping` state—a Lambda node that is not actively running; 2) An `Active` state—an actively running Lambda node; 3) A `Maybe` state—during data backup (§4.2) the original Lambda connection might have been temporarily replaced with a new connection connecting the proxy to the destination Lambda node. Figure 6 and Figure 7 depict the state transition graphs for the proxy and the Lambda function runtime, respectively. Note the step numbers show the interactions between a proxy (Figure 6) and a Lambda function (Figure 7).

Connection Lifecycle. Initially, no Lambda node is connected to the proxy. The connection is `(Sleeping,Unvalidated)`. ① When a request comes, or if a pre-warm-up is necessary, ② the proxy invokes a Lambda node. ③ Once the Lambda node is actively running and has successfully connected to its proxy, the Lambda runtime sends a `PONG` message to proxy. Now the connection's state becomes `(Active,Validated)`, and the proxy can start issuing chunk requests. ④ After the proxy sends a chunk request, the connection transits to the state `(Active,Unvalidated)`. Having served the request (⑤ transits from `Active,Idling` to `Active,Serving` while ⑥ transits back), if the proxy forwards the next request continuously, a re-validation of the connection is necessary. ⑦ A `PING` message is sent. ⑧ This time, the Lambda node replies with a `PONG` directly, which ⑨ makes the connection `(Active,Validated)` again, and ⑩ the proxy continues to issue the next chunk request. Note that the Lambda node may return anytime, or a message may timeout. In this case, the proxy re-invokes the Lambda node while marking the connection as `(Sleeping,Validating)`. Having served the request (⑪ transits from `Active,Idling` to
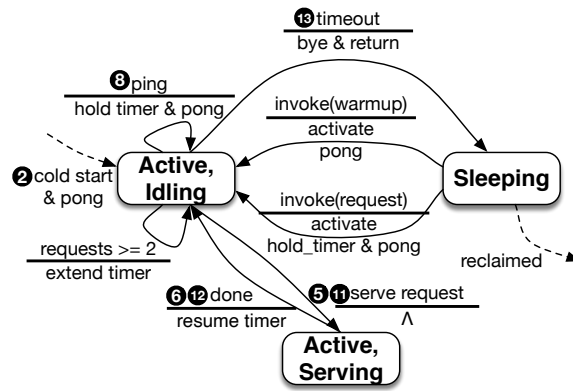
`Active,Serving` while ⑫ transits back), if no request arrives, the Lambda node ⑬ sends `BYE` to the proxy and returns, and then the proxy ⑭ transits the connection state back to `(Sleeping,Unvalidated)`.

When a connection is in the `Maybe` state, it behaves like an `Active` connection except that the proxy ignores the "return" of the source Lambda node. This does not cause a correctness issue since the source has already been replaced by a new one (i.e., the destination). The connection is marked as `(Sleeping,Unvalidated)` if a `BYE` message is received via the connection.

4 Data Availability and Fault Tolerance

In this section, we conduct a case study with AWS Lambda and describe the approaches INFINICACHE employs for maintaining practical data availability and fault tolerance over a fleet of ephemeral cloud functions with a high churn rate.

4.1 AWS Lambda Properties

While AWS allows function caching to mitigate "cold start" overhead, it does not provide any availability guarantees for the cached function and can reclaim it anytime. Hence, IN-FINICACHE needs to be robust against frequent failures of cache nodes. To better understand the stateless property of AWS Lambda and its implications on short-term data availability, we conduct an extensive black-box analysis. We analyze reclamation behaviors by quantifying the number of reclaimed Lambda functions in a 24-hour period under different warm-up strategies.

According to a recent study [54], a Lambda function that finishes execution is kept by AWS for at most 27 minutes if that function is not invoked again. A function's lifespan can be extended to hours if that function instance is invoked periodically (i.e., by so-called warm-up operations). The lifespan extension varies according to the warm-up strategy as well as AWS' internal resource management policy.

We deploy a pool of 300–400 Lambda functions with the same memory configuration, and re-invoke each one of them every *N* minute(s). Each function simply returns an ID value that the function computed when it was invoked the first time. If AWS reclaims an already invoked, cached function, a new
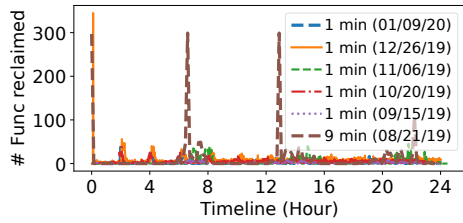
Figure 8: Number of functions being reclaimed over time under various warm-up strategies.
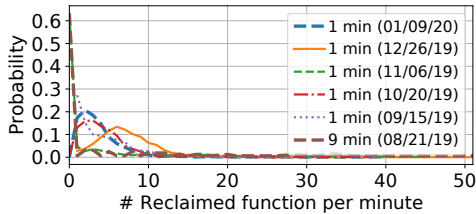

Figure 9: Probability distribution of the number of functions reclaimed per minute on the sampled days.

function instance will be instantiated at the next invocation request and the ID of this function will change. We keep track of the ID to detect whether a function has been reclaimed or not. We evaluate two warm-up strategies: a low warm-up frequency (every 9 minutes) and a relatively high frequency (every 1 minute). We ran each strategy during a span of 6 months (from August 2019 to January 2020), and recorded the number of function reclaiming events.

As shown in Figure 8, for `9 min (08/21/19)`, we observe a large number of function reclaiming events clustered around hour 6, hour 12, and hour 20–22. The number of reclaimed functions spiked roughly every 6 hours and almost all the functions get reclaimed. For `1 min (09/15/19)`, the situation got much better; the peak number of reclaiming events gets reduced to 22, 21, and 16 at hour 6, respectively. Similar trends appeared in November, but got substantially changed in December and January – for example for `1 min (12/26/19)`, instead of spiking every 6 hours, AWS continuously reclaimed Lambda functions with an hourly reclaiming rate of 36. This is possibly due to AWS Lambda's internal policy changes after AWS announced the launch of provisioned concurrency [3] for Lambda on December 03, 2019.

Figure 9 shows the function reclaiming events roughly follow a Zipf distribution for August, September, and November (with different s values), and a Poisson distribution for October, December, and January (with different λ values). With that, we can calculate an approximate range of probabilities of r functions being reclaimed simultaneously in a user-defined interval (§4.3). Motivated by these observations, we argue that with careful design, we can improve the data availability for INFINICACHE.

4.2 Maximizing Data Availability

INFINICACHE adopts three techniques for maximizing data availability: (1) EC is used to enable data recovery for up to p object chunk losses given an RS code $(d + p)$. In the case that there are more than p chunks lost, tenants need
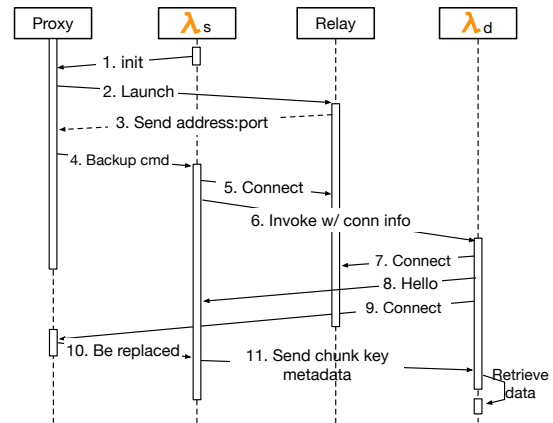

Figure 10: INFINICACHE's backup protocol.

to retrieve the data from the backing object store; (2) each Lambda runtime is warmed up after every T_{warm} interval of time; we use a T_{warm} value of of 1 minute as motivated by our observations in §4.1; (3) to further enhance availability, a delta-sync based data backup scheme provides incremental backups every T_{bak} interval. The selection of T_{bak} is a trade-off between availability, runtime overhead, and cost effectiveness: a shorter interval lowers the data loss rate while a longer interval incurs less backup overhead and less cost. In the following, we explain the backup scheme in more detail.

Backup Protocol. INFINICACHE performs periodic delta-sync backups between two peer replicas[7] of the same Lambda function. We choose peer replicas instead of distinct Lambdas to be able to seamlessly failover to one of them in case the other gets reclaimed.

In light of the observations in Figure 8, we design a backup scheme that preserves the following properties: (1) autonomicity—a Lambda node should backup itself with minimum help from the proxy to keep proxy logic simple; (2) high availability—the service provided by the Lambda node should not be interrupted; and (3) low network overhead—large object workloads are network bandwidth sensitive so backups should cause low or no extra network overhead. To this end, we adopt an efficient, Lambda-aware mechanism that performs delta-sync between two peer replicas of the same Lambda function.

The protocol sequence graph is depicted in Figure 10. In Step 1, a Lambda node λ_s, serving as the source cache node, sends an `init-backup` message to its proxy to initialize a backup process every T_{bak}. Acknowledging this message, in Step 2, the proxy launches a new process called *relay* (co-allocated with proxy), which serves to forward TCP packets between λ_s and a destination Lambda node λ_d, i.e., the Lambda node that receives the backup. In Step 3, the relay process sends its own network information (address:port) to the proxy, which issues a `backup` command in Step 4 to λ_s, piggybacked with the relay's connection information.

---

[7]Concurrent invocations to the same function produce multiple concurrent Lambda instances – this process is called auto-scaling. Here we call each instance of the same function a *peer replica*.

In Step 5, λ_s establishes a TCP connection with the relay and in Step 6 invokes a peer replica instance of the λ_s function, which serves as λ_d; at the same time, λ_s passes the connection information of both the relay and proxy to λ_d as the Lambda invocation parameters. In Step 7, λ_d establishes a TCP connection with the relay. If connected successfully, an indirect network channel is bridged through a relay between λ_s and λ_d. Then λ_d sends a `hello` message to λ_s in Step 8 and connects to the proxy in Step 9.

Upon establishing the connection with λ_d, in Step 10, the proxy disconnects from λ_s, which makes λ_d the only active connection to the data of λ_s. Hence, the proxy forwards all requests to λ_d while λ_d forwards requests to λ_s, if it has not yet received the requested data. To receive data, λ_d sends a `hello` to λ_s in Step 11 and λ_s starts sending metadata (stored chunk keys) in an order from MRU to LRU. Once λ_d has received all the keys, it starts the data migration by retrieving the data associated with the keys from λ_s.

If λ_d receives a `PUT` request during data retrieval and the key is not found, it inserts the new data in its cache and then forwards it to λ_s. If a `GET` request is received for a key that has been retrieved already from λ_s, λ_d directly responds with the requested chunk. Otherwise, λ_d forwards the request to λ_s, responds to the proxy, and then caches the key and the corresponding chunk.

After data retrieval completes, λ_d returns and the connection to the proxy becomes inactive. Hence, the next time the proxy invokes this Lambda function, AWS would launch one of the two, λ_s or λ_d, if they have not been reclaimed yet. As they are now in sync, they can both serve the data. After another interval T_{bak}, the whole backup procedure repeats. λ_d only retrieves the "delta" part of data to reduce overhead.

4.3 Data Availability and Cost Analysis

Availability Analysis. To better understand the data availability of INFINICACHE, we build an analytical model. Assume N_λ is the total number of Lambda nodes. At time T_r, a number r of nodes are found reclaimed. m is the minimum number of chunks that leads to an object loss and n is the number of EC chunks of a object. An object is considered not available if there are at least m chunks lost due to function reclaiming. The probability $P(r)$ that an object is not available (i.e., lost) is formalized as: $P(r) = \sum_{i=m}^{n} p_i$, where:

$$p_i = \frac{C(r,i)C(N_\lambda - r, n - i)}{C(N_\lambda, n)}. \quad (1)$$

Here $C(r,i)$ is the combinations in which r reclaimed Lambda nodes happens to hold i chunks belonging to the same object. $C(N_\lambda - r, n - i)$ is the combinations in which the rest chunks of that object are held in Lambda nodes that have not been reclaimed. $C(N_\lambda, n)$ is the combinations in which all Lambda nodes hold all chunks of an object.

Assuming $p_d(r)$ is the probability distribution of reclaiming r Lambda nodes at T_r, the probability of losing an object

P_l is the sum of the probabilities of losing one object when at least m Lambda nodes are reclaimed:

$$P_l = \sum_{r=m}^{N_\lambda} P(r)p_d(r) = \sum_{r=m}^{N_\lambda} \sum_{i=m}^{n} \frac{C(r,i)C(N_\lambda - r, n - i)}{C(N_\lambda, n)} p_d(r). \quad (2)$$

One observation is that $\frac{p_m}{p_{m+1}}$ can be larger than 10. E.g., for a 400-Lambda nodes deployment with $N_\lambda = 400$, an RS code of $(10 + 2)$, and a warm-up interval of 1 minute, if 12 nodes get reclaimed simultaneously at time T_r, we have $p_3/p_4 = 18.8$ for $r = 12$, and $P(r)$ is only about 5% larger than p_3. So we can simplify the formulation as $P(r) \approx p_m$, thus P_l can be simplified as:

$$P_l \approx \sum_{r=m}^{N_\lambda} \frac{C(r,m)C(N_\lambda - r, n - m)}{C(N_\lambda, n)} p_d(r). \quad (3)$$

In our case study, $N_\lambda = 400$, $n = 12$, $m = 3$, and $T_{warm} = 1$ *min*. With Equation 3 we get $P_l = 0.0039\% \sim 0.11\%$ or an availability $P_a = 99.89\% \sim 99.9961\%$ for 1 minute, and $93.36 \sim 99.76\%$ for 1 hour based on the variable probability distribution of Lambda reclaiming policies we observed over a six-month period (§4.1).

Cost Analysis. To maintain high availability, INFINICACHE employs EC, warm-up, and delta-sync backup, which all incur extra cost. For a better understanding of how these techniques impact total cost, we build an analytical cost model. To simplify our presentation, we do not explicitly express the EC configuration using an RS code $(d + p)$, but rather reflect it in the total number of instances N_λ. The total cost per hour C is therefore composed of (1) serving chunk requests (C_{ser}), (2) warming-up functions (C_w), and (3) backing up data, (C_{bak}). Thus, $C = C_{ser} + C_w + C_{bak}$ Next, we introduce each term respectively.

• *Serving cost C_{ser}.* AWS charges function invocations and function duration. We denote the price per invocation as c_{req} and the duration price of per GB-second as c_d. The function duration is rounded up to the nearest 100 ms, we define a round-up operation $ceil_{100}(.)$. Assume Lambda's memory is M GB, the average hourly request rate is n_{ser}, and the duration of each invocation is t_{ser} ms, we have:

$$C_{ser} = n_{ser} * c_{req} + n_{ser} * ceil_{100}(t_{ser})/1000 * M * c_d. \quad (4)$$

• *Warm-up cost C_w.* The backup frequency $f_w = 60/T_{warm}$. The warm-up duration t_w is typically in the range of a few ms and therefore we have $ceil_{100}(t_w) = 100$ ms. Thus we have:

$$C_w = N_\lambda * f_w * c_{req} + N_\lambda * f_w * 0.1 * M * c_d. \quad (5)$$

• *Backup cost C_{bak}.* The backup frequency is denoted as $f_{bak} = 60/T_{bak}$. We have:

$$C_{bak} = N_\lambda * f_{bak} * c_{req} + N_\lambda * f_{bak} * t_{bak} * M * c_d. \quad (6)$$

As shown in §5.2, the backup cost is a dominating factor whose proportion increases as more data are being cached.

5 Evaluation

In this section, we evaluate INFINICACHE on AWS Lambda using microbenchmarks and a production workload from the IBM Docker registry [17].
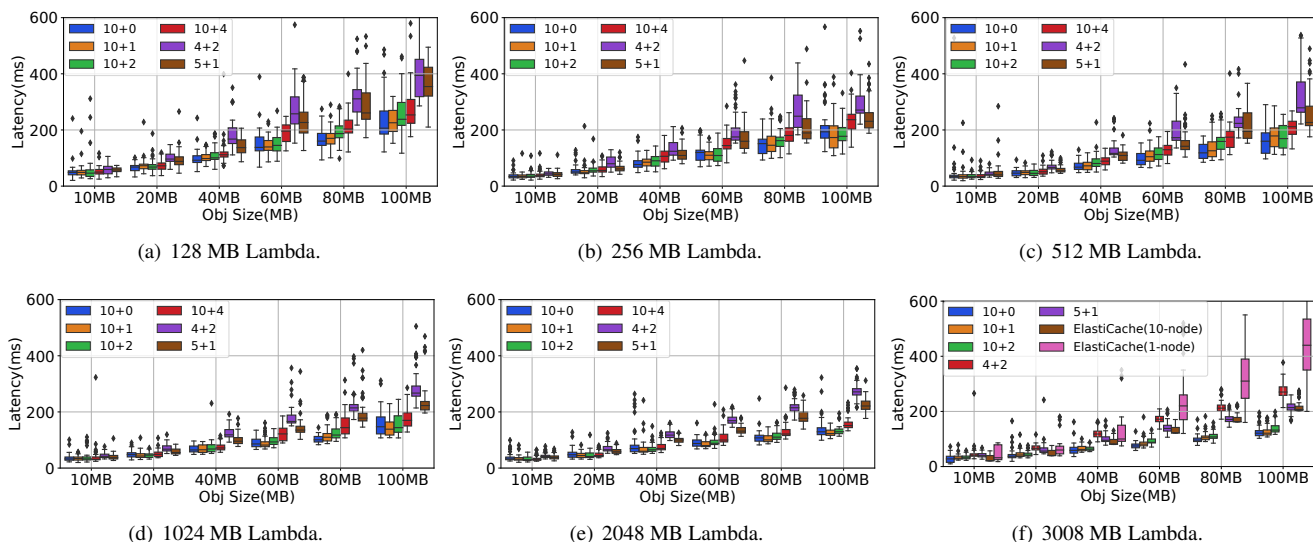
(a) 128 MB Lambda.	(b) 256 MB Lambda.	(c) 512 MB Lambda.
(d) 1024 MB Lambda.	(e) 2048 MB Lambda.	(f) 3008 MB Lambda.

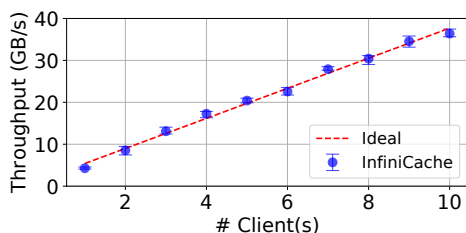**Figure 11:** Microbenchmark performance.

Figure 12: Scalability of INFINICACHE.

Implementation. We have implemented a prototype of IN-FINICACHE using $5,340$ lines of Go (460 LoC for the client library, $3,447$ for the proxy, and $1,433$ for the Lambda runtime). The EC module of the client library is implemented using the Golang reedsolomon lib [11], which uses Intel's AVX-512 for accelerating EC computation.

Setup. Our experiments use AWS Lambda functions with various configurations. Unless otherwise specified, we deploy the client (with INFINICACHE's client library) and proxy on `c5n.4xlarge` EC2 VM instances. The Lambda functions are in the same Amazon Virtual Private Cloud (VPC) as the EC2 instances and are equipped with a 10 Gbps network connection. The Lambda functions' network bandwidth increases with its memory amount; we observed a throughput of 50–160 MBps (from the smallest memory amount of 128 MB to the largest memory amount of 3008 MB) between a `c5n.4xlarge` EC2 instance and a Lambda function using `iperf3`.

5.1 Microbenchmark Performance

We first evaluate the performance of INFINICACHE under synthetic GET-only workloads generated using a simple benchmark tool. With the microbenchmarking tests, we seek to understand how different configuration knobs impact INFINI-CACHE's performance. The evaluated configuration knobs include: EC RS code (we compare $(10+1)$, $(10+2)$, $(4+2)$, $(5+1)$, with a $(10+0)$ baseline, which directly splits an object into 10 chunks without EC encoding/decoding), ob-

ject sizes (10–100 MB), and the Lambda function's resource configurations (128–3008 MB).

Figure 11 shows the distributions of end-to-end request latencies seen under different configuration settings. Invoking a warm Lambda function takes about 13 ms on average (with the Go AWS SDK API), which is included in the end-to-end latency results. We observe that the $(10+1)$ code performs best compared to other RS code configurations. This is due to two reasons. First, $(10+1)$ results in a maximum I/O parallelism factor of 10 (first-k parallel I/O is described in §3.2), and second, it keeps the EC decoding overhead at a minimum (the higher the number of parity chunks, the longer it takes for RS to decode). The caveat of using $(10+1)$ is that it trades off fault tolerance for better performance.

Another observation is that the $(10+0)$ case does not seem to lead to a better performance than that of $(10+1)$ and in several cases even sees higher tail latencies. This is due to the fact that $(10+0)$ suffers from Lambda straggler issues, which outweighs the performance gained by fully eliminating the EC decoding overhead. In contrast, $(10+1)$'s first-d approach adds redundancy and this request-level redundancy helps mitigate the impact of stragglers.

A Lambda function's resource configuration has a great impact on INFINICACHE's latency. For example, $(10+1)$ achieves latencies in the range of 110–290 ms (Figure 11(c)) with 512 MB Lambda functions for objects of 100 MB, whereas with 2048 MB Lambda functions, latencies improve to 100–160 ms (Figure 11(e)). In addition, latency improvement hits a plateau for Lambda functions equipped with more than 1024 MB memory because larger Lambda functions eliminate the network bottleneck for large chunk transfers.

To compare INFINICACHE with an existing solution, we choose ElastiCache (Redis) and deploy it in two modes, a `1-node` deployment using a `cache.r5.8xlarge` instance, and a scale-out `10-node` deployment using `cache.r5.xlarge` in-
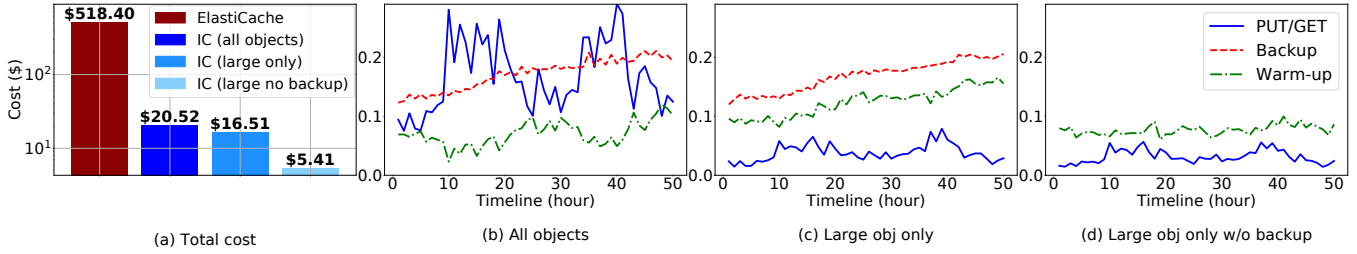
Figure 13: Total $ cost (a) for ElastiCache and INFINICACHE (IC), and INFINICACHE's hourly cost breakdown under various settings (b)-(e).
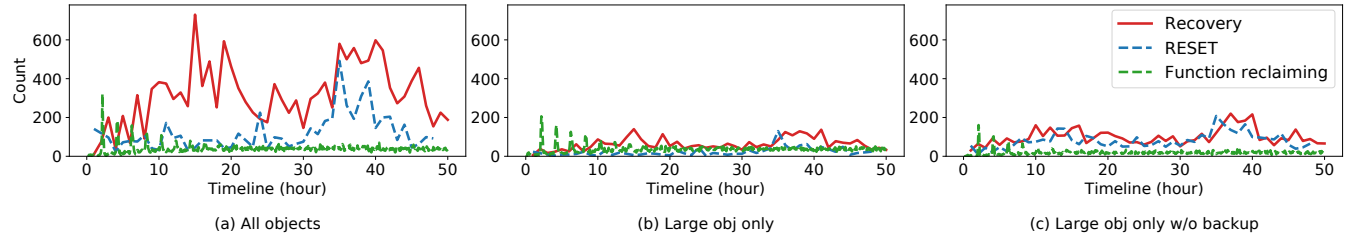

Figure 14: Timeline of INFINICACHE's fault tolerance activities under various workload settings.

Workload	WSS	Thpt	EC	IC	IC w/o backup
All objects	1,169 GB	3,654	67.9%	64.7%	-
Large obj. only	1,036 GB	750	65.9%	63.6%	56.1%

Table 1: Workloads' working set sizes (WSS), throughput (average GETs per hour), and the cache hit ratio achieved by ElastiCache (EC) and INFINICACHE (IC).

stances. As shown in Figure 11(f), INFINICACHE outperforms the 1-node ElastiCache for all object sizes, as Redis is single-threaded and cannot handle concurrent large I/Os as efficiently. For larger object sizes, INFINICACHE with $(10 + 1)$ and $(10 + 2)$ consistently achieves lower latencies compared to the 10-node ElastiCache, thanks to INFINICACHE's first-d based data streaming optimization. These results show that INFINICACHE's performance is competitive as an IMOC.

Scalability. In this test, we setup a multi-client deployment to simulate a realistic use case in which a tenant has multiple microservices that concurrently read from and write to INFINICACHE. To do so, we vary the number of clients from 1 to 10. We also deploy a 5-proxy cluster where each proxy manages a 50-node Lambda pool (and each Lambda function has 1024 MB memory). Each client uses consistent hashing to talk to different proxies for shared data access (see Figure 2).

Figure 12 shows the throughput in terms of GB/s. We observe that INFINICACHE's throughput scales linearly as the number of clients increases. Ideally, INFINICACHE can scale linearly as long as more Lambda nodes are available for serving GET requests.

5.2 Production Workload

In this section, we evaluate INFINICACHE using the IBM Docker registry production workload (detailed in §2). The original workload contains a 75-day request trace spanning 7 geographically distributed datacenters. Out of the 7 datacenters, we select Dallas, which features the highest load. We parse the Dallas trace for GET requests that read a blob (i.e., a Docker image layer). We test two workload settings: 1) all

objects (including both small and large, with a working set size (WSS) of 1,169 GB as shown in Table 1), and 2) large object only (only including objects larger than 10 MB, with a WSS of 1,036 GB).

We replay the first 50 hours of the Dallas trace in *real time* and skip the largest object which was 8 GB (there was only one object). A GET upon a miss results in a PUT that inserts the object into the cache. INFINICACHE is configured with a pool consisting of 400 1.5 GB Lambda functions, which are managed by one proxy co-located with our trace replayer as the client. We use an EC RS configuration of $(10 + 2)$ to balance performance with fault tolerance. We select a warm-up interval T_{warm} as 1 minute (due to our study in Figure 8) and a backup interval T_{bak} as 5 minutes (to balance the cost-availability tradeoff). For the large object only workload, we test two INFINICACHE configurations: the default case with backup enabled, and a case with backup disabled (without backup).

Cost Savings. Figure 13(a) shows the accumulated monetary cost of INFINICACHE in comparison with an ElastiCache setup of one cache.r5.24xlarge Redis instance with 635.61 GB memory. By the end of hour 50, ElastiCache costs $518.4, while INFINICACHE with all objects costs $20.52. Caching only large objects bigger than 10 MB leads to a cost of $16.51 for INFINICACHE. INFINICACHE's pay-per-use serverless substrate effectively brings down the total cost by 96.8% with a cost effectiveness improvement of $31\times$. By disabling the backup option, INFINICACHE further lowers down the cost to $5.41, which is $96\times$ cheaper than ElastiCache. However, the low monetary cost for tenants comes at a price of impacted availability and hit ratio – INFINICACHE without backup sees a lower hit ratio of 56.1% (Table 1) – thus presenting a reasonable tradeoff for tenants to choose.

INFINICACHE's monetary cost is composed of three parts: (1) serving GETs/PUTs, (2) warming-up Lambda functions, and (3) backing up data. Figure 13(b)-(d) details the cost
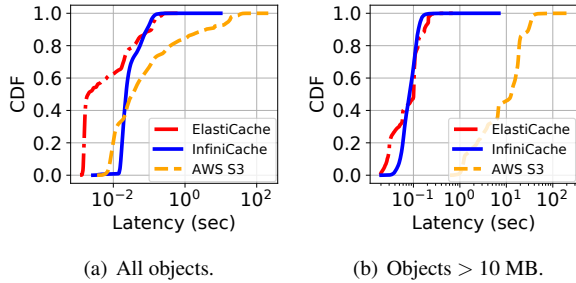
(a) All objects.   (b) Objects > 10 MB.

Figure 15: INFINICACHE latencies vs. AWS S3 and ElastiCache.

Figure 16: Normalized latencies grouped by object sizes. Each is normalized to that of ElastiCache.

breakdown, further explaining the cost variations of different combinations of workload and INFINICACHE settings. In Figure 13(b), we see that about 41% of the total cost is spent on serving data requests under the workload with `all objects`; this is because a significant portion of requests are for small objects. In contrast, for the `large object only` workload shown in Figure 13(c), the backup and warmup cost dominates, occupying around 88.3% of the overall cost. This is because the hourly request rate for `large object only` is significantly lower than that for the `all object` workload. Furthermore, disabling backup leads to a dramatic cost-effectiveness improvement (see Figure 13(d)). The warm-up cost is different between Figure 13(c) and Figure 13(d), because with the backup option enabled, a warm-up invocation may trigger a backup, and thus increase the warm-up duration.

Fault Tolerance. Figure 14 shows INFINICACHE's fault tolerance activities for different cases. An object loss (losing all the replicas of more than p chunks) results in a cache miss which triggers a RESET; the RESET fetches the lost object from a backing store and reinserts it into INFINICACHE. We observe that EC-based recovery activities and RESETs mostly coincide with the occurrence of request spikes at hour 15–20 and hour 34–42. Under the workload of `all objects` (Figure 14(a)), we see a total of 5,720 RESET events. This number is reduced to 1,085 for the `large object only` workload (Figure 14(b)), leading to an availability of 95.4%; as shown in Figure 14(c). INFINICACHE `without backup` sees 3,912 RESETs, which is 18.6% of 21,022 read hits in total. RESETs also result in a lower cache hit ratio for INFINICACHE, compared to ElastiCache, as shown in Table 1.

Performance Benefit. We replay the first 50 hours of the Dallas trace against AWS S3 to simulate a deployed Docker registry service using S3 as a backing store. We compare INFINICACHE's performance against AWS ElastiCache and S3 seen under the same workload (`all objects`). Figure 15 shows the overall trend of latency distribution, and Figure 16 shows the distribution of the normalized latencies as a function of the object sizes.

We make the following three observations. (1) In Figure 15(b), we see that, compared to S3, INFINICACHE achieves superior performance improvement for large objects. For about 60% of all large requests, INFINICACHE is able to achieve an improvement of at least 100×. This trend demon-
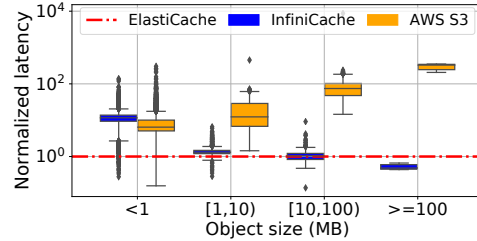
strates the efficacy of INFINICACHE in serving as an IMOC in front of a cloud object store. (2) INFINICACHE is particularly good at optimizing latencies for large objects. This is evidenced by two facts: i) INFINICACHE achieves almost identical performance as ElastiCache for objects sizing from 1–100 MB; and ii) INFINICACHE achieves consistently lower latencies than ElastiCache for objects larger than 100 MB (see Figure 16), due to INFINICACHE's I/O parallelism. (3) INFINICACHE incurs significant overhead for objects smaller than 1 MB (Figure 16), since fetching an object from INFINICACHE typically requires to invoke Lambda functions, which takes on average 13 ms and is much slower than directly fetching a small object from ElastiCache.

6 Discussion

In this section, we discuss the limitations and possible future directions of INFINICACHE.
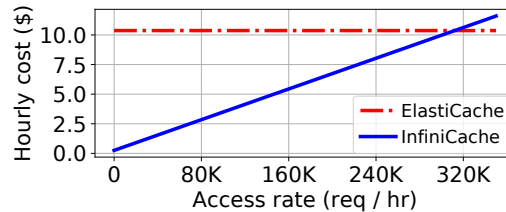
Figure 17: Hourly $ cost (Y-axis) of INFINICACHE with 400 1.5 GB Lambdas vs. one `cache.r5.24xlarge` ElastiCache instance, as a function of access rate (X-axis).

Small Object Caching. Small object-intensive memory caching workloads have a high traffic rate, typically ranging from thousands to hundreds of thousands of requests per second [19]. Serverless computing platforms are not cost-effective under such workloads, because the high data traffic will significantly increase the per-invocation cost and completely outweigh the pay-per-use benefit. Figure 17 compares the hourly cost of INFINICACHE with ElastiCache, assuming the cost models in §4.3 and configurations in §5.2. The hourly cost increases monotonically with the access rate, and eventually overshoots ElastiCache when the access rate exceeds 312 K requests per hour (86 requests per second).

Porting INFINICACHE to Other FaaS Providers. To the best of our knowledge, major serverless computing service providers such as Google Cloud Functions and Microsoft Azure Functions all provide function caching with various lifespans to mitigate the cost of cold startups [54]. Google

Cloud Functions imposes similar constraints on tenants: e.g., banned in-bound TCP connections and limited function CPU/memory resources. The design of INFINICACHE should be portable to other major serverless computing platforms such as Google Cloud Functions, with minor source code modifications to work with Google Cloud's APIs.

Service Provider's Policy Changes. Service providers may change their internal implementations and policies in response to systems like INFINICACHE. On the one hand, *statefulness* is urgently demanded by today's FaaS tenants – providing durable state caching is critical to support a broader range of complex stateful applications [12, 21, 52] such as data analytics [45] and parallel & scientific computing [25, 49]. On the other hand, to strike a balance, providers could introduce new pricing models for *stateful* FaaS applications – tenants can get stateful Lambda functions by paying slightly more than that is charged by a completely stateless one. The new feature recently launched by AWS Lambda, provisioned concurrency [3], pins warm Lambda functions in memory but without any availability guarantee (provisioned Lambdas may get reclaimed, and re-initialized periodically. But the reclamation frequency is low compared to non-provisioned Lambdas), and charges tenants hourly ($0.015 per GB per hour, no matter whether the provisioned functions get invoked), which is similar to EC2 VMs' pricing model. Nonetheless, it opens up research opportunities for new serverless-oriented cloud economics. We leave developing durable storage atop INFINICACHE in support of new stateful serverless applications as considerations for our future work.

Using INFINICACHE as a White-Box Approach. INFINICACHE presents a practical yet effective solution that exploits AWS Lambda as a black-box to achieve cost effectiveness, availability, and performance for cloud tenants. Our findings also imply that modern datacenter management systems could potentially leverage such techniques to provide short-term (e.g., intermediate data) caching for data-intensive applications such as big data analytics. Serving as a white-box solution, datacenter operators can use global knowledge to optimize data availability and locality. We hope future work will build on ours to develop new storage frameworks that can more efficiently utilize ephemeral datacenter resources.

7 Related Work

Cost-Effective Cloud Storage. Considerable prior work [14, 44, 46, 56, 57] has examined ways to minimize the usage cost of cloud storage. SPANStore [56] adopts a hybrid cloud approach by spreading data across multiple cloud service providers and exploits pricing discrepancies across providers. By contrast, INFINICACHE focuses on exploiting stateless cloud function services to achieve pay-per-use storage elasticity with dramatically reduced cost.

Exploiting Spot Cloud Resources. Researchers have explored spot and burstable cloud resources to improve the cost effectiveness of applications such as memory caching [53],

IaaS services [50], and batch computing [51]. INFINICACHE differs from them in several aspects: (1) ephemeral cloud functions exhibit significantly higher churn than the more stable spot instances; (2) cloud functions are inherently "serverless" and cannot directly host serverful long-running applications which accept inbound network connections; and (3) spot instances are not automatically cached by providers unlike cloud functions.

In-Memory Key-Value Stores. A large body of research [26, 27, 28, 29, 37, 39, 41, 42, 48, 55] focuses on improving the performance of in-memory key-value stores for small-object intensive workloads. INFINICACHE is specifically designed and optimized for large objects with sizes ranging from MBs to GBs. EC-Cache [47] and SP-Cache [58] are in-memory caches built atop Alluxio [38] to provide large object caching for data-intensive cluster computing workloads. They split the large objects into smaller chunks (EC-Cache leverages erasure coding while SP-Cache directly partitions objects) and perform curated chunk placement to achieve load balancing. The role of erasure coding in INFINICACHE is multi-fold: similar to EC-Cache [47], INFINICACHE leverages erasure coding to mitigate the cloud functions' straggler issue; erasure coding also provides space-efficient fault tolerance against potential loss of cloud functions.

New Applications of Serverless Computing. Researchers have identified new applications for serverless computing in data analytics [25, 35], video processing [18, 32], linear algebra [49], machine learning [24, 34], and software compilation [31]. However, these applications exploit the computing power of serverless platforms to parallelize and accelerate compute-intensive jobs, whereas INFINICACHE presents a completely new use case of cloud function services— implementing a stateful storage service atop stateless cloud functions by exploiting transparent function caching.

8 Conclusion

With web applications becoming increasingly storage-intensive, it is imperative to revisit the design of in-memory object caching in order to efficiently deal with both small and large objects. We have presented a novel in-memory object caching solution that achieves high cost effectiveness and good availability for large object caching by building INFINICACHE on top of a popular serverless computing platform (AWS Lambda). For the first time in the literature, INFINICACHE enables request-driven pay-per-use elasticity at the cloud storage level with a serverless architecture. INFINICACHE does this by synthesizing a series of techniques including erasure coding and a delta-sync-based data backup scheme. Being serverless-aware, INFINICACHE intelligently orchestrates ephemeral cloud functions and improves cost effectiveness by 31× compared to ElastiCache, while maintaining 95.4% availability for each hour time window.

INFINICACHE's source code is available at:

https://github.com/mason-leap-lab/InfiniCache.

Acknowledgments

References

[1] 2018 Serverless Community Survey: huge growth in serverless usage. https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/.

[2] AWS Lambda. https://aws.amazon.com/lambda/.

[3] AWS Lambda announces Provisioned Concurrency (Posted on: Dec 3, 2019). https://aws.amazon.com/about-aws/whats-new/2019/12/aws-lambda-announces-provisioned-concurrency/.

[4] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[5] Docker Hub: Container Image Library. https://www.docker.com/products/docker-hub.

[6] Facebook's Top Open Data Problems. https://research.fb.com/blog/2014/10/facebook-s-top-open-data-problems/.

[7] Google Cloud Functions. https://cloud.google.com/functions/.

[8] IBM Cloud Functions. https://console.bluemix.net/openwhisk/.

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

[10] Redis. https://redis.io/.

[11] Reed-Solomon Erasure Coding in Go. https://github.com/klauspost/reedsolomon.

[12] The Serverless Supercomputer: Harnessing the power of cloud functions to build a new breed of distributed systems. https://read.acloud.guru/https-medium-com-timawagner-the-serverless-supercomputer-555e93bbfa08.

[13] Varnish HTTP Cache. https://varnish-cache.org/.

[14] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. Racs: A case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 229–240, New York, NY, USA, 2010. ACM.

[15] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, 2018. USENIX Association.

[16] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 267–280, San Jose, CA, 2012. USENIX.

[17] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 265–278, Oakland, CA, 2018. USENIX Association.

[18] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 263–274, New York, NY, USA, 2018. ACM.

[19] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[20] AWS. Whitepaper: Security overview of AWS Lambda Security and compliance best practices. March 2019.

[21] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, pages 41–54, New York, NY, USA, 2019. ACM.

[22] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 47–60, Berkeley, CA, USA, 2010. USENIX Association.

[23] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, Boston, MA, 2017. USENIX Association.

[24] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew

Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 13–24, New York, NY, USA, 2019. ACM.

[25] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. In search of a fast and efficient serverless dag engine. In *4th International Parallel Data Systems Workshop (PDSW 2019)*, 2019.

[26] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 4:1–4:16, New York, NY, USA, 2015. ACM.

[27] 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, Santa Clara, CA, March 2016. USENIX Association.

[28] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 79–94, Boston, MA, February 2019. USENIX Association.

[29] Bin Fan, David G. Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, Lombard, IL, 2013. USENIX.

[30] Fernando J Corbato. A paging experiment with the multics system. Technical Report.

[31] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[32] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.

[33] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 3–18, New York, NY, USA, 2019. ACM.

[34] V. Ishakian, V. Muthusamy, and A. Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262, April 2018.

[35] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 445–451, New York, NY, USA, 2017. ACM.

[36] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[37] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[38] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[39] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[40] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. R. Butt. Bolt: Towards a scalable docker registry via hyperconvergence. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 358–366, July 2019.

[41] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li,

Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, Boston, MA, February 2019. USENIX Association.

[42] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[43] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, 2018. USENIX Association.

[44] T. G. Papaioannou, N. Bonvin, and K. Aberer. Scalia: An adaptive scheme for efficient multi-cloud storage. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, Nov 2012.

[45] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, 2019. USENIX Association.

[46] Krishna P.N. Puttaswamy, Thyaga Nandagopal, and Murali Kodialam. Frugal storage for cloud file systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 71–84, New York, NY, USA, 2012. ACM.

[47] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 401–417, Savannah, GA, November 2016. USENIX Association.

[48] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, February 2014. USENIX Association.

[49] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[50] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 16:1–16:15, New York, NY, USA, 2015. ACM.

[51] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. Spoton: A batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 329–341, New York, NY, USA, 2015. ACM.

[52] Tim Wagner. Serverless State: What comes next for serverless. In *ServerlessConf NYC'19*.

[53] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 620–634, New York, NY, USA, 2017. ACM.

[54] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.

[55] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. Zexpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[56] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 292–308, New York, NY, USA, 2013. ACM.

[57] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 543–557, Oakland, CA, May 2015. USENIX Association.

[58] Yinghao Yu, Renfei Huang, Wei Wang, Jun Zhang, and Khaled Ben Letaief. Sp-cache: Load-balanced, redundancy-free cluster caching with selective partition. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

Quiver: An Informed Storage Cache for Deep Learning

Abhishek Vijaya Kumar
Microsoft Research India

Muthian Sivathanu
Microsoft Research India

Abstract

We introduce Quiver, an informed storage cache for deep learning training (DLT) jobs in a cluster of GPUs. Quiver employs domain-specific intelligence within the caching layer, to achieve much higher efficiency compared to a generic storage cache. First, Quiver uses a secure hash-based addressing to transparently reuse cached data across multiple jobs and even multiple users operating on the same dataset. Second, by co-designing with the deep learning framework (e.g., PyTorch), Quiver employs a technique of substitutable cache hits *to get more value from the existing contents of the cache, thus avoiding cache thrashing when cache capacity is much smaller than the working set. Third, Quiver dynamically prioritizes cache allocation to jobs that benefit the most from the caching. With a prototype implementation in PyTorch, we show that Quiver can significantly improve throughput of deep learning workloads.*

1 Introduction

The more you know, the less (cache) you need.

- Australian proverb

Increasingly powerful compute accelerators, such as faster GPUs [33] and ASICs [17], have made the storage layer a potential bottleneck in deep learning training (DLT) jobs, as these jobs need to feed input training data fast enough to keep the compute units busy. For example, a popular benchmark for deep learning training is the ResNet50 [16] model on ImageNet data [14]. On 8 V100s, the training job can process 10,500 images/sec [6]. As each input image in ImageNet is roughly 200 KB, this translates to a storage bandwidth requirement of nearly 2 GB/s, to keep the GPUs busy. Newer, faster hardware (*e.g.*, TPUv3 [20], GraphCore [13]) will push this bandwidth requirement even higher, as their increased compute speeds require faster feeding of input data.

While such bandwidth requirements are challenging in their own right, three aspects of deep learning training (DLT) jobs exacerbate this problem.

First, for hyper-parameter tuning, users typically run tens or hundreds of instances of the same job, each with a different configuration of the model (*e.g.*, learning rate, loss function, *etc.*). In such a scenario, the same underlying store must service reads from several such jobs, each reading in a different random order, placing significantly higher bandwidth demand on the store.

Second, data sizes for input training data in deep learning jobs have been increasing at a rapid pace. While the 1M ImageNet corpus is hundreds of GB in size (the full corpus is 14x larger), newer data sources that are gaining popularity are much larger. For example, the youtube-8M dataset used in video models, is about 1.53 TB for just frame-level features [12], while the Google OpenImages dataset [10], a subset of which is used in the Open Images Challenge [11], has a total size of roughly 18 TB for the full data [10].

Third, the most common mode of running deep learning jobs is by renting GPU VMs on the cloud (partly because of the high cost of GPUs); such VMs have limited local SSD capacity (*e.g.*, most Azure GPU series VMs have a local SSD of 1.5 to 3TB). Further, local SSDs are "ephemeral" across VM migrations. Pre-emptible VMs are provided by cloud providers at a significantly lower cost (6-8x cheaper) compared to dedicated VMs [1, 22]; such VMs running DLT jobs may be preempted at any time, and resume from a checkpoint on a different VM [3], losing all local SSD state. As a result, users keep input training data in reliable persistent cloud storage (*e.g.*, in a managed disk or a data blob) within the same data center region, and access the store remotely from GPU VMs that run the training job. Egress bandwidth from the store to the compute VMs is usually a constrained resource, especially when several VMs read from the same storage blob.

In this paper, we present *Quiver*, a storage management solution for supporting the I/O bandwidth requirements of DLT jobs in the above setting where the input training data resides in a cloud storage system, and the DLT jobs are run in a few GPU VMs in the cloud (typically in the same datacenter region). *Quiver* is an intelligent, distributed cache for input training data, that is shared across multiple jobs or even across multiple users, in a secure manner without leaking data.

The key insight in *Quiver* is to fundamentally improve cache efficacy by exploiting several characteristics of the DLT workflow that simplify storage management. First, deep learning datasets are *head-heavy*. A few popular input datasets (such as ImageNet) are used across numerous DLT jobs and across several model architectures. Even in companies, different members of a team may be iterating on different alternative ideas, all of which target the same end-to-end problem/dataset such as web search. Second, each DLT job runs

multiple (typically 50-100) *epochs* of training, with each epoch consuming the entire training data once (in a random permutation order). These two characteristics make the workload very cache-friendly. However, when the datasets are too large to fit in a single VM, the cache needs to span multiple VMs (possibly across multiple users in the organization) in a secure manner.

Another key observation that *Quiver* exploits is that the data read by DLT jobs is *transparently substitutable*. A single epoch of a DLT job consumes the entire training data in a random permutation order, and as long as the order is random and the entire data is consumed exactly once, the exact sequence in which inputs are read does not matter. This allows *Quiver* to provide *thrash-free caching*, a powerful technique in scenarios when the available cache capacity is much smaller than the working set size; such a scenario usually makes the cache ineffective because of thrashing. In contrast, *Quiver* allows a small slice of the dataset to be cached and shared efficiently in a decentralized manner across multiple jobs accessing that dataset, without causing thrashing.

Sharing a cache across multiple users, where each user may have private training data (perhaps to augment standard datasets) needs to preserve privacy so that training data of one user is not leaked to another. The need for isolation of data conflicts with reuse of cache across shared data. To bridge this, *Quiver* does secure content-based indexing of the cache, so that the cache is reused even across different physical datasets with the same content (*e.g.*, multiple copies/blobs of ImageNet data). The content-hash of a data item (or a group of data items) is used to address the cache. A digest file available with the DLT job of a particular user contains the hashes of individual data items in the dataset; the very possession of a valid content hash serves as a *capability* to access that data item, thus providing a simple yet effective form of access control similar to those explored in content-addressed filesystems [9, 23].

An important signal that *Quiver* uses to prioritize cache placement and eviction, is the effective *user-perceived benefit* from caching, for every DLT job. Whether a DLT job needs a cache is primarily a function of two factors: (a) the remote storage bandwidth (b) amount of compute per byte of data read by the job from the store. A DLT job with a very deep model would perform lot of computation per input, and thus the I/O time can be hidden/pipelined behind compute time even if the storage bandwidth is low, and vice versa. Further, some jobs may overlap I/O and computation with pipelining, while some may perform I/O synchronously. Thus, modelling the sensitivity of a DLT job's performance to caching is not straightforward. *Quiver* simplifies this by exploiting the predictability of DLT jobs across mini-batches [31, 37], and uses controlled probing to measure the time for a fixed number of mini-batches, with and without caching. The difference in performance between these two modes is an accurate empirical metric of how much a particular DLT job benefits from caching, and is used in eviction and placement decisions.

We have implemented *Quiver* as a dynamic distributed cache shared across multiple users and jobs running within a cluster of GPUs, and integrated it with the PyTorch deep learning toolkit [25]. *Quiver* has three components: a cache server that runs in a separate container under a dedicated user, a cache manager that co-ordinates actions across multiple cache servers, and a cache client that runs in the same container as every DLT job accessing the cache; in fact the client is integrated with the PyTorch data input layer. Such tight integration allows *Quiver* to exploit intricate knowledge of the specific DLT job.

We have evaluated *Quiver* in a cluster of 48 GPUs, across a multi-user, multi-job workload. We demonstrate that *Quiver* speeds up DLT jobs by up to 3.8x and improves overall cluster throughput by up to 2.1x under a mixed workload. We also show that the substitutable cache hits feature of *Quiver* avoids cache thrashing with a small cache, allowing jobs to make good use of a fractional cache, and that benefit-aware cache prioritization improves overall cluster efficiency by allocating constrained cache space wisely.

This paper makes the following key contributions:

- We characterize the I/O behavior of deep learning training jobs, and identify various key characteristics such as substitutability, predictability, and shareability.

- We provide the first storage solution that allows DLT jobs to get much higher effective I/O bandwidth for training data reads, by using a distributed cache that is shared across multiple users in a secure manner.

- We identify and implement several efficiency improvements to the cache layer by exploiting intricate knowledge about how sensitive a job is to I/O performance, to prioritize cache eviction and placement.

- We provide a novel thrash-proof caching strategy that exploits the substitutability of a DLT job's I/O requests, and provide a way for multiple jobs to access a shared slice of a data set in an efficient manner.

- We demonstrate the efficacy of our techniques and policies with a real implementation and empirical evaluation on a cluster of 48 GPUs.

The rest of the paper is structured as follows. We present a brief background of DLT jobs in Section 2. In Section 3, we present various key characteristics of DLT jobs from an I/O perspective. We present the design of *Quiver* in Section 4, and provide more detail on the cache management policies in Section 5. We discuss the implementation of *Quiver* in Section 6, and evaluate it in Section 7. Finally, we present related work in Section 8, and conclude in Section 9.

2 Background

A deep learning training (DLT) job takes training data as input, and learns a model that represents the training data.

To perform the learning, a DLT job takes a small *random* sample *i.e.*, a *mini-batch* of input items at a time (typically 32 to 512 items), and uses stochastic gradient descent [29] to slowly learn the parameters such that the prediction loss is minimized. Each mini-batch is compute-intensive (mostly involving multiplications of large matrices/tensors) and runs on accelerators such as GPUs. Because each mini-batch runs the same computation on inputs that have the same *shape*, all mini-batches take identical time on the GPU [31, 37].

Input training data: Training data, at a high level, is a list of tuples of the form $< input, label >$, where input is an image or speech sample or text to be fed to the neural network, and label is the ground truth of what the network should learn to classify that input as. Training large networks such as ResNet50 [16] or GNMT [36] requires millions of training examples. For example, ImageNet-1M, a popular training data for image classification, has 1 million images (the full dataset has 14 million), each of which can be about 200 KB in size. Recent datasets such as youtube-8m [12] and OpenImages [10] are several terabytes in size as well.

To feed input items in a randomized order, DLT frameworks such as PyTorch use the notion of *input indices* to access the training data. For example, if the training data has a million items, they track a list of indices to each of these items, and then randomly permute this list. They then perform random access on the store to fetch the data items corresponding to fixed number (*i.e.*, the mini-batch size) of these indices. An *epoch* of training completes when all these indices are exhausted, *i.e.*, the model has looked at all data items once. For the next epoch, the list of indices is again randomly permuted, so that different set of mini-batches get fed to the network. A DLT job typically runs several epochs, ranging from 50 to 200.

Transformations: Input data items read from the store are then *transformed* by the DLT framework. Typical transformations include decompression of the image to convert from say, jpg format to a pixel format, applying various forms of augmentation (scaling, rotation, etc.). These transformations are usually CPU intensive.

Multi-jobs: Because of the trial-and-error nature of DLT experimentation [28, 37], users often run multiple instances of the same model simultaneously, each with different configurations of parameters such as learning rate. Each of these jobs would access the entire training data, but in different random orders.

3 IO Characteristics of DLT

In this section, we describe the key characteristics of DLT jobs from an I/O access perspective.

1. Shareability: There is a high degree of overlap in I/Os performed by a DLT job, both within and across jobs. Within a job, as each job makes multiple passes over the same input training data (*i.e.*, multiple epochs), there is a clear benefit to caching the data for use in subsequent epochs. More importantly, there is also extensive inter-job sharing, because of two reasons. First, with hyper-parameter exploration, a *multi-job* [37] may have several jobs running different configurations of the same model, operating on the same data. These jobs may be running on different machines, but access the same underlying data on cloud storage. Second, the input training datasets that DLT jobs use are quite head-heavy; popular datasets (*e.g.*, ImageNet) are used in several jobs. Even in enterprise settings, multiple team members work to improve accuracy on a dataset (*e.g.*, web search click-data), each running a different model. There is hence a significant inter-job reuse.

2. Random Access: While shareability seems to make DLT jobs extremely cache-friendly, it is only true if the *whole* input training data can fit in cache. Otherwise, the random access pattern (different permutation each epoch) makes it cache-unfriendly (in fact, adversarial) for *partially cached* data. A cache that can hold say 20% of the training data would simply thrash because of random I/O.

Partial caching of DLT training data is important, because training data size for several datasets are already large, and only getting larger as models get bigger. For example, the youtube-8M dataset used in video models, is about 1.53 TB for just frame-level features [12], while the Google OpenImages dataset, a subset of which is used in the Open Images Challenge [11], has a total size of roughly 18 TB for the full data [10]. Even the full ImageNet corpus of the entire 14 million images is several terabytes in size. Further, a single server in a GPU cluster often runs multiple jobs, or even time-slices across several jobs [37]. As each of these jobs could be accessing different data sets, the local cache may be contended across multiple datasets. So, getting useful performance out of the cache under partial caching is important for DLT jobs.

3. Substitutability: Fortunately, another trait of DLT jobs helps address the challenge posed by random access. From an I/O perspective, an epoch of a DLT job only requires two properties to hold: (a) each input data item must be touched exactly once; and (b) a random sample of inputs must be chosen for each mini-batch. Interestingly, the *exact* sequence of data items does not matter for the correctness or accuracy of the job, which means the I/O is *substitutable*; instead of seeking for particular files, the DLT job can now ask for *some random subset* that was not already accessed. From a caching perspective, this is a unique property that can significantly help with cache reuse. With substitutability, even a small cache for say 20% of the training data can provide good caching performance, because if an item is not in the cache, we could return a substitute item from the cache that preserves the randomness and uniqueness properties. As we show in § 7, substitutable caching does not impact the final accuracy of the learning job.

4. Predictability: Another favorable property of DLT jobs is their predictability across mini-batches [31, 37]. Because

the time per mini-batch is known in advance, one can predict how sensitive each job is to I/O performance, which can in turn allow the cache placement and eviction to give higher priority to jobs that benefit the most from caching.

4 Design of *Quiver*

In this section, we present the design of *Quiver*, a distributed cache that improves I/O efficiency for DLT jobs running in a cluster of GPUs. *Quiver* is a *co-designed* cache that is tightly coupled with the DLT framework (*e.g.*, PyTorch or Tensorflow). By modifying the DLT framework, the cache client integrates deeply into the I/O access of the DLT job, and shares richer information with the cache server.

4.1 System Architecture

Before getting into the details of *Quiver*, we describe the broader context in which *Quiver* fits. *Quiver* is designed for a shared GPU cluster that an organization creates on GPU VMs allocated in the cloud. Each GPU VM has a certain amount of local SSD storage. A DLT job runs within its own container, thus isolating jobs of multiple users from each other. A higher level scheduler and container manager such as Kubernetes [7] manages submission of jobs and scheduling of DLT job containers on a specific VM. *Quiver* is agnostic to the exact scheduling mechanism used, and makes no assumptions about the scheduler.

The input training data for jobs of a particular user is stored in the cloud storage account of the corresponding user, which ensures privacy and access control; while general datasets such as ImageNet do not require this, in general, users sometimes augment standard datasets with their own private training samples which may be sensitive, or train on entirely private data (*e.g.*, surveillance videos, enterprise data). The DLT job running in a VM would perform random reads on this remote storage (*e.g.*, an Azure blob [21] or Amazon S3 [2]).

A DLT job may move across VMs, because of VM redeployments, because of job migration [37], or because it runs on cheaper preemptible VMs [1, 3, 22]. Hence, the local SSD data at each VM is *soft state*. Thus, even if the whole training dataset fits in one VM's local SSD, the simple solution of copying data once from the remote store to local SSD does not work. With *Quiver*, a job can move around across VMs and still transparently benefit from a shared distributed cache.

4.2 Security model

Quiver is a cache that is shared across multiple jobs and multiple users, so the security semantics are important. *Quiver* guarantees that a user can see only data content that she has access to otherwise (*i.e.*, no training data is leaked across multiple users). This requirement of *data isolation* conflicts with the need to share/reuse the cache for effective performance.
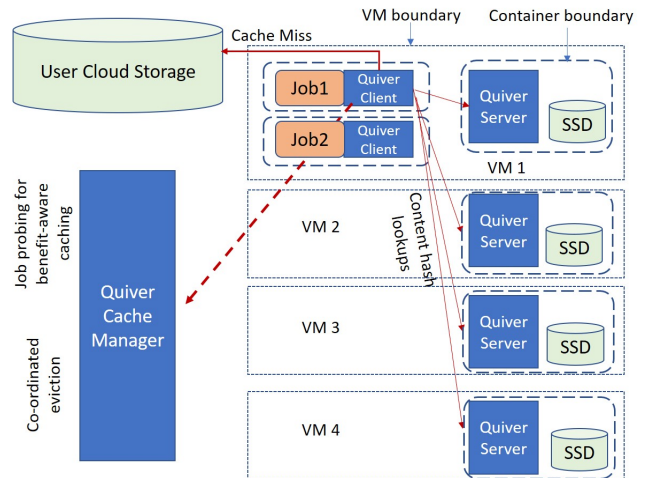

Figure 1: **Architecture of Quiver** Cache servers run on all VMs, but within their own container. *Quiver* clients run in the address space of the DLT job, and include changes to the DLT framework. User's input training data is fetched from cloud storage on a cache miss. Each job's dataset is sharded across multiple cache servers, and looked up using content hashes.

For example, if two different users have their own copies of the ImageNet dataset, those would be two different sets of files in two different cloud storage accounts and thus would logically need to be cached separately, thus preventing reuse across users. *Quiver* uses *content-addressed capabilities* to achieve cache reuse while preserving isolation.

4.3 Content-addressed Cache

The cache in *Quiver* is addressed not by file names and offsets, but by content hashes, similar to content-addressed file systems [9, 23]. The granularity of a cache entry in *Quiver* is decided by the DLT job, and it could be either an individual data item (*e.g.*, image training data where each item is hundreds of KB) or a group of data items (*e.g.*, in text training data). For simplicity, we assume in this paper that the granularity is a single data item. For each data item in the dataset, a content hash (*e.g.*, SHA1) of that item is calculated, and the resulting hash acts as the index for cache inserts and lookups. The benefit of content addressing is that the same data items across multiple copies (say copies of ImageNet data in different storage accounts of different users), will map to the same hash, allowing reuse across users.

To ensure isolation, *Quiver* uses the notion of digest files for the input training data. For each dataset that a user owns, the user computes the content hashes of each data item, and stores just the hashes in a digest file. The digest file contains entries of the form <content_hash: file_location>, where the file_location indicates the path and offset where that particular data item resides within the cloud storage account of

this particular user. Thus, across multiple users sharing the same data set, while the hash component would be the same, each user will have a different entry in the `file_location` component, as they would point to that particular user's backing store in the cloud. Because the DLT job is calculating these hash digests only from data that the user already has access to, the very presence of a hash value serves as a capability for that user to access that content. As a result, when a cache server gets a lookup for a certain hash value, it can safety return the data associated with that key. The user cannot manufacture or guess legal hashes without having the content, because of the sparsity of the hash function and its collision-resistance properties.

As the digest file is small (few MBs), it is stored locally within the container of the DLT job. The DLT job first looks up the cache with the hash capabilities. If the content is not in the cache, it fetches it from remote storage (using the `file_location` corresponding to the hash entry), and then adds that content into the cache keyed by the hash value.

4.4 *Quiver* Server

The cache server in *Quiver* is a distributed, peer-to-peer service that runs on all GPU VMs in the cluster. The cache server runs as a separate "privileged" user (*e.g.*, organization admin) in its own container, so other users running DLT jobs do not have access to that container. DLT jobs interact with the cache server through RPC interfaces for `lookup` and `insert`. Internally, the cache server is a key-value store maintained on local SSD. The key space is partitioned across multiple cache-server instances via consistent hashing; each cache server instance handles its partition of the key space.

4.5 Cache Manager

Because *Quiver* is a distributed cache, it needs to co-ordinate eviction and placement decisions so that all cache servers roughly agree on the which parts of which data sets to cache. The cache manager in *Quiver* interacts with both the *Quiver* clients and *Quiver* servers to co-ordinate these decisions. The cache manager is also responsible for measuring the likely benefit that each job would get from caching, by probing DLT jobs. It does this by instructing cache servers to temporarily return cache misses for all data read by the DLT job for a few mini-batches. It then compares this execution time with the time during normal operation with caching, and uses this to prioritize cache placement (§ 5).

4.6 *Quiver* Client

A significant part of the intelligence in *Quiver* exists at the cache client. The cache client runs as part of the DLT job within the user's container, in the same address space as the DLT framework such as PyTorch, and interposes at the interface used by the DLT script to access training data. For example, in PyTorch, the `DataSet` abstraction is used to iterate over training data, and it has a simple `Next` interface to get the next set of input data items for the next mini-batch. Internally, the `DataSet` abstraction maintains a randomly permuted list of indices that determines the order in which the data items are fetched. *Quiver* augments this component to also manage the digest-file of hashes, and when a set of indices are to be fetched from the store, it first does the lookup in the cache using the hash values in the digest.

In addition, the *Quiver* client also exports job-specific information to the cache servers, such as the time taken per mini-batch on the GPU. This allows the cache servers in *Quiver* to probe and perform a controlled measurement of performance of the DLT job with and without caching, and use that to prioritize cache placement.

4.7 Substitutable hits

Quiver incorporates the notion of substitutable I/O into the data fetch component of the DLT framework. Today, if a mini-batch requires 512 data items, the dataset loader provides 512 indices to be fetched from the store; if only data pertaining to a subset of the indices was cached, some items may be missing, resulting in remote I/O in the critical path. In *Quiver*, the loader looks up lot more (*e.g.*, 10x) indices from the cache and fills the mini-batch opportunistically with whichever 512 it is able to fetch from the cache, so that the DLT job can make progress without blocking on the cache misses. It then marks the indices that missed in cache as "pending". The data loader continues with the remaining indices for subsequent mini-batches. Once it reaches the end of that list, it makes additional passes over the index list, this time focusing only on the indices previously marked pending.

To see why this would work, assume that only 10% of the training dataset is in cache (for simplicity, a contiguous 10% in the original data set order *i.e.*, without any random permutation). Now, because the lookups from the DLT job are a randomly permuted order of indices, each sequence of k indices is *expected* to get cache hits for $k/10$ indices; hence, if it looks up a sequence of length $10 * k$, it can fill its mini-batch of size k. During its second pass over the pending entries, a different, non-overlapping 10% of the dataset may be in the cache, which means it would get hits for $1/9th$ of the lookups. Note that this property also holds across multiple jobs each with their own random permutations. For the same 10% that is cached, regardless of the permutation each job has, each job is expected to get hits for $1/10th$ of its lookups. Thus, multiple jobs can proceed at full-cache-hit speeds although each of them is accessing a completely different random permutation. Such a workload would normally cause thrashing on a small cache that contains only 10% of the data items. With substitutable cache hits, we prevent thrashing and provide cache-hit

performance. Of course, this assumes an intelligent cache eviction policy, which we describe in § 5.

Impact on Accuracy: A natural question that arises with substitutable hits is whether it impacts training accuracy. As we show in § 7 across multiple models, substitutable hits do not affect accuracy of the job, as the randomness within a reasonable fraction of the training data (*e.g.*, 10%) is sufficient.

4.8   Failure recovery

The substitutability property also helps mask failures of cache servers, such as due to VMs going away. In a traditional cache, failure of a cache server would cause a spike in miss traffic to fetch the lost cache items from the store. *Quiver* can handle it gracefully by simply returning substitute data items, while fetching the contents of the failed cache server in the background. The DLT jobs do not incur the miss handling cost in the critical path; they just continue with whatever data is available in the live cache servers; a subsequent pass over the list of indices will use the re-populated data.

4.9   Locality of cache servers

While the simple version of *Quiver* (focus of this paper) has a unified cache spread across all VMs, the *Quiver* design also permits a locality-aware cache layout. For example, datasets used by VMs within a rack in the data center (or a top-level switch) could be cached only within other VMs under the same switch, so that most fetches avoid the over-subscribed cross-rack switches. In such a setting, each rack would have its own logical *Quiver* instance with its own cache manager. *Quiver* can thus also help save cost for the cloud provider by reducing cross-rack network traffic.

5   Cache Management

In this section, we describe various aspects of cache management in *Quiver*.

5.1   Co-ordinated eviction

As described in § 4.7, when only a part of the dataset (say 10%) is cached, *Quiver* does multiple passes over the list of permuted indices of the dataset within a single epoch. To get good hit-rate during the second pass, a *different* part of the dataset must be cached during that second pass. In a scenario where multiple DLT jobs (*e.g.*, a multi-job doing hyper-parameter exploration) are accessing the same dataset, this is tricky because different jobs may exhaust their first pass over the list of permuted indices at different times.

Quiver handles this by allocating cache space for two *chunks* of the data set, and using a technique similar to double-buffering [35]. First, the digest file representing the complete dataset, is partitioned into a fixed number of *chunks*, such

that each chunk is, say, 10% of the dataset. The chunking of the dataset has to be done intelligently, to ensure randomness of the input data within each chunk. Some datasets such as LibriSpeech [24] order data items by the sequence length; chunking them in logical byte order would result in the first chunk comprising entirely of short sequences, thus affecting randomness. Recurrent neural networks (RNNs) [4, 36] require all inputs within a mini-batch to be of the same sequence length; if a mini-batch comprises of inputs with different sequence lengths (*e.g.*, randomly chosen inputs), they pad all inputs to match the length of the longest input within the mini-batch. Thus, for compute efficiency, it makes sense for all inputs within the mini-batch to be roughly of the same length.[1]. To allow for such efficient bucketing of inputs within a mini-batch, we define the chunk to be a *striped partition*; let us refer to each contiguous 10% of the input dataset as a *partition*. Each partition is chunked into 10 *stripe units*; a logical chunk is simply the complete stripe formed by stitching the corresponding stripe unit within each partition. As much as possible, a mini-batch is formed purely from inputs in a single stripe unit, for homogeneity of sequence lengths, while also ensuring uniform distribution of inputs.

Dataset chunking allows co-ordinated access of the cache across multiple jobs. While the jobs operate on the first chunk, the second chunk is brought into the cache, so that it is ready when (some of) the jobs switch to the next pass, possibly in a staggered manner. An important question is when to evict the first chunk from the cache. If evicted too soon, a subset of jobs that are still in their first pass and accessing that chunk will see misses, whereas if it remains in the cache for too long, the next (third) chunk cannot be preloaded. *Quiver* uses a two-step process to handle eviction. A chunk is *marked for eviction* when another chunk of the dataset is fully loaded into cache; all new jobs will now get hits only from the latest chunk. However, existing jobs that are still running their pass over the first chunk, will continue to get hits on the first chunk. When all existing jobs have exhausted their pass over the first chunk (and notify the cache server), the first chunk is *actually evicted*. At this point, the preload for the third chunk of the data set can start.

In the above example, note that if a job proceeds at a much slower rate compared to other jobs accessing the same dataset, it could continue to access the first chunk for a long time, preventing the load of the third chunk into the cache. Different jobs in a multi-job are typically designed to proceed at a similar pace, so this is not a common occurrence within a multi-job, but could happen across very different models on the same dataset. Interestingly, a job that is much slower than other jobs on the same dataset means that it spends more time per mini-batch on the GPU, which means it is less sensitive to I/O performance(§ 5.3); a cache miss would not affect that

---

[1]Dynamic graph computation in modern frameworks such as PyTorch [25] ensures that a mini-batch with short sequence length uses correspondingly lesser computation

job by much. Hence, *Quiver* does a forced-eviction of a chunk after a threshold time has expired from the completion of the first job on that chunk.

Algorithm 1 Substitutable hits & Co-operative miss handling

 1: global **gChunkIndex** = -1
 2: ▷ Returns: List of indices of data items to be fetched for current mini-batch
 3: **function** GETBATCH(SIZE)
 4:     ▷ Try to randomly sample 10 x size unused elements
 5:     pendingIndices = **getPendingIndices**(size * 10)
 6:     cacheHits = **cacheClient.lookup**(pendingIndices)
 7:     **if** len(cacheHits) >= size **then**
 8:         return **pickAndMarkUsed**(cacheHits, size)
 9:     **end if**
10:     ▷ Not enough cache hits, perform co-operative
11:     ▷ cache miss handling
12:     result = List()
13:     **result.addAll**(
            pickAndMarkUsed(cacheHits, len(cacheHits)))

14:     **if** gChunkIndex < 0 **then**
15:         ▷ cacheManager returns 0 if no chunk is cached
16:         gChunkIndex =
            cacheManager.getCurrentChunk(datasetId)
17:     **end if**
18:     chunksChecked=0
19:     **while** chunksChecked < totalChunks **do**
20:         ▷ Tell cache servers that I am using this chunk
21:         ▷ (if not done already)
22:         **informServers**(jobId, datasetId, gChunkIndex)
23:         unusedIndices = **getRandomUnusedIndices** (
            gChunkIndex, size - len(result))
24:         **if** len(unusedIndices) == 0 **then**
25:             **informServersDoneUsingChunk**(
                jobId, datasetId, gChunkIndex)
26:         **end if**
27:         result.append(unusedIndices)
28:         **if** len(result) == size **then**
29:             return result
30:         **end if**
31:         gChunkIndex =
            (gChunkIndex + 1) % totalChunks
32:         ++chunksChecked
33:     **end while**
34: **end function**

5.2   Co-operative cache miss handling

A common workload that places significant demand on the storage bandwidth, is a multi-job [37] where a DLT user runs tens or hundreds of jobs for the same model on the same dataset, but with different hyper-parameter configurations. Without *Quiver*, each of these jobs will read the same

data from the remote store, causing the remote store to become a bottleneck, resulting in poor I/O throughput per job. *Quiver* uses co-operative miss handling, where it *shards* the cache fetches across multiple jobs, to avoid multiple fetches of the same data items by multiple jobs. This sharding is done implicitly by simply randomizing the order of fetch of missing files, thus avoiding direct co-ordination among the (independent) jobs. Thus, each job first checks the cache if a set of (say 2048) data items exist, then reads a random subset of those items, and adds the read items into the cache. After the additions, it performs another cache lookup, but this time it would get hits for not only the data items it added, but also the other (mostly non-overlapping) data items that were added simultaneously by other jobs that performed a similar random fetch. Thus, even in the case of a cold cache, or if the entire dataset cannot fit in cache, *Quiver* provides benefits by conserving remote store bandwidth, reading most data items only once across multiple jobs within a single epoch.

A high-level algorithm for substitutable cache hits and co-operative miss handling is presented in Algorithm 1.

5.3   Benefit-aware Cache placement

When total cache space is constrained, *Quiver* utilizes job heterogeneity to preferentially allocate cache space to the jobs that benefit the most from the cache. A DLT job performs both compute (on the GPU) and I/O. Intuitively, if the compute time is higher than the I/O time to read from the remote store, the I/O time can be overlapped, and the job performance would be the same whether it reads from the cache or from the remote store. However, this is a complex phenomenon to model, because it depends on the degree of parallelism of the job (*i.e.*, number of GPUs it runs on), how large the model is, whether the model is written in a way to pipeline computation and I/O, etc.

Interestingly, the tight integration with the DLT framework allows *Quiver* to intelligently probe and measure the job's performance with and without caching. When a new job requests for adding entries into the cache, the cache manager picks the job for probing. Probing operates in two steps. In the first step, the cache manager instructs all cacheservers to reject all cache lookups for that job, thus forcing the job to fetch from the remote store. At the end of this probing phase, *e.g.*, 100 mini-batches, the cache manager gets the total elapsed time from the cache client (which runs as part of the DLT job). The cache manager then monitors the job's performance periodically with the default caching policy. If the times with the default caching policy and without caching don't differ by much, it concludes that the job is not bottlenecked on remote I/O bandwidth, and decides to turn off caching for that job. A dataset touched only by such jobs would thus never enter the cache, freeing up space for other datasets that benefit job performance. *Quiver* runs the probing phase not only at job start time, but periodically , as effective I/O throughput may

have reduced because of increased load on the remote store (*e.g.*, newer jobs reading from the same store), thus making the job more sensitive to I/O performance, or vice versa.

Let t_h^i be the average per-mini-batch time for job i under cache hit, and t_m^i be the corresponding time under cache miss. The benefit from caching for job i is thus $b^i = t_m^i / t_h^i$. Let n^i be the number of GPUs taken by job i. The GPU resources saved for job i by caching its dataset is thus $g^i = b^i * n^i$.

For each data set D^k, there could be multiple jobs in the cluster accessing the same data set. Because the cache is shared by all such jobs, if N jobs access D^k, the total GPU resources saved by caching the dataset is $G_{D^k} = \sum_{i=0}^{N} g^i$. Interestingly, the cache manager has to decide only among three options for each data set: (a) fully cache (space cost is the full size of the dataset (b) enable co-operative miss by caching a fixed size chunk (*e.g.*, 15G, or 10% dataset whichever is smaller (cost is 2 chunks for double buffering), or (c) no caching (zero cost). Note that intermediate sizes for caching are useless, as the benefits are the same as with caching two chunks, given the substitutable cache-hits in *Quiver*.

Given a total cluster-wide cache space budget of S, the cache manager uses a greedy algorithm to preferentially assign cache space to datasets or dataset chunks with the highest ratio of benefit-to-cost.

5.4 Cache sharing scenarios

Quiver transparently benefits a variety of DLT scenarios:
Single job accessing a dataset: If the entire dataset can fit in cache, *Quiver* caches the data items accessed in the first epoch of the DLT job. As the DLT job runs several epochs over the same data, subsequent epochs get full cache-hits from *Quiver*, and run faster. If the dataset does not fit in cache, the DLT job does not benefit from *Quiver* as it reads from remote store in the steady state.
A multi-job accessing a single dataset: A multi-job is a set of jobs run by the same user on the same dataset, but with different configurations of hyperparameters [37]. Today, each jobs reads the same content in different random orders from remote storage. With *Quiver*, if the data fits in cache, all jobs share the cache and get full cache-hits. Interestingly, even if only 10% of the data fits in cache, *Quiver* still gives better performance, because it shards the reads across jobs with co-operative miss handling (§ 5.2).
Different jobs accessing the same dataset: Another scenario that *Quiver* benefits is opportunistic sharing of popular datasets *across* jobs even from multiple users. By doing so, *Quiver* extracts more value out of the same SSD space especially for popular datasets such as ImageNet.

6 Implementation

The *Quiver* client is implemented in PyTorch 1.1.0 (about 900 LOC). Pytorch's data model consists of three abstractions:

Config	Top-1 Acc. (%)	Top-5 Acc. (%)
Baseline sampling	75.87	92.82
Quiver sampling	75.89	92.76

Table 1: **ResNet50 on ImageNet: Final Accuracy after 90 epochs (higher is better)** Average of two runs.

Config	Word error rate (WER) (%)
Baseline sampling	22.29
Quiver sampling	22.32

Table 2: **Accuracy of DeepSpeech2 on LibriSpeech: Final WER (*lower* is better)** Average of two runs (30 epochs).

Dataset, Sampler, and DataLoader. Dataset returns a data item corresponding to a given *index*. Sampler creates random permutations of indices in the range of dataset length. Dataloader fetches one mini-batch worth of indices from the sampler, and adds these to the index queue. The worker threads of DataLoader consume these indices, and fetch data items from Dataset. To use *Quiver*, instead of `torch.utils.Dataset`, the model must use `QuiverDataset` (same interface as existing `Dataset`), that handles the digest file containing hashes. Similarly, the model must extend from `QuiverDataLoader` (same interface as standard `DataLoader`), that probes and monitors the job's mini-batch progress in the `__next__` routine; it also ignores the default Sampler passed into the DataLoader API, instead using its custom Sampler that handles substitutable hits, by creating a list of hashes from indices sampled from chunks of the dataset.

The cache client uses RPC endpoints to look up the cache using hashes, fetch files from cache, and finally, to write to cache and communicate mini batch times to the cache manager. Data fetch from Azure blob on the cache miss path happens over a regular TCP socket connection. QuiverDataset uses either the cache client or the blob client depending on whether it is looking up the cache, or filling a cache miss.

The *Quiver* server is a network server written in C++ in about 1200 lines of code. In addition to batched interfaces for lookup/insert on cache, the server also exposes interfaces to get the current active chunk and notify "ref_chunk" and "unref_chunk"; the cache client uses these to assist with co-ordinated eviction at the server. The server also exposes an interface to set the *caching mode*, used by the cache manager, *e.g.*, to disable caching for a job during probe phase.

The cache manager is a simple python server with an RPC endpoint used by the client to report mini-batch times, and it informs the cache servers which datasets to cache in which mode, based on its benefit-aware allocation decisions.

7 Evaluation

In this section, we evaluate *Quiver* along several dimensions. We answer the following questions in the evaluation:
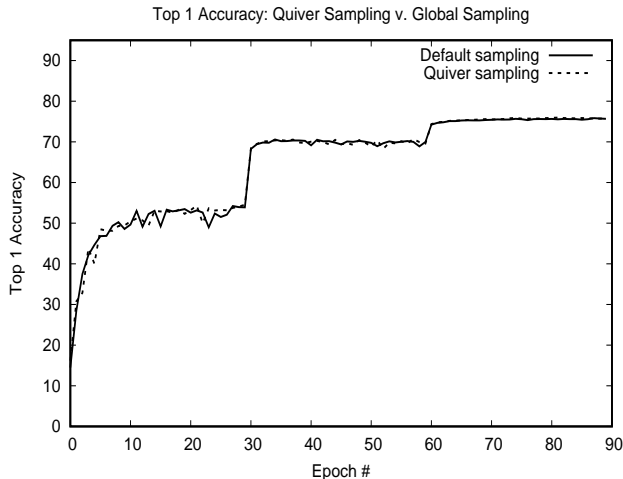

Figure 2: Top-1 Accuracy of ResNet50 ImageNet model under globally random sampling and chunked sampling.

- Do substitutable cache hits impact learning accuracy?
- How much does *Quiver* speed up different DLT jobs?
- How effective is co-ordinated eviction in *Quiver*?
- How effective is benefit-aware caching in *Quiver*?

7.1 Experimental setup

For our evaluation, we use a cluster of 48 GPUs across 12 VMs on Azure. 6 VMs contain 4 NVidia P100 GPUs each while the other 6 contain 4 NVidia P40 GPUs each. All VMs contain 3 TB of local SSD. Different experiments use a subset of these VMs or the full set. The input datasets reside in Azure storage blobs [21] within the same region. We use a diverse set of deep learning workloads: ResNet50 [16] on the 154 GB ImageNet dataset [14], Inception v3 [32] on the 531 GB OpenImages dataset [10], and DeepSpeech2 [4] on the 90 GB LibriSpeech dataset [24]. For substitutable caching, we use a fixed chunk-size of 15GB.

7.2 Accuracy with substitutability

We first show that substitutable caching in *Quiver* (*i.e.*, restricting the shuffle to a fraction of the dataset rather than the entire dataset) has no impact on accuracy. As can be seen from Figure 2, the top-1 accuracy curves closely match. Table 1 shows the final top-1 and top-5 accuracies in both configurations; *Quiver* sampling achieves the same accuracy as globally random sampling. Table 2 shows results for the DeepSpeech2 model on LibriSpeech dataset. Again, the chunked sampling of *Quiver* converges to a similar word-error-rate compared to globally random sampling.

Workload	Time for 7000 mini-batches (s)		
	Baseline	*Quiver*	
	Cache Miss	Cache Hit	Co-op. Miss
ResNet50	2505	646 (3.88x)	1064 (2.35x)
Inception	2874	1274 (2.26x)	1817 (1.58x)
DeepSpeech	1614	1234 (1.31x)	1265 (1.28x)

Table 3: **Speedups from *Quiver* across three workloads**

7.3 Improvement in job throughput

We now evaluate the performance gains from *Quiver*, on three different workloads: ResNet50, Inception, and DeepSpeech2. In each workload, we run a multi-job on 28 GPUs. Recall that a multi-job runs multiple hyper-parameter configurations of the same model/job. For each multi-job, we run 7 jobs (of different configurations), where each job runs on 4 GPUs in a single VM. We show the aggregate throughput (mini-batches/second) of the multi-jobs under three configurations:

1. The baseline configuration, where all jobs read from the remote storage blob. This configuration is referred to as "Cache miss" in the graphs;

2. When all data fetches result in cache hits in *Quiver*. This is the best case performance with *Quiver*, and is shown as "Cache hit" in the graphs;

3. When *Quiver* starts with a cold cache, and the DLT jobs perform co-operative cache miss handling to avoid redundant I/O on the remote store. This also represents the performance when only a 10% or 20% slice of the dataset is cached(§ 4.7).

Figure 3 shows the results for the three workloads. As can be seen, the slope of the "cache hit" curve is consistently much less compared to the "cache miss" curve. In other words, the same number of mini-batches are processed much faster with *Quiver*, resulting in better efficiency. The "co-operative miss" curve is in between the cache hit and cache miss configurations. Thus, even when starting with a cold cache, the ability of *Quiver* to avoid redundant I/O to the remote store from all 7 VMs allows it to extract much higher useful bandwidth out of the remote storage, resulting in better efficiency. Interestingly, in Figure 3(c), the difference between co-operative miss and cache hit is minor, indicating that the workload can run equally fast with just a small slice of the cache (§ 5.3). The overall speedups achieved by *Quiver* for the three workloads is shown in Table 3.

7.4 Interaction with I/O pipelining

DLT frameworks including PyTorch pipeline I/O with computation, to hide I/O latency. In particular, the data loader maintains a queue of fetched mini-batch inputs, and the computation engine picks from the in-memory queue. Both baseline and *Quiver* benefit from pipelining, so the benefits from *Quiver* shown in the previous subsection are *in addition to*

(a) ResNet50/ImageNet   (b) Inception/OpenImages   (c) DeepSpeech2/LibriSpeech
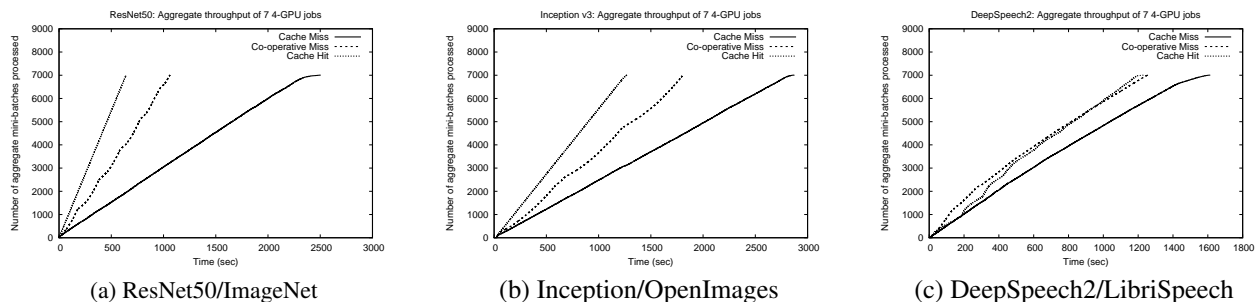
Figure 3: Multi-job progress timeline with *Quiver* for multi-jobs of 7 jobs each in three models: ResNet50, Inception v3, and DeepSpeech2. Each job runs on 4 GPUs within a single VM.

pipelining. We now analyze the time breakup of multiple pipeline stages within a mini-batch, to understand how exactly the faster I/O due to cache hits improves job performance. For this, we zoom-in on 20 mini-batches of a single ResNet50 job on 4 GPUs within a VM.
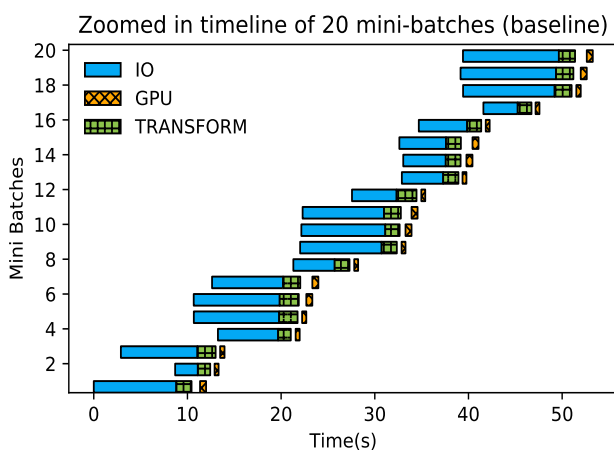

Figure 4: *Detailed timeline of 20 consecutive mini-batches of ResNet50 (different stages), under remote I/O*

Figure 4 is a Gantt chart [34] showing the micro-timeline of a ResNet50 job execution (20 consecutive mini-batches, each processing 512 images) when data is read from remote I/O. The X-axis plots time, while the Y-axis plots the mini-batch index from 1 to 20, starting from a random mini-batch during training. The three boxes in each of the bars pertain to the three main stages of mini-batch processing: *I/O* corresponds to reading the input (from remote storage or Quiver), *Transform* corresponds to performing transformation on the inputs, such as image augmentation (CPU-intensive), and *GPU* is the actual computation on GPU. Ideally, the GPU being the most expensive resource, must not be idle. However, with remote I/O, the GPU is idle most of the time (as seen from the gap between GPU phases for mini-batch i and mini-batch
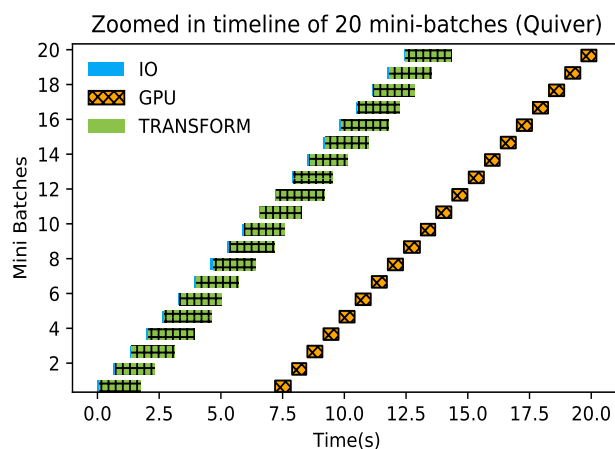

Figure 5: *Detailed timeline of 20 consecutive mini-batches of ResNet50 (different stages), under Quiver hits*

$i+1$), as I/O time constrains job progress. Figure 5 shows the micro-timeline under cache-hit in *Quiver*. As can be seen, the GPU is almost fully utilized in this setting, as the I/O finishes much faster. Although data transformation takes a long time per-mini-batch in both baseline and *Quiver*, because it is parallelized (due to pipelining) across multiple mini-batches on multiple CPU cores, it does not affect GPU utilization in *Quiver*. Thus, while both cases benefit from the I/O pipelining in PyTorch, *Quiver* is able to hide I/O latency much better.

7.5 Cache-constrained scenario

In this experiment, we run 4 ResNet50 jobs (each on 4 GPUs within a single VM) accessing the same ImageNet dataset. After about 15 minutes, we start 3 more ResNet50 jobs in 3 other VMs. We constrain the cache space to be capable of only fitting 20% of the input data. This causes *Quiver* to chop the training data into 10 chunks, and perform double buffering with two chunks at a time (§ 4.7). Figure 7 shows

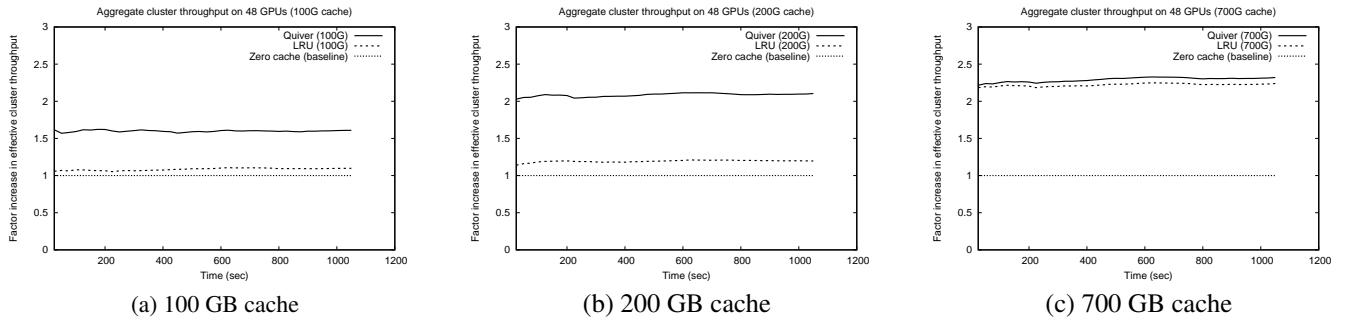(a) 100 GB cache   (b) 200 GB cache   (c) 700 GB cache

Figure 6: Cluster GPU Throughput under multiple simultaneous jobs on 48 GPUs with Quiver, basic LRU, and without caching (baseline). The workload consists of 4 jobs each of ResNet50, Inception, and DeepSpeech2. Each of the 12 jobs runs on 4 GPUs, using a total of 48 GPUs. Average cluster throughput normalized to the non-cached scenario is shown.
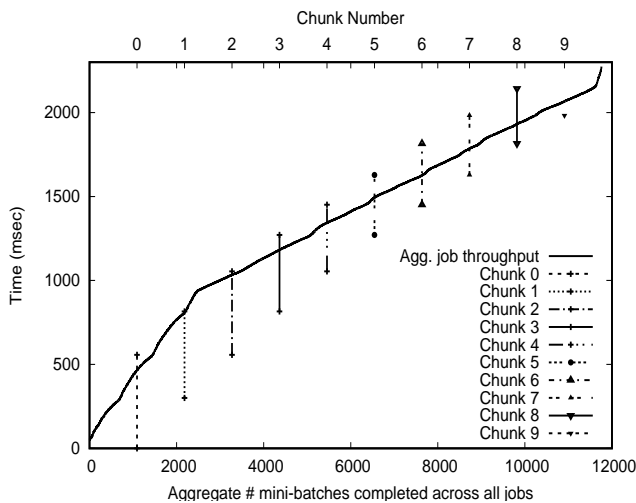

Figure 7: *Coordinated eviction with multiple jobs sharing a small slice of the cache.*

the aggregate throughput across these jobs. Every vertical line in the graph indicates the duration for which a specific chunk resides in the *Quiver* cache (the top x-axis plots the chunk number).

There are several aspects that can be seen from this graph. First, if one slices the graph by drawing a line parallel to the x-axis for any time t, it indicates the number of chunks that were cached by *Quiver* at that time, just by counting the number of vertical lines that intersect. It can be seen that at any given time, only 2 chunks are actually resident in the cache, demonstrating co-ordinated eviction. Second, in the aggregate throughput, one can see an increase in the progress rate around roughly 15 mins into the experiment (*i.e.*, when the number of jobs increased from 4 to 7), as more jobs now participate in the co-operative miss handling, improving per-job throughput. Finally, one can notice that when the three jobs start (around t=900 sec), the first two chunks of the cache have already been evicted. Despite that, the jobs are able to

make good progress, as they perform substitutable caching, but starting wth the third chunk first (while the first 4 jobs started with the first chunk). This dynamic replaceability is ensured by the cache management policy which directs new jobs to the currently active chunks in order to evict older chunks that other jobs have already exhausted.

7.6 Benefit-aware caching

In this experiment, we demonstrate the efficacy of benefit-aware caching in *Quiver*, and compare it with a simple LRU-based cache replacement policy. For this, we run a workload with a mix of three different DLT models on 48 GPUs. We run four jobs each of ResNet50, Inception, and DeepSpeech, where each job takes a single VM with 4 GPUs. As we previously saw in Figure 3, the three jobs benefit differently from caching. The jobs use three datasets: ImageNet, OpenImages, and LibriSpeech respectively.

Figure 6 shows the steady state timeline (for about 1000 seconds after cache warmup) of normalized cluster throughput. To quantify relative cluster throughput, we calculate the relative improvement in job progress rate (mini-batches processed) for all the 12 jobs compared to the baseline (no cache) configuration. We show cluster throughput under different cache sizes: no caching, 100 GB cache, 200 GB cache, and 700 GB cache. Note that the complete size of the three datasets is about 780 GB. As the 700 GB configuration is close to the complete dataset size, the performance of LRU comes close to *Quiver*. However, thrashing on the remaining 80 GB results in only a 2.2x higher throughput for LRU compared to 2.32x for *Quiver*.

More interesting is the performance of *Quiver* under more constrained caching scenarios, *i.e.*, when the cache size is much lower than the combined sizes of the datasets. In these configs (100 GB and 200 GB), *Quiver* is able to intelligently allocate cache space based on its dynamic mini-batch-aware probing (§ 5.3), besides using co-operative miss handling and substitutable hits to improve throughput. For 100G, it uses co-

operative misses for all three datasets, using a fixed chunksize of 15GB (a total of about 90GB for double buffering of three datasets). At 200 GB cache, *Quiver* automatically chooses to completely cache the ImageNet dataset (as ResNet50 benefits the most from caching), while performing co-operative misses on the other two. At 700G cache, it caches both the ImageNet and OpenImages dataset. *Quiver* is able to preferentially allocate cache space to the jobs benefiting the most, thus maximizing cluster throughput. In both these configurations, LRU performs quite poorly compared to *Quiver*, as it suffers from thrashing because of the random access pattern of the DLT jobs. Overall, even with a tiny cache (100G), *Quiver* still yields sizeable benefits of around 1.6x; the improvement in overall cluster throughput ranges between 1.6x to 2.3x depending on cache size.

8 Related Work

Improving I/O performance for DLT jobs has received some recent attention. DeepIO [39] explored pipelining of I/O fetches with computation by using an in-memory cache, and using an *entropy-aware* sampling technique. DeepIO looks at an individual DLT job in isolation; the benefits from caching for a single job are minimal unless the entire data fits in cache, because workers of a single DLT job read each data item exactly once per epoch. In contrast, *Quiver* achieves cache reuse across *multiple* jobs securely. As a result, even when only a small part of data fits in cache, it improves performance by using substitutable hits and co-operative miss handling to co-ordinate I/O access across multiple jobs. *Quiver* is also benefit-aware in its placement and thus uses the cache frugally, prioritizing jobs that benefit the most. As the authors of DeepIO note, the (modest) benefits from DeepIO in the partial caching scenario are a result of reduced copy overheads and thread scheduling cost by using RDMA shuffling; in contrast, *Quiver* actually reduces the time spent waiting on I/O by employing co-operative miss handling.

Distributed caching in the cluster context has been explored more broadly in the analytics community. For instance, Pac-Man [5] explored co-ordinated caching of data across different workers of an analytics job to extract most benefit for query performance. Similarly, intelligent scheduling of big data jobs to maximize cross-job sharing of cached data was explored in Quartet [8] The co-ordinated eviction policy in *Quiver* has some parallels to these, but the ability to handle partial data caching without thrashing is unique to *Quiver*, as it's possible only because of the substitutability property of DLT jobs. EC-cache [27] is at a distributed cluster cache that uses erasure coding for dynamic load balancing during reads. Because of the regularity of the DLT workload, the simple static partitioning in *Quiver* seems sufficient. There has been other work on caching of various forms in the big data world [15, 19, 38].

Quiver is also related to recent work on systems for deep learning, that use predictability of DLT jobs to improve efficiency. Gandiva [37] uses predictability across mini-batches to introspect on job performance, and uses it to migrate jobs across GPUs or to pack jobs tightly within the same GPU. Astra [31] exploits mini-batch predictability to perform dynamic compilation and optimization of a DLT job by online profiling of multiple choices of optimizations. *Quiver* draws on a similar insight, but uses the predictability for intelligent cache prioritization and prefetching.

Making caching and prefetching decisions informed by application-provided hints has also been studied [26]. General-purpose hints that are application-agnostic are both challenging and limiting; by building a vertically integrated, domain-specific cache exclusively for DLT jobs, the interface in *Quiver* is both simple and powerful. Co-operative caching has also been studied [18, 30]; unlike past work, *Quiver* manages a partial working set with substitutable caching and co-ordinated evictions.

The content hash-based addressing in *Quiver* is based on the notion of using a hash as a *capability*; a similar approach has been explored in content-indexed file systems [9, 23]. *Quiver* applies this idea to the context of a shared cache that simultaneously provides both data isolation and cache reuse.

9 Conclusion

Deep learning has become an important systems workload over the last few years: the number of hardware startups on accelerators for deep learning is testament to its popularity. Systems for deep learning have mostly focused on improving compute efficiency and network efficiency, and the storage layer has been largely handled by ad hoc solutions such as manual staging of data to local SSD, that have significant limitations. With *Quiver*, we provide an automated caching mechanism that helps bridge the storage performance gap in the face of ever-increasing compute capacity for deep learning. *Quiver* achieves cache efficiency by tightly integrating with the deep learning workflow and the framework, and exploits characteristics such as I/O substitutability to ensure an efficient cache even when only a subset of data can fit in cache.

Acknowledgments

References

[1] AMAZON. Amazon ec2 spot instances. run fault-tolerant workloads for up to 90% off. In *https://aws.amazon.com/ec2/spot/*.

[2] AMAZON. Amazon s3: Object storage built to store and retrieve any amount of data from anywhere. In *https://aws.amazon.com/s3/*.

[3] AMAZON. Train deep learning models on gpus using amazon ec2 spot instances. In *https://aws.amazon.com/blogs/machine-learning/train-deep-learning-models-on-gpus-using-amazon-ec2-spot-instances/*.

[4] AMODEI, D., ANUBHAI, R., BATTENBERG, E., CASE, C., CASPER, J., CATANZARO, B., CHEN, J., CHRZANOWSKI, M., COATES, A., DIAMOS, G., ELSEN, E., ENGEL, J. H., FAN, L., FOUGNER, C., HAN, T., HANNUN, A. Y., JUN, B., LEGRESLEY, P., LIN, L., NARANG, S., NG, A. Y., OZAIR, S., PRENGER, R., RAIMAN, J., SATHEESH, S., SEETAPUN, D., SENGUPTA, S., WANG, Y., WANG, Z., WANG, C., XIAO, B., YOGATAMA, D., ZHAN, J., AND ZHU, Z. Deep speech 2: End-to-end speech recognition in english and mandarin. *CoRR abs/1512.02595* (2015).

[5] ANANTHANARAYANAN, G., GHODSI, A., WARFIELD, A., BORTHAKUR, D., KANDULA, S., SHENKER, S., AND STOICA, I. Pacman: Coordinated memory caching for parallel jobs. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* (2012), pp. 267–280.

[6] BOYD, T., CAO, Y., DAS, S., JOERG, T., AND LEBAR, J. Pushing the limits of gpu performance with xla. *https://medium.com/tensorflow/pushing-the-limits-of-gpu-performance-with-xla-53559db8e473*.

[7] BREWER, E. A. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing* (2015), ACM, pp. 167–167.

[8] DESLAURIERS, F., MCCORMICK, P., AMVROSIADIS, G., GOEL, A., AND BROWN, A. D. Quartet: Harmonizing task scheduling and caching for cluster computing. In *8th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 16)* (2016).

[9] FU, K., KAASHOEK, M. F., AND MAZIERES, D. Fast and secure distributed read-only file system. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4* (2000), USENIX Association, p. 13.

[10] GOOGLE. Open images dataset. In *https://github.com/cvdfoundation/open-images-dataset* (2018).

[11] GOOGLE. Overview of the open images challenge 2018. In *https://storage.googleapis.com/openimages/web/challenge.html* (2018).

[12] GOOGLE. Youtube-8m dataset. In *https://research.google.com/youtube8m/* (2018).

[13] GRAPHCORE, AND TØRUDBAKKEN, O. Introducing the graphcore rackscale ipu pod. In *https://www.graphcore.ai/posts/introducing-the-graphcore-rackscale-ipu-pod* (2018).

[14] GROUP, D. Dawnbench: Imagenet training on resnet50. *https://dawn.cs.stanford.edu/benchmark/*.

[15] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in datacenters. In *OSDI* (2010), vol. 10, pp. 1–8.

[16] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[17] JOUPPI, N. P., YOUNG, C., PATIL, N., PATTERSON, D., AGRAWAL, G., BAJWA, R., BATES, S., BHATIA, S., BODEN, N., BORCHERS, A., ET AL. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), IEEE, pp. 1–12.

[18] KIM, H., JO, H., AND LEE, J. Xhive: Efficient cooperative caching for virtual machines. *IEEE Transactions on Computers 60*, 1 (2010), 106–119.

[19] LI, H., GHODSI, A., ZAHARIA, M., SHENKER, S., AND STOICA, I. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–15.

[20] LINLEY, M. Google announces a new generation for its tpu machine-learning hardware. *https://techcrunch.com/2018/05/08/google-announces-a-new-generation-for-its-tpu-machine-learning-hardware/*.

[21] MICROSOFT. Blob storage: Massively scalable object storage for unstructured data. In *https://azure.microsoft.com/en-in/services/storage/blobs/*.

[22] MICROSOFT. Use low-priority azure vms with batch. In *https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms*.

[23] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review* (2001), vol. 35, ACM, pp. 174–187.

[24] OPENSLR. Librispeech asr corpus. In *http://www.openslr.org/12*.

[25] PASZKE, A., GROSS, S., CHINTALA, S., AND CHANAN, G. Pytorch. In *https://pytorch.org* (2017).

[26] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 79–95.

[27] RASHMI, K., CHOWDHURY, M., KOSAIAN, J., STOICA, I., AND RAMCHANDRAN, K. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 401–417.

[28] RASLEY, J., HE, Y., YAN, F., RUWASE, O., AND FONSECA, R. Hyperdrive: Exploring hyperparameters with pop scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (2017), ACM, pp. 1–13.

[29] ROBBINS, H., AND MONRO, S. ᵃa stochastic approximation method, ᵒ annals math. *Statistics 22* (1951), 400–407.

[30] SARKAR, P., AND HARTMAN, J. Efficient cooperative caching using hints. In *OSDI* (1996), pp. 35–46.

[31] SIVATHANU, M., CHUGH, T., SINGAPURAM, S. S., AND ZHOU, L. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, ACM, pp. 909–923.

[32] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., SHLENS, J., AND WOJNA, Z. Rethinking the inception architecture for computer vision. *CoRR abs/1512.00567* (2015).

[33] VOLTA, I. The worlds most advanced data center gpu.

URL https://devblogs. nvidia. com/parallelforall/inside-volta.

[34] WIKIPEDIA. Gantt chart. *https://en.wikipedia.org/wiki/Gantt_chart*.

[35] WIKIPEDIA. Wikipedia: Multiple buffering. In *https://en.wikipedia.org/wiki/Multiple_buffering*.

[36] WU, Y., SCHUSTER, M., CHEN, Z., LE, Q. V., NOROUZI, M., MACHEREY, W., KRIKUN, M., CAO, Y., GAO, Q., MACHEREY, K., ET AL. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[37] XIAO, W., BHARDWAJ, R., RAMJEE, R., SIVATHANU, M., KWATRA, N., HAN, Z., PATEL, P., PENG, X., ZHAO, H., ZHANG, Q., ET AL. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 595–610.

[38] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 2–2.

[39] ZHU, Y., CHOWDHURY, F., FU, H., MOODY, A., MOHROR, K., SATO, K., AND YU, W. Entropy-aware i/o pipelining for large-scale deep learning on hpc systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2018), IEEE, pp. 145–156.

CRaft: An Erasure-coding-supported Version of Raft for Reducing Storage Cost and Network Cost

Zizhong Wang[†‡*]   Tongliang Li[†‡*]   Haixia Wang[‡]   Airan Shao[†‡]   Yunren Bai[†‡]

Shangming Cai[†‡]   Zihan Xu[†‡]   Dongsheng Wang[§†‡]

[†]*Department of Computer Science and Technology, Tsinghua University*

[‡]*Beijing National Research Center for Information Science and Technology, Tsinghua University*

Abstract

Consensus protocols can provide highly reliable and available distributed services. In these protocols, log entries are completely replicated to all servers. This complete-entry replication causes high storage and network costs, which harms performance.

Erasure coding is a common technique to reduce storage and network costs while keeping the same fault tolerance ability. If the complete-entry replication in consensus protocols can be replaced with an erasure coding replication, storage and network costs can be greatly reduced. RS-Paxos is the first consensus protocol to support erasure-coded data, but it has much poorer availability compared to commonly used consensus protocols, like Paxos and Raft. We point out RS-Paxos's liveness problem and try to solve it. Based on Raft, we present a new protocol, CRaft. Providing two different replication methods, CRaft can use erasure coding to save storage and network costs like RS-Paxos, while it also keeps the same liveness as Raft.

To demonstrate the benefits of our protocols, we built a key-value store based on CRaft, and evaluated it. In our experiments, CRaft could save 66% of storage, reach a 250% improvement on write throughput and reduce 60.8% of write latency compared to original Raft.

1 Introduction

Consensus protocols, such as Paxos [12] and Raft [14], can tolerate temporary failures in distributed services. They allow a collection of servers to work as a coherent group by keeping the commands in each server's log in a consistent sequence. These protocols typically guarantee *safety* and *liveness*, which means they always return correct results and can fully functional if no majority of the servers fail. Using these consensus protocols, commands can be properly replicated into each server in the same order, even if machine failures

may happen. Google's Chubby [3] is one of the earliest systems using consensus protocols. In Chubby, metadata, like locks, are replicated through different nodes by Paxos. Since Gaios [2], consensus protocols have been used to replicate all user data (typically much larger than metadata) rather than only metadata. Recently, Raft and Paxos have been applied in real large-scale systems like etcd [8], TiKV [1] and FSS [11], to replicate terabytes of user data with better availability.

In such systems, data operations will be translated into log commands and then replicated into all servers by consensus protocols. Thus, data will be transferred to all servers, and then flushed to disks. In consensus problems, to tolerate any F failures, at least $N = (2F + 1)$ servers are needed. Otherwise, a network partition may cause split groups to agree on different contents which is against the concept of consensus. Therefore, using consensus protocols to tolerate failures may cause high network and storage costs which can be around N times of the original amount of data. Since these protocols are now applied in large-scale systems and the data volume is growing larger, these costs become real challenges and they can prevent systems from achieving low latency and high throughput.

Erasure coding [16] is an effective technique to reduce storage and network costs compared to full-copy replication. It divides data into fragments, and encodes the original data fragments to generate parity fragments. The original data can be recovered from any large-enough subset of fragments, so erasure coding can tolerate faults. If each server only needs to store a fragment (can be either an original data fragment or a parity one), not the complete copy of the data, storage and network costs can be greatly reduced. Based on the above properties, erasure coding may be a good solution to the challenges of storage and network costs in consensus protocols. Erasure coding is deployed in FSS [11] for reducing storage cost. However, FSS uses a pipelined Paxos to replicate complete user data and metadata 5-ways before encoding. Therefore, extra network cost of FSS is still four times of the amount of data, which harms performance.

RS-Paxos [13] is the first consensus protocol to support erasure-coded data. Combining Paxos and erasure coding, RS-

---

Paxos reduces storage and network costs. However, RS-Paxos has poorer availability compared to Paxos. RS-Paxos trades liveness to use erasure coding for better performance. In other words, a RS-Paxos-applied system of $N = (2F + 1)$ servers cannot tolerate F failures any longer. This may be a serious problem since the system should tolerate enough failures. At the theoretical level, we tend to design a consensus protocol with the same level of liveness as Paxos and Raft. We examine this liveness problem and point out that this problem exists because the requirement of committing becomes stricter in RS-Paxos.

We present an erasure-coding-supported version of Raft, CRaft (Coded Raft). In CRaft, a leader has two methods to replicate log entries to its followers. If the leader can communicate with enough followers, it will replicate log entries by coded-fragments for better performance. Otherwise, it will replicate complete log entries for liveness. Like RS-Paxos, CRaft can handle erasure-coded data, so it can save storage and network costs. However, one major difference between CRaft and RS-Paxos is that CRaft has the same level of liveness as Paxos and Raft while RS-Paxos does not.

To verify the benefits of CRaft, we designed and built key-value stores based on different protocols, and evaluated them on Amazon EC2. In our experiments, CRaft could greatly save network traffic, leading to a 250% improvement on write throughput and a 60.8% reduction of write latency compared to original Raft. In addition, we proved that CRaft has the same availability as Raft.

In the remainder, first we briefly go through the background knowledge of Raft, erasure coding and RS-Paxos in Section 2. Next, we explain the details of our CRaft protocol in Section 3 and prove the safety property of CRaft in Section 4. Section 5 describes our implementation, experiments and evaluation. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Background

We begin by briefly describing Raft, erasure coding, RS-Paxos and then discuss RS-Paxos's liveness problem.

2.1 Raft

Raft [14] is one of the consensus protocols and it provides a good foundation for building practical systems. There are three server states in Raft, as shown in Figure 1. A leader is elected when a candidate receives votes from a majority of servers. A server can vote for a candidate only if the candidate's log is at least as up-to-date as the server's. Each server can vote at most once in each term, so Raft guarantees that there is at most one leader in one term.

The leader accepts log entries from clients and tries to replicate them to other servers, forcing the others' logs to agree with its own. When the leader finds out that one log
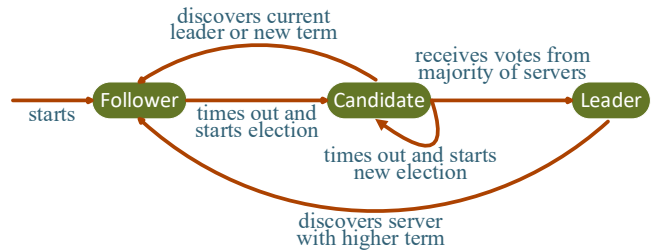

Figure 1: Three server states in Raft [14]

entry accepted in its term has been replicated to a majority of servers, this entry and its previous ones can be safely applied to its *state machine*. The leader will commit and apply these entries, and then inform followers to apply them.

Consensus protocols for practical systems typically have the following properties:

- *Safety*. They never return incorrect results under all non-Byzantine conditions.

- *Liveness*. They are fully functional as long as any majority of the servers are alive and can communicate with each other and with clients. We call this group of servers *healthy*.

The safety property in Raft is guaranteed by the Leader Completeness Property [14]: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.

Liveness is guaranteed by Raft's rules. Typically, the number of servers in systems using consensus protocols, N, is odd. Assume that $N = 2F + 1$, then Raft can tolerate any F failures. We define a consensus protocol's *liveness level* as the number of failures that it can tolerate, so Raft has an F liveness level. Higher liveness level means better liveness. No protocol can reach an $(F + 1)$ liveness level. If there exists a protocol with an $(F + 1)$ liveness level, there can be two split groups of F healthy servers and these two groups can agree on different contents respectively, which is against the safety property.

Safety and liveness are the most important properties of consensus protocols. Raft can guarantee that the safety property always holds and it also reaches the highest possible liveness level F. Furthermore, Raft has been proved to be a good foundation for system building. According to these properties, we choose Raft as the basis to design our new protocol CRaft.

2.2 Erasure Coding

Erasure coding is a common technique to tolerate faults in storage systems and network transmissions. A large number of codes have been put forward, but Reed-Solomon (RS) codes [16] are the most commonly used ones. There are two configurable positive integer parameters in RS codes, k and m.

In this technique, data are divided into k fragments with equal sizes. Then, using these k original data fragments, m parity fragments can be computed by an encoding procedure. So there will be $(k+m)$ fragments generated from the original data. The magic of a (k,m)-RS code is that any k out of total $(k+m)$ fragments are enough to recover the original data, and that is how RS codes tolerate faults.

When a consensus protocol is applied, the number of the servers, N, is usually fixed. If each server only stores one fragment produced by a (k,m)-RS code whose parameters k and m are subject to $k+m=N$, storage and network costs can be reduced to $1/k$ compared to full-copy replication. However, how to guarantee the safety property and keep liveness as good as possible cannot be ignored.

2.3 RS-Paxos

Combining erasure coding and Paxos, RS-Paxos is a reform version of Paxos which can save storage and network costs. In Paxos, commands are transferred completely. However, commands are transferred by coded-fragments in RS-Paxos. According to this change, servers can store and transfer only fragments in RS-Paxos, so storage and network costs can be reduced. The complete description of RS-Paxos can be found in the RS-Paxos paper [13].

To guarantee safety and liveness, Paxos and Raft are based on the inclusion-exclusion principle as follows.

$$|A \cup B| = |A| + |B| - |A \cap B| \tag{1}$$

The inclusion-exclusion principle guarantees that there is at least one server in two different majorities of servers,[1] and then the safety property can be guaranteed.

The insight of RS-Paxos is to increase the size of the intersection set. Specifically, after choosing a (k,m)-RS code, the read quorum Q_R, the write quorum Q_W, and the number of the servers N, should fit the following formula.

$$Q_R + Q_W - N \geq k \tag{2}$$

Then if a command is *chosen* (like *committed* in Raft), at least Q_W servers have accepted it. If a server wants to propose its own command, it will contact at least Q_R servers in *Prepare* phase. Because of (2) and (1), at least k among this Q_R servers have a fragment of the chosen command. So the proposer can recover the original command by using the k fragments and then it proposes the chosen value rather than its own.

With the benefits of erasure coding, using RS-Paxos can greatly reduce storage and network costs when $k > 1$. However, RS-Paxos decreases the fault tolerance number of failed servers. As Theorem 1 shows, RS-Paxos's liveness cannot be as good as Paxos or Raft.

Theorem 1. *Liveness level of RS-Paxos, L_{RSP}, is always less than F when $k > 1$.*

---

[1] If $|A| > |A \cup B|/2$ and $|B| > |A \cup B|/2$, $|A \cap B| = |A| + |B| - |A \cup B| > 0$.

Proof. RS-Paxos works only if at least $\max\{Q_R, Q_W\}$ servers are alive, so $L_{RSP} \leq N - \max\{Q_R, Q_W\}$.

According to (2), we have

$$\max\{Q_R, Q_W\} \geq (Q_R + Q_W)/2 \geq (N+k)/2.$$

Therefore, $L_{RSP} \leq N - (N+k)/2 = F - (k-1)/2 < F$. $\square$

RS-Paxos roughly solves the consensus problem with erasure coding, but it cannot reach an F liveness level any longer as Theorem 1 shows. RS-Paxos requires more healthy servers than Paxos or Raft to function. It is important to present a consensus protocol that not only supports erasure coding but also possesses the same liveness level as Paxos and Raft.

3 CRaft, a Reform Version of Raft that Supports Erasure Coding

Liveness is one of the most important properties of consensus protocols. However, the previous erasure-coding-supporting protocol, RS-Paxos, fails to reach an F liveness level. Thus, our goal is to design a new erasure-coding-supporting protocol (so it can save storage and network costs) that possesses an F liveness level. This new protocol is based on Raft, so it inherits the basic concepts in Raft.

To reduce network cost, leaders in the new protocol should be able to replicate their log entries to followers by using coded-fragments, like RS-Paxos. However, as Theorem 1 shows, a protocol with only coded-fragment replication method cannot reach an F liveness level. In fact, Theorem 2 shows that the complete-entry replication method in Raft is necessary for an F liveness level protocol.

Theorem 2. *When there are only $(F+1)$ healthy servers in an F liveness level protocol, an entry e can be committed only after the complete entry has been stored in all $(F+1)$ healthy servers' logs.*

Proof. If a healthy server S did not store complete entry e when e was committed, the protocol could not guarantee that it could work fully functionally in any $(F+1)$ healthy servers. Suppose only S and the previous unhealthy servers were healthy at the next moment, these $(F+1)$ currently healthy servers could not recover complete e, then the protocol had to wait for other servers. So when e was committed, all $(F+1)$ healthy servers had this complete entry. $\square$

Both coded-fragment replication and complete-entry replication are required in our new protocol. Using coded-fragment replication can save storage and network costs, while complete-entry replication can keep liveness.

Next we will discuss the details of these two replication methods, and then we try to integrate them into a complete protocol, CRaft. To explain the details of CRaft, we first define some parameters. We assume that there are $N = (2F+1)$
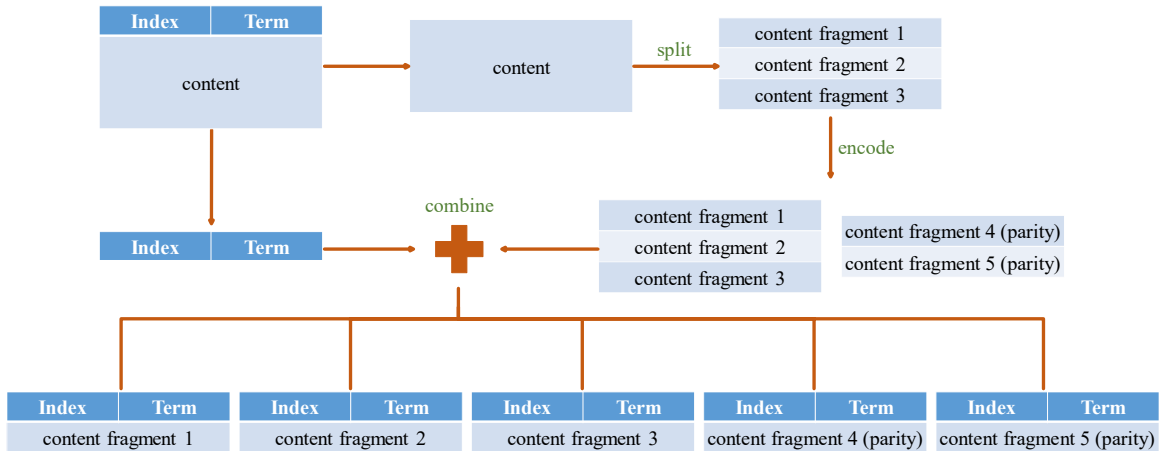
Figure 2: The encoding procedure in CRaft.

Table 1: Comparisons among Different Protocols

Performance Indicators	Different Protocols		
	CRaft	*Raft*	*RS-Paxos*
storage cost	$2F/k+1$	$2F+1$	$2F/k+1$
network cost	$2F/k$	$2F$	$2F/k$
disk I/O	$2F/k+1$	$2F+1$	$2F/k+1$
liveness level	F	F	$F-(k-1)/2$

servers in the protocol. Since CRaft should have the same availability as Raft, its liveness level should be F, which means that CRaft can still work when at least $(F+1)$ servers are healthy. We choose a (k,m)-RS code for CRaft. k and m should satisfy $k+m=N$, so each server in the protocol can correspond to one coded-fragment for each log entry. As Table 1 shows, CRaft supports erasure coding so it can save storage and network costs, while it possesses an F liveness level at the same time.

3.1 Coded-fragment Replication

When a leader in CRaft tries to replicate an entry by coded-fragment replication method, it first encodes the entry. In Raft, each log entry should contain its original content from clients and also its term and index in the protocol. When a CRaft leader tries to encode an entry, the content can be encoded into $N=(k+m)$ fragments by the (k,m)-RS code that the protocol chooses. Term and index should not be encoded, since they play important roles in the protocol. Figure 2 shows the encoding procedure.

After encoding, the leader will have N coded-fragments of the entry. Then it will send the corresponding coded-fragment to each follower. After receiving its corresponding coded-fragment, each follower will reply to the leader. When the

leader confirms that at least $(F+k)$ servers store a coded-fragment, the entry and its previous ones can be safely applied. The leader will commit and apply these entries, and then inform followers to apply them. The commitment condition of coded-fragment replication is stricter than Raft's. This commitment condition also implies that a leader cannot use coded-fragment replication to replicate an entry and then commit it when there are not $(F+k)$ healthy servers.

When a leader is down, a new leader will be elected. If an entry is already committed, the election rule of Raft guarantees that the new leader at least has a fragment of the entry, which means the safety property can be guaranteed. Since at least $(F+k)$ servers store a fragment of a committed entry, there should be at least k coded-fragments in any $(F+1)$ servers.[2] So the new leader can collect k coded-fragments and then recover the complete entry when there are at least $(F+1)$ healthy servers, which means liveness can be guaranteed.

Figure 3 shows an example of coded-fragment replication and explains why the commitment condition becomes stricter in this replication method. If a leader can commit an entry when it only confirms that $F+1=4$ servers store the entry, new leaders may be unable to recover committed entries. Like the Index 3 entry in Figure 3, it should not be committed because only five servers stored it.[3] If it was committed, consider the situation that first three servers could not connect to other servers while other four servers were all healthy. CRaft should still be able to work because its liveness level is $F=3$. However, there were at most two fragments of the Index 3 entry in the healthy servers, so new leaders were not able to recover the complete entry. The protocol had to wait for the first three servers, which means liveness cannot be guaranteed.

In coded-fragment replication, followers can receive and

---

[2]According to (1), the number of the servers storing a fragment of a committed entry is at least $(F+k)+(F+1)-N=2F+1-N+k=k$.

[3]The entry can be committed only if at least $(F+k)$ servers store it. Since $F+k=3+3>5$, the Index 3 entry should not be committed.
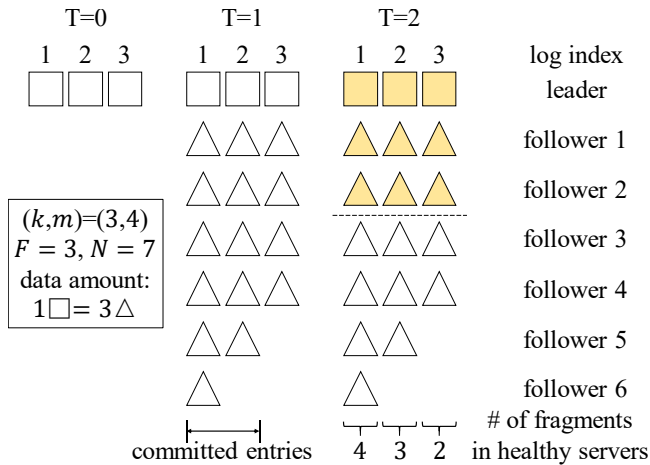
Figure 3: An example of coded-fragment replication. A square represents a complete entry, while a triangle represents a fragment of an entry. Yellow shadow means that the corresponding servers of the entries were not healthy. At $T = 0$, the leader got three entries and it tried to replicate them. At $T = 1$, followers received entry fragments with varying degrees of success. At $T = 2$, three servers, including the leader, failed.

store coded-fragments of entries. However, in Raft, followers must receive and store complete entries. According to the encoding procedure, the size of coded-fragments are about $1/k$ of the size of complete entries. So storage and network costs can be greatly reduced when using coded-fragment replication.

3.2 Complete-entry Replication

To reduce storage and network costs, leaders are encouraged to use coded-fragment replication. However, coded-fragment replication will not work when there are not $(F + k)$ healthy servers. When the number of healthy servers is greater than F and less than $(F + k)$, the leader should use complete-entry replication method to replicate entries.

In complete-entry replication, the leader has to replicate the complete entry to at least $(F + 1)$ servers before committing the entry, just like Raft. Since the committing rule is the same as Raft, safety and liveness are not problems. However, since CRaft supports coded-fragments, the leader can replicate an entry by coded-fragments rather than the complete entry to remaining followers after committing the entry.

In practical implementations, there are many strategies to replicate an entry via complete-entry replication. Define an integer parameter $0 \le p \le F$. The leader can first send complete copies of an entry to $(F + p)$ followers and then send coded-fragments to remaining $(F - p)$ followers. A smaller p means less storage and network costs, but it also means a higher probability to have longer committing latency (if no F out of $(F + p)$ answers return in time, more rounds of

communications may be required before commitment). When $p = F$, the strategy becomes the same as Raft's replication method. Figure 4 shows different strategies when $p = 0, 1, F$. In our implementation for experiments, we choose $p = 0$.

3.3 Prediction

Using coded-fragment replication rather than complete-entry replication can achieve better performance, if both methods can replicate successfully. A greedy strategy is that the leader always tries to replicate entries by coded-fragment replication. If it finds out that there are not $(F + k)$ healthy servers, it turns to replicate the entry by complete-entry replication. However, if the leader already knows that the number of healthy servers is less than $(F + k)$, the first replication attempt via coded-fragments is meaningless.

Choosing the replication method accurately can reach the best performance. However, the leader cannot be sure about the status of other servers. So it can only predict how many healthy servers it could communicate with when it tries to replicate an entry. The leader can use the most recent heartbeat answers to estimate the number of healthy servers. This prediction should be accurate enough.

When a leader tries to replicate an entry, it should use this prediction method to determine how to replicate. If the number of most recent heartbeat answers are not less than $(F + k)$, the leader should use coded-fragment replication first, and then it tries complete-entry replication if coded-fragment replication does not work. Otherwise, the leader directly uses complete-entry replication. Figure 5 concludes this process.

It is worth noting that this prediction is independent of the method that the leader chose to replicate last entry. It only relies on the most recent heartbeat answers. So it is quite possible that a leader used complete-entry replication to replicate the last entry and then it automatically chose coded-fragment replication to replicate a new entry.

3.4 Newly-elected Leader

Both replication methods can guarantee safety and liveness when leaders have all complete entries. However, when a leader is newly elected, it is likely that the newly-elected leader's log does not have complete copies but only coded-fragments of some entries. These incomplete entries are not guaranteed recoverable when there are only $(F + 1)$ healthy servers. If some of these unrecoverable entries have not been applied by the newly-elected leader, the leader has no way to deal with these entries. The leader cannot send *AppendEntries* RPCs containing any one of these entries to the followers who need them,[4] so these unrecoverable entries will retain unapplied. According to the rules of Raft, the leader's new entries received from clients cannot be replicated to the followers as

---

[4] CRaft inherits Raft's RPCs [14], the only difference between their RPCs is that entries can be encoded in CRaft's *AppendEntries* RPC.
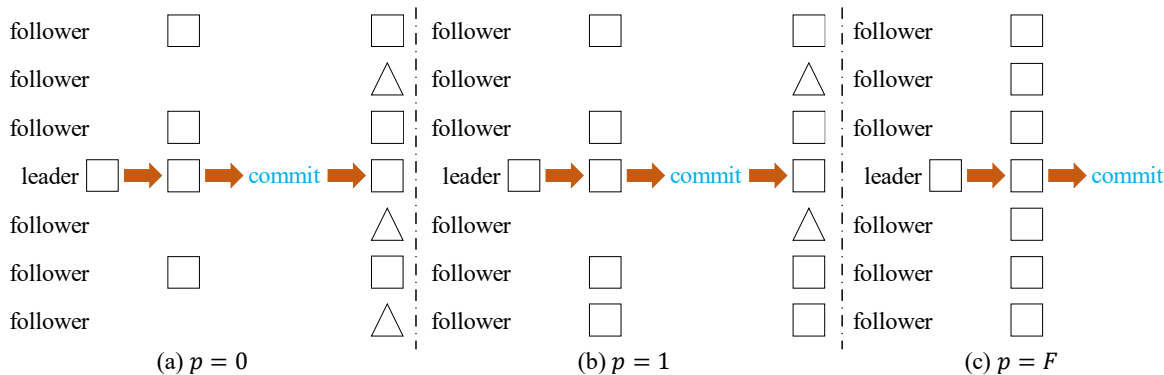
(a) $p = 0$ (b) $p = 1$ (c) $p = F$

Figure 4: Examples of complete-entry replication with parameter $p = 0, 1, F$ when $N = 7$. A square represents a complete entry, while a triangle represents a fragment of an entry.
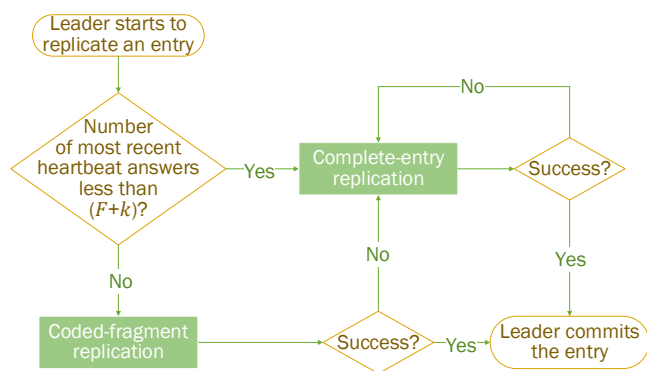

Figure 5: Flow chart of log entry replication.

well. So the protocol fails to function fully and the protocol's liveness property cannot be guaranteed. Therefore, some extra operations are required to guarantee liveness.

The coded-fragments in the newly-elected leader's log can be applied or unapplied by the leader. If a coded-fragment is applied, the entry must have been committed by a previous leader. According to the commitment condition of two replication methods, at least k coded-fragments or one complete copy of the entry are stored in any $(F + 1)$ servers. So the leader can always recover this entry when there are $(F + 1)$ healthy servers. However, if a coded-fragment is unapplied, no rules can guarantee that this entry can be recovered when there are $(F + 1)$ healthy servers.

To deal with unapplied coded-fragments, newly-elected leaders in CRaft should do the *LeaderPre* operation, before they can become fully-functioned leaders.

When a leader is newly-elected, it first checks its own log, finds out its unapplied coded-fragments. Then it asks followers for their own logs, focusing on the indexes of the unapplied coded-fragments. At least $(F + 1)$ answers (including the new leader itself) should be collected or the new leader should keep waiting. The new leader should try to recover its unapplied coded-fragments in sequence. For each entry, if

there are at least k coded-fragments or one complete copy in $(F + 1)$ answers, it can be recovered, but not allowed to be committed or applied immediately. Otherwise, the new leader should delete this entry and all the following ones (including complete entries) in its log. After recovering or deleting all the unapplied entries, the whole *LeaderPre* operation can be done. During *LeaderPre*, the newly-elected leader should keep sending heartbeats to other servers, preventing them from timing out and starting new elections.

Figure 6 shows examples of *LeaderPre*. In Figure 6, $N = 5$ and $k = 3$. S1 committed the first two entries and then crashed, and other servers had only applied the first entry. S2 was elected as a new leader, and it would do *LeaderPre*. It first asked followers about the entries in Index 2 and Index 3. In Figure 6(a), after receiving answers from itself, S3 and S4, it tried to recover the two entries. There were three fragments of the Index 2 entry and two fragments of the Index 3 entry. So S2 should recover the Index 2 entry and delete the Index 3 entry. While in Figure 6(b), S3, S5 and S2 itself all had the Index 2 entry and the Index 3 entry. So S2 could recover both of them. Though the uncommitted Index 3 entry would be handled differently if S2 collected answers from different groups of servers, the committed Index 2 entry would be guaranteed to be recovered by *LeaderPre*.

After adding *LeaderPre*, the Leader Append-Only Property in Raft has an exception: deletion in *LeaderPre*. In original Raft, the original Leader Append-Only Property is the key to prove safety, so it is necessary to prove that *LeaderPre* will not harm safety. The proof can be found in Section 4.

There are two major reasons that leaders in original Raft do not delete entries. First, leaders have no way to find out whether an unapplied entry was committed by old leaders or not. Second, even though an entry is unapplied, leaders can still replicate it to followers since it has the entry's complete copy, so there is no need to delete it. In CRaft, if there are enough fragments of an unapplied entry, the new leader can recover it and be able to replicate it. Otherwise, the new leader can conclude that this entry is uncommitted. Unrecoverable
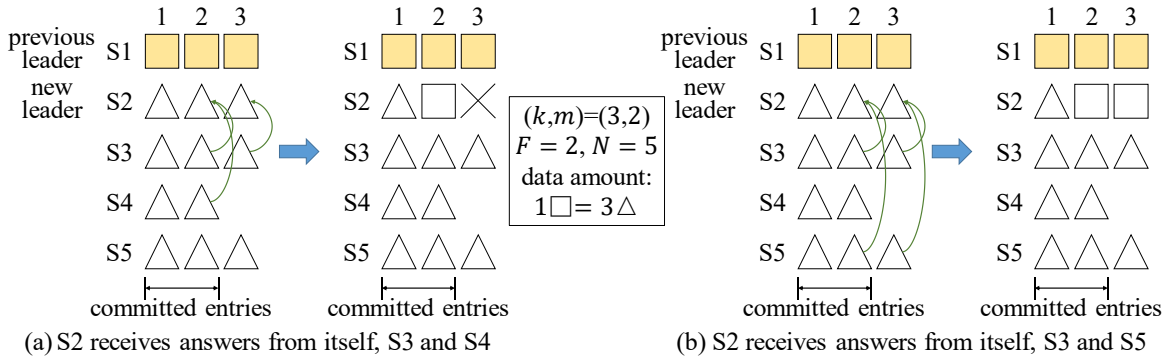
Figure 6: Examples of *LeaderPre*. A square represents a complete entry, while a triangle represents a fragment of an entry.

entries may harm CRaft's liveness, but unrecoverable also means uncommitted, so it is reasonable to delete them.

Based on Raft, CRaft provides two different replication methods for supporting erasure coding while keeping liveness. A prediction based on the most recent heartbeat answers helps the leader to choose replication method. In addition, to guarantee liveness, *LeaderPre* can help newly-elected leaders deal with unapplied coded-fragments.

3.5 Performance

The advantages of CRaft are shown in Table 1. Using a (k,m)-RS code, CRaft has advantages in reducing storage and network costs. In CRaft, ideally, only coded-fragments are needed to be transferred between the leader and followers, which indicates that the network cost can be saved to $1/k$. With this huge saving, CRaft can reach a much shorter latency and a higher throughput compared to original Raft.

The major difference between CRaft and RS-Paxos is liveness. To tolerate F failures, CRaft only needs to deploy $(2F+1)$ servers. However, RS-Paxos needs to deploy at least $(2F+3)$ servers. With the same parameter k in erasure coding, less servers required means that CRaft can save more storage and network costs compared to RS-Paxos.

One of the major concerns is the extra consumption when a leader is newly-elected. The new leader has to collect entry fragments if there are some behind followers. However, storage and network costs of CRaft in the worst situations are basically the same as Raft in any situations. Also, in most cases, the first new leader can replicate the old entries to all behind followers, so each entry only needs to be collected once extra. This harms the performance a little, but network cost is still greatly reduced generally, compared to Raft.

LeaderPre latency may affect election time, so it may affect the protocol's availability. This kind of latency is possibly affected by the number of the new leader's unapplied entries. However, a new leader can get brief information of its unapplied entries first and then collect them later. The time consumption of communicating brief information is quite

short so that *LeaderPre* latency will not harm the protocol's availability seriously.

It is optional that a newly-elected leader first collect the whole state machine by communicating with its healthy followers. This operation is helpful to reduce read latency in the future, while it may significantly increase election time so that it may harm the protocol's availability. So there is a trade-off between using it or not.

If there are far behind followers, we recommend that the followers should catch up with the leader entry by entry when they become healthy again. Snapshots can be used to compact logs in CRaft. However, the deployment of snapshots can be much more complex than original Raft, since different servers store different fragments in CRaft.

Encoding time can be a problem too. However, many studies showed that encoding time is short enough compared to transfer time in practical systems [6]. It is worth having a slightly longer encoding time to reduce network cost.

4 Safety Argument

The key of safety in Raft is the Leader Completeness Property. Since we add a new operation *LeaderPre* in CRaft, we have to prove that the property still holds. First we give the proofs of the Log Matching Property and its two related lemmas, then we use them to prove the Leader Completeness Property.

Lemma 1. *A server S has a log entry e, and e was first added into the protocol in Term T, then e and its previous entries in S's log now are the same as the entries in leader$_T$'s log when e was first added into the protocol.*

Proof. Guaranteed by contents in *AppendEntries* RPC [14]. Noticing deletions in *LeaderPre* always delete the newest part in a log, this Lemma can be proved by the same induction technique in the Raft paper [14]. □

Theorem 3. *Log Matching Property: if two logs contain an entry with the same index and term, then the logs are the same in all entries up through the given index.*

Proof. As Lemma 1 holds, these two logs are the same as the leader's log when the entry was first added into the protocol. □

Lemma 2. *A server S has a log entry **e**, then entries with the same term and smaller index are in S's log.*

Proof. A leader cannot add entries to its log until *LeaderPre* is done. When **e** was accepted, entries with the same term and smaller index must be in the leader's log. According to Lemma 1, Lemma 2 holds. □

Theorem 4. *Leader Completeness Property: if a log entry **e** is committed in a given term (Term T), then **e** will be present in the logs of the leaders for all higher-numbered terms, and **e** will not be deleted in any higher-numbered term's LeaderPre.*

Proof. We assume that the Leader Completeness Property does not hold, then we prove a contradiction. Since indexes are positive integers, there is a log entry **e** with a smallest index that breaks the property.

Consider two kinds of events: one, Leader_U $(U > T)$ does not have **e** at the time of its election; and two, **e** is deleted in *LeaderPre* by Leader_U $(U > T)$.

Assume that event one first appears. According to assumption, leaders between Term T and Term U had **e** at the time of their own elections, and **e** was never deleted in *LeaderPre*. So **e** was never deleted from anyone's log since Term T. Leader_T replicated **e** on at least $(F + 1)$ servers (no matter which replication method Leader_T used), and Leader_U received votes from at least $(F + 1)$ servers. Since $(F + 1) + (F + 1) = N + 1 > N$, at least one server both accepted **e** from Leader_T and voted for Leader_U. This server must have accepted **e** from Leader_T before voting for Leader_U, otherwise it would reject Leader_T's *AppendEntries* request. Since **e** was never deleted since Term T, this voter had **e** and voted for Leader_U at the same time. So Leader_U's log must have been as up-to-date as the voter's. If the voter and Leader_U shared the same last log term, then Leader_U's log must have been at least as long as the voter's. According to Lemma 2, Leader_U's log must have **e** and this is a contradiction. Otherwise, Leader_U's last log term must have been larger than the voter's. Since **e** was in the voter's log, Leader_U's last log term, P, was larger than T. According to assumption, in Term P, Leader_P's log had **e**. According to Lemma 1, Leader_U's log must have **e** and this is a contradiction.

So event two must appear earlier than event one. According to assumption, leaders after Term T had **e** at the time of their own elections, and **e** was never deleted in *LeaderPre* before. So **e** was never deleted from anyone's log since Term T. Since **e** was deleted in *LeaderPre*, there was an unrecoverable entry e_1. If e_1 was not **e**, since **e** was deleted, the index of e_1 must be smaller than **e**'s. Because **e** was committed by Leader_T, and e_1 had a smaller index than **e**, so e_1 had been committed. Then e_1 broke the Leader Completeness Property and had a smaller
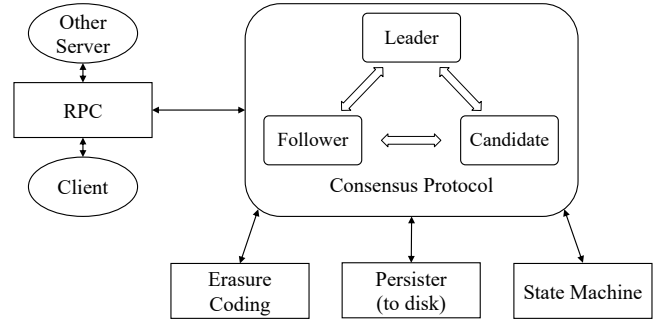

Figure 7: The structure of each server in our key-value store.

index than **e**, this is a contradiction. So **e** was deleted because it was unrecoverable. In Term T, Leader_T replicated **e** to at least $(F + 1)$ servers by complete copies, or at least $(F + k)$ servers by coded-fragments. Since **e** was never deleted from anyone's log since Term T, According to (1), there were at least one complete copy or k coded-fragments in any $(F + 1)$ answers. Then **e** was recoverable and this is a contradiction.

Then the contradiction is completely proved. The Log Matching Property guarantees that future leaders will also contain entries that are committed indirectly (not by its term's leader). So, the Leader Completeness Property holds. □

After proving the Leader Completeness Property, we can conclude the State Machine Safety Property effortlessly.

Theorem 5. *State Machine Safety Property: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.*

Proof. Suppose T is the lowest term in which any server applies an entry at the given index i. If a server applied an entry at Index i in Term U, the entry's term must be the same as the term of the Index i entry in Leader_U's log. According to the Leader Completeness Property, the term of the Index i entry in Leader_U's log should be identical to the term of the Index i entry in Leader_T's log. Since T is constant when i is given, the State Machine Safety Property holds. □

5 Experiments and Evaluation

To evaluate our protocol, we first designed a key-value store based on Raft. Then we modified it to adapt CRaft. Since RS-Paxos is based on Paxos but not Raft, it is difficult to compare RS-Paxos with CRaft or Raft directly. We took the insight of RS-Paxos and implemented an equivalent protocol named *RS-Raft* onto our key-value store. The parameters in RS-Raft have the same meanings as the ones in RS-Paxos, which are described in Section 2.3. We ran experiments on the key-value store with different protocols to present an evaluation.
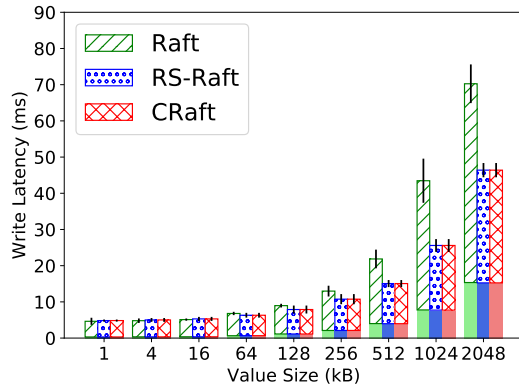
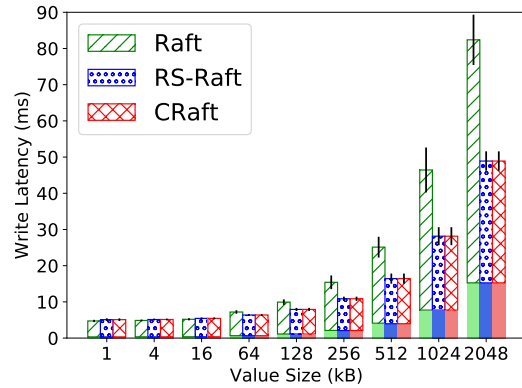Figure 8: Latency in different value sizes when $N = 5$.

Figure 9: Latency in different value sizes when $N = 7$.

5.1 Key-value Store Implementation

The key-value store we design supports three kinds of operations: *Set*, *Append* and *Get*. *Set* and *Append* operations must be logged, while *Get* operations are not. The keys were accessed uniformly in our experimental workloads. Followers can just store fragments of their entries to reduce storage cost. However, the leader should keep complete copies of entries to ensure performance of *Get*. After a new leader is elected, if there is a *Get* operation and the new leader only has a fragment of the data, it should first force at least $(k-1)$ followers' log to catch up with its own, then collect enough data fragments from them and decode the data. If the leader can directly respond to client's *Get*, we call this operation a *fast read*. Otherwise, if the leader should collect fragments from followers first, we call this operation a *recovery read*.

We used C++ to implement our key-value store. The structure of each server in our key-value store is shown in Figure 7. The consensus protocol can be Raft, CRaft and RS-Raft. We used RCF 3.0 [18] to implement RPC, and we chose TCP as transmission protocol. Jerasure 2.0 [15] is the library that we used for erasure coding.

5.2 Setup

We ran experiments on the configurations of $N = 5$ and $N = 7$, which are reasonable choices when using consensus protocols supporting erasure coding. k was set to 3, so the erasure code we used is a $(3,2)$-RS code (when $N = 5$) or a $(3,4)$-RS code (when $N = 7$).

In $N = 5$ configuration, $F = 2$, so Raft and CRaft can tolerate any two failures. We chose $Q_R = Q_W = 4$ for RS-Raft, so it can tolerate one failure. In $N = 7$ configuration, $F = 3$, so Raft and CRaft can tolerate any three failures. We chose $Q_R = Q_W = 5$ for RS-Raft, so it can tolerate two failures.

Our experiments were run on Amazon EC2 platform. We used six (when $N = 5$) or eight (when $N = 7$) instances, one of them played the role of clients and the other instances were servers. Each instance has two virtual CPU cores and 8 GiB

memory. The network bandwidth of each instance is about 550 Mbps. The storage devices we used are Amazon EBS General Purpose SSDs, each with 80000 IOPS and 1750 MB/s throughput.

5.3 Evaluation

We evaluated the protocols by measuring write latency, write throughput, network cost, liveness level and recovery read latency. Each experiment is repeated at least 100 times.

5.3.1 Latency

Figure 8 and Figure 9 show commitment latency of various value-sized write requests with error bars. Operations with a value size that larger than 2 MB can be solved by splitting it into multiple *Append* operations. Each latency consists of two parts. The part at the bottom with shadow in Figure 8 and Figure 9 is communication time from clients to the leader. This part of time is only influenced by value size. The other part is latency from the moment that the leader starts the entry to the moment that the leader commits it, and it is the part that we focus.

When value size is lower than 128 kB, three protocols perform evenly. In these situations, latency is mainly dominated by disk I/O. Since data amount is too small, even though CRaft and RS-Raft can save the amount of data flushed to disks, the I/O time usage remains almost the same, so there is not much difference between these protocols on latency.

When value size becomes larger, the advantage of CRaft and RS-Raft can be revealed. Network traffic and disk I/O both affect latency. Since CRaft and RS-Raft save network cost and disk I/O greatly, they reduce 20%–45% of latency compared to Raft when $N = 5$, and 20%–60.8% when $N = 7$.

5.3.2 Throughput

Since CRaft and RS-Raft can save the amount of data transferred and flushed to disks, they are expected to have bet-

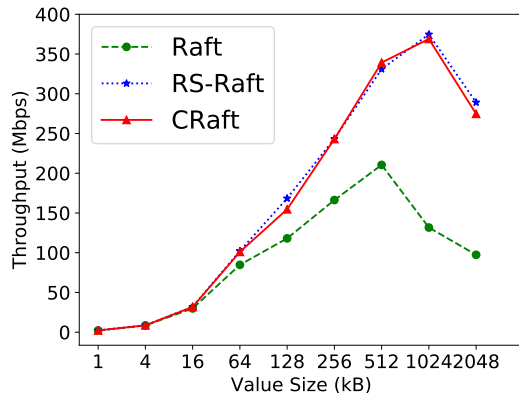Figure 10: Throughput in different value sizes when $N = 5$.

Figure 11: Throughput in different value sizes when $N = 7$.

ter throughput than Raft. We simulate the situation that 100 clients raise write request, and evaluate throughput of the leader. Figure 10 and Figure 11 show the experiment results.

The results show that CRaft and RS-Raft can improve throughput compared to Raft. They can reach about 150%–200% improvements when value size is relatively large.

With value size grows larger, throughput first increases and reaches a peak, then it will fall. Throughput will fall because of network congestion. How to prevent this network congestion problem is interesting, but it is not our concern in this paper. We compare the peak throughput of these three protocols. CRaft and RS-Raft can have a 180% improvement on write throughput when $N = 5$ and 250% when $N = 7$. Also, the throughput peaks of CRaft and RS-Raft both appear much later than Raft's. This is another advantage of CRaft and RS-Raft because of their reductions on network cost.

RS-Raft's throughput can be slightly better than CRaft's when the numbers of servers are equal, because more *AppendEntries* replies are needed before commitment in CRaft. However, it is unfair to compare these two protocols' throughput in such way, since RS-Raft's liveness is worse than CRaft's. To tolerate two failures, seven servers are required when using RS-Raft, while only five servers are required when using CRaft. So it is fairer to compare RS-Raft's throughput when $N = 7$ with CRaft's throughput when $N = 5$. According to Figure 10 and Figure 11, in this comparison, CRaft has an advantage.

5.3.3 Network Cost

We monitored the amount of data transferred from the leader to directly prove that our protocol can save network cost. In this experiment, clients raised a write request every 70 ms. Figure 12 shows the monitoring results when $N = 7$. The leader in Raft transfers about 250% of data amount compared to the leader in CRaft. This result directly proves that CRaft can greatly reduce network cost. However, ideally, when $k = 3$, the ratio between the amount of data transferred from a Raft
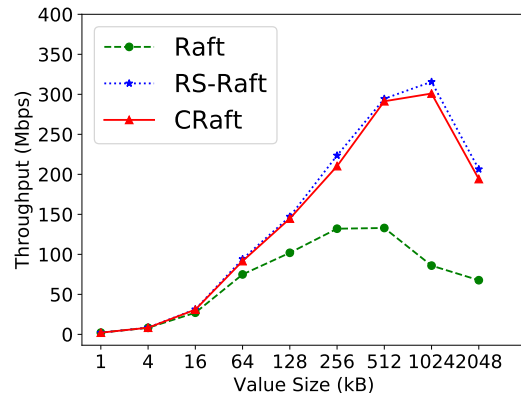
leader and a CRaft leader should be close to 300%. The gap between 250% and 300% may be caused by costs that are not generated by the consensus protocols.

5.3.4 Liveness

The major difference between CRaft and RS-Raft is liveness. CRaft can tolerate any two failures when $N = 5$, and it can tolerate any three failures when $N = 7$. Though we choose the parameters for RS-Raft to reach its highest possible liveness level, RS-Raft can only tolerate one failure when $N = 5$, and it can only tolerate two failures when $N = 7$.

Figure 13 shows the throughput of different protocols when the number of healthy servers changes in $N = 7$ experiments with error bars. RS-Raft performs very well when the number of healthy servers is no less than 5, but it cannot work when the number is 4. CRaft performs just like RS-Raft when the number of healthy servers is 6 or 7, while it performs worse than RS-Raft when the number is 5. This is because CRaft can only use complete-entry replication when the number of healthy servers is 5, so its throughput is degraded. However, CRaft can still work when the number of healthy servers is 4, just like Raft. And this proves CRaft's liveness advantage to RS-Raft.

5.3.5 Recovery Read

One of our concerns is that *recovery read* will take too much time compared to *fast read*. The leader in Raft always does *fast read*, but sometimes new leaders in CRaft may have to do *recovery read*. Noticing that a new leader only needs to do at most one *recovery read* to a specific key. If a leader needs to handle several *Get* operations of one key in its term, only the first time it has to do *recovery read*. So the existence of *recovery read* may not harm performance too much when servers do not crash too often. We made a new leader handle a *Get* operation in different protocols, and then we repeated this *Get* operation nine more times and calculated average latency. The results are shown in Figure 14. CRaft takes at
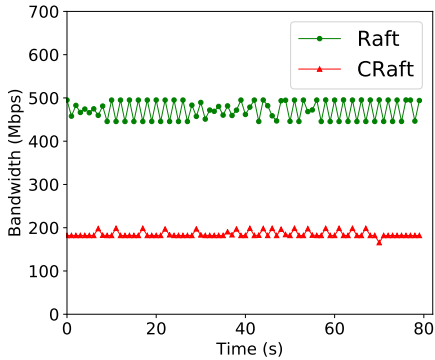
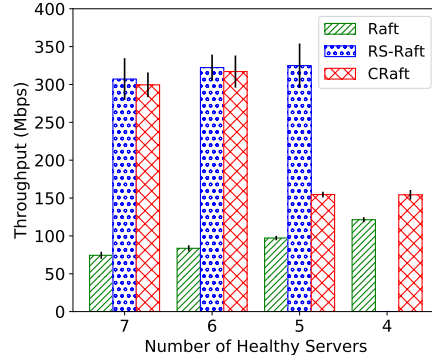Figure 12: The leader's network consumption in Raft and CRaft when $N = 7$.

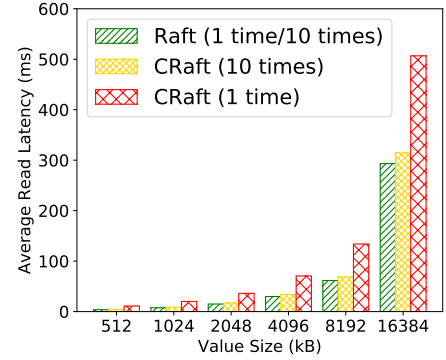Figure 13: Throughput when the number of healthy servers changes in $N = 7$ experiments.

Figure 14: Average latency of *Get* operations when $k = 3$. Only the first *Get* operation harms performance in CRaft.

most 140% more time compared to Raft handling the first *Get* operation. However, time usages of ten operations between different protocols become close enough. So we can conclude that extra time usage of *recovery read* is acceptable.

6 Related Work

Many systems use consensus protocols to provide highly reliable and available distributed services. In early years, most of them use Paxos to achieve consistency, like Chubby [3] and Spanner [5]. After the presence of Raft, many systems are using it for understandability, such as etcd [8] and TiKV [1].

Recent years, consensus protocols are not only used to replicate small size database records, but also files and data objects. Using Paxos, Gaios [2] builds a high performance data store. To prevent the service from compromising availability, FSS [11] uses a pipelined Paxos to replicate both user data and metadata. Also, etcd and TiKV use Raft to consistently distribute user data to different servers. This kind of systems are target systems of CRaft.

Erasure coding is first developed in network transmission area and now it is applied in many distributed storage systems, such as Ceph [19], HDFS [17] and Microsoft Azure [10]. The most focus problem about erasure coding now is that its recovery cost is too high compared to simple replication, and there are many works trying to solve this problem [7, 10]. Our work does not focus on this area, but we have another contribution on erasure coding. The methods that most systems replicate erasure-coded fragments are similar to using the two-phase commit protocol [9]. This kind of methods have a high probability to fail in an asynchronous network, while CRaft can still work well in this situation.

RS-Paxos [13] is the first consensus protocol supporting erasure coding. However, it cannot reach the best liveness level and it misses important details to build a practical system. Our new protocol CRaft solves the above problems. Giza [4] uses metadata versioning to provide consistency for erasure coding objects. However, its method mainly focuses on safety and ignores liveness when transferring user data. Liveness can be optimized by using CRaft.

7 Conclusions

We presented CRaft, an erasure-coded version of Raft. CRaft is based on Raft while it extents Raft to support erasure coding. With the help of erasure coding, storage and network costs can be greatly reduced.

The previous erasure-coding-supporting protocol, RS-Paxos, fails to retain an F liveness level like Paxos or Raft. CRaft solves this problem. In other words, to tolerate F faults, CRaft only needs $(2F + 1)$ servers while RS-Paxos needs more. So CRaft can save more storage and network costs.

We analyzed the performance of different protocols and we concluded that CRaft can reduce storage and network costs most while it has the best liveness. We designed a key-value store and ran experiments on it. The results show that CRaft can reduce 60.8% of latency and improve throughput by 250% compared to Raft. In the future, we will attempt to implement CRaft onto practical systems.

Acknowledgments

References

[1] TiKV Authors. TiKV: A distributed transactional key-value database. 2019. https://tikv.org/.

[2] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, pages 141–154, 2011.

[3] Michael Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[4] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. Giza: Erasure coding objects across global data centers. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, pages 539–551, 2017.

[5] 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. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8:1–8:22, 2013.

[6] Loic Dachary. Ceph Jerasure and ISA plugins benchmarks. 2015. https://blog.dachary.org/2015/05/12/ceph-jerasure-and-isa-plugins-benchmarks/.

[7] Alexandros G. Dimakis, Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010.

[8] The etcd authors. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. 2019. https://etcd.io/.

[9] Jim Gray. Notes on database operating systems: operating systems: an advanced course. *Lecture Notes in Computer Science*, 1979.

[10] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*, pages 15–26, 2012.

[11] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler. Everyone loves file: File Storage Service (FSS) in Oracle Cloud Infrastructure. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pages 15–32, 2019.

[12] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[13] Shuai Mu, Kang Chen, Yongwei Wu, and Weimin Zheng. When Paxos meets erasure code: Reduce network and storage cost in state machine replication. In *Proceedings of the 23rd International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*, pages 61–72, 2014.

[14] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference, (USENIX ATC '14)*, pages 305–319, 2014.

[15] James S. Plank and Kevin M. Greenan. Jerasure: A library in C facilitating erasure coding for storage applications version 2.0. Technical report, Department of Electrical Engineering and Computer Science, University of Tennessee, 2014. http://jerasure.org/jerasure-2.0/.

[16] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of The Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[17] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST2010)*, pages 1–10, 2010.

[18] Delta V Software. Remote call framework. 2019. http://www.deltavsoft.com/index.html/.

[19] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 307–320, 2006.

Hybrid Data Reliability for Emerging Key-Value Storage Devices

Rekha Pitchumani
Memory Solutions Lab
Samsung Semiconductor Inc.

Yang-suk Kee
Memory Solutions Lab
Samsung Semiconductor Inc.

Abstract

Rapid growth in data storage technologies created the modern data-driven world. Modern workloads and application have influenced the evolution of storage devices from simple block devices to more intelligent object devices. Emerging, next-generation Key-Value (KV) storage devices allow storage and retrieval of variable-length user data directly onto the devices and can be addressed by user-desired variable-length keys. Traditional reliability schemes for multiple block storage devices, such as Redundant Array of Independent Disks (RAID), have been around for a long time and used by most systems with multiple devices.

Now, the question arises as to what an equivalent for such emerging object devices would look like, and how it would compare against the traditional mechanism. In this paper, we present Key-Value Multi-Device (KVMD), a hybrid data reliability manager that employs a variety of reliability techniques with different trade-offs, for key-value devices. We present three reliability techniques suitable for variable length values, and evaluate the hybrid data reliability mechanism employing these techniques using KV SSDs from Samsung. Our evaluation shows that, compared to Linux mdadm-based RAID throughput degradation for block devices, data reliability for KV devices can be achieved at a comparable or lower throughput degradation. In addition, the KV API enables much quicker rebuild and recovery of failed devices, and also allows for both hybrid reliability configuration set automatically based on, say, value sizes, and custom per-object reliability configuration for user data.

1 Introduction

Modern applications require a simpler, fast and flexible storage model than what the traditional relational databases and file systems offer, and key-value stores have emerged as the popular alternative and the backbone of many scalable storage systems [1–3]. To meet the needs of such applications and to simplify the process of storing such user data even further (without added software bloat), modern storage devices have undergone a new key-value face-lift [4–8].

The Samsung Key-Value (KV) SSDs [4, 5] have incorporated the key-value store logic with the NAND flash SSD firmware, and has adopted a key-value user interface, instead of the traditional block interface to store and retrieve user data. The commercial success and widespread adoption of devices such as these will be the first step towards more intelligent and smart storage devices. A practical issue in the adoption of these devices is the identification and evaluation of suitable data reliability techniques for data stored in these devices.

Traditional systems with multiple block storage devices employ fixed-length, block-based data reliability techniques to overcome data loss due to data corruptions and device failures, and Redundant Array of Independent Disks (RAID) [9] has been the de-facto standard for these devices. KV devices, on the other hand, allows for the storage and retrieval of variable-length objects associated with variable-length user keys. Their storage semantics and as such, the data reliability techniques/recovery mechanisms are different from traditional block devices.

In this work, we address this need for a tailored data reliability solution for KV devices and present KVMD, a hybrid data reliability manager for such devices. KVMD is to KV devices as RAID is to block devices. We present four different configurable reliability techniques, all suitable for variable-length data addressed by variable-length keys, to be used in KVMD: *Hashing, Replication, Splitting* and *Packing*. These techniques serve as counterparts to the traditional RAID0, RAID1, and RAID6 architectures. We also present the different throughput, storage and reliability trade-offs of these mechanisms, enabling the users to make an informed decision.

In addition, we present three different modes of KVMD operation: a *standalone* mode, where the workload size and characteristics may remain more or less the same and is known beforehand to the user, and the user can choose a single reliability technique for all data, a *hybrid* mode, where the user can configure different reliability techniques for KVs with value sizes in different pre-configured ranges, and a *custom* mode, where the user can specify a reliability technique per KV pair and can be used in combination with either the standalone

mode or the hybrid mode.

We also evaluate the above individual techniques for different value sizes, in both the *standalone* and the *hybrid* mode (since *custom* mode is just a functional extension and the performance characteristics does not require a separate evaluation), using Samsung's NVMe Key-Value SSDs (KV SSDs). We show that, when compared to the Linux mdadm-based RAID throughput degradation for block devices, data reliability for KV devices can be achieved at a comparable or lower throughput degradation. KVMD, enabled by the flexible KV interface, also provides much quicker rebuild and recovery compared to Linux mdadm-based RAID. Finally, we conclude that, thanks to the flexible, modern device interface, KVMD for KV devices not only provides custom configuration convenience for the users, but is also either equivalent or superior to schemes for block devices in many ways.

2 Key-Value SSDs

Storage device technologies have undergone tremendous changes since the first disk drive was introduced several decades ago. Yet, the traditional random access block interface is still being used to access most modern storage devices, even the NAND flash SSDs, until recently. Here, we describe the enterprise grade NVMe Key-Value Solid State Drives from Samsung [4].
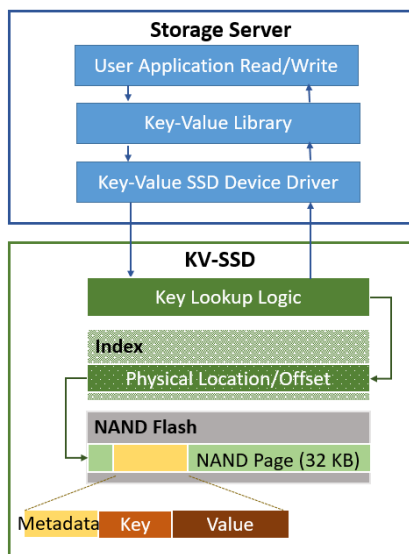

Figure 1: **Key-Value SSD IO path.** Key Lookup logic is added to the device.

NAND flash density has grown tremendously over the years; internal parallelism and read/write bandwidth of the devices have improved drastically. In addition, a NAND flash memory cell can be read and programmed only in units of pages of size 8-32 KB, and a page can be programmed only after an erase, done in larger units of size 4-8 MB. To handle such device characteristics and manage the placement

API Kind	APIs
Device API	kvs_[open/close]_device, kvs_get_device_[info/capacity/utilization], kvs_get_[min/max]_[key/value]_length, kvs_get_optimal_value_length
Container API	kvs_[create/delete/open/close]_container, kvs_list_containers, kvs_get_container_info
Key-Value API	kvs_[store/retrieve/delete]_tuple[_async], kvs_get_tuple_info, kvs_exist_tuples[_async]
Iterator API	kvs_[open/close]_iterator, kvs_iterator_next[_async]

Table 1: **Samsung Key-Value SSD API**

and retrieval of the host-addressable 4 KB logical blocks to the storage media, traditional NAND flash solid state drives already come equipped with very capable hardware and enhanced firmware.

The KV SSDs used for evaluation in this paper use the same hardware resources as those of their block SSD counterparts used for evaluation. The KV firmware is based on the block firmware and has modifications to support the storage, retrieval and cleanup of variable-length values and key. Whether the KV IO throughput matches the block IO throughput, or what the effects of increased hardware resources on KV IO throughput would be, are the topics for another paper altogether, and will not be discussed here for the sake of brevity.

Figure 1 illustrates the major components in the IO path of a KV SSD. The Samsung KV-SSDs use the Non-Volatile Memory express (NVMe) interface protocol, developed for low-latency, high-performance non-volatile memory devices connected via PCIe. As seen in the figure, the variable-length KV pair is stored along with any internal metadata in the NAND flash page in a log-like manner, and the index stores the physical location/offset of this variable-length blob, instead of storing a fixed 4 KB data in a log-like manner and indexing the 4 KB block location. The firmware now also has hash-based key lookup logic instead of the traditional logical block number based lookup. In addition, the garbage collection logic is also equipped to deal with variable-length KV pair cleanup. Kang et al. [5] describe the design and benefits of these devices in more detail.

User applications in the storage server can use the KV library API, and the KV library in turn talks to the KV SSD device driver to talk to the KV SSDs. The open-source KV-SSD host software package provides the KV API library and access to both a user-space and kernel device driver for the KV SSDs [10]. Samsung Key-Value SSD API is listed in Table 1 and the detailed description of the API can be found in the KV API spec provided with the host software package.

As can be seen in the table, the API provides management

calls to open/close a device and get information such as device capacity/utilization, min/max key and value lengths supported and optimal value length. The API also includes the concept of containers to group KV pairs. The KV API includes both asynchronous and synchronous calls to store, retrieve and delete KV pairs. Further, a user can get information about KV pair or check the existence of keys in the device. Finally, a user can open an iterator set on a predicate and can iterate over either key only or key and value in lexicographic key ordering.

3   KVMD Design

KVMD is a virtual device manager for multiple KV devices. As shown in Figure 2, KVMD handles the KV operations sent to the virtual device and stores the user data chunks in underlying KV devices it manages. KVMD's reliability manager relies on multiple pluggable reliability mechanism (RM) implementations, and can handle huge value sizes unsupported by the underlying KV devices. It can also have an optional data/metadata caching layer to improve performance.
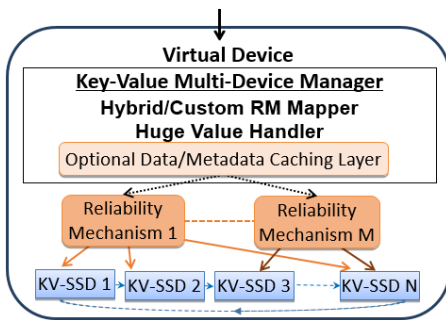

Figure 2: **KVMD Reliable Device.** Reliability device encapsulated the underlying KV devices and employs a hybrid reliability manager.

The virtual device layer works in a stateless manner, i.e., it does not have to maintain any KV to device mapping to work. KVMD can operate in three modes:

- A *standalone* mode, where the workload size and characteristics may remain more or less the same and/or is known beforehand to the user, and the user chooses a single reliability mechanism for all KV data stored in the group of devices,
- A *hybrid* mode, where the user pre-configures different reliability mechanisms for KVs in different value size ranges, and
- A *custom* mode, where the user can by default set either the *standalone* mode or the *hybrid* mode, and in addition specify a reliability technique per KV pair, upon which the specified technique will be used for the KV pair regardless of the default setting.

The size-thresholds and the corresponding reliability mechanisms of the *hybrid* mode are specified using a configuration

file. The *custom* mode is activated if the individual store call specifies a RM different than the default configuration. The configuration file is also used to specify any RM specific parameters and the erasure code implementation to use for the RM.

The KVMD manager is responsible for the creation and deletion of the underlying device abstraction layer, which handles queue-depth maintenance and calls to the underlying storage devices. The individual RMs share the underlying device abstraction objects owned by the hybrid manager. The underlying device order specified during the virtual device creation is retained by the KVMD manager. This ordering is used to determine the adjacent devices (preceding and following devices) in a circular manner. The virtual device's API is designed to be very similar to that of the KV SSD API as seen in Table 1, with an additional rebuild device call, to recover from entire device failure and rebuild the device contents, and the ability to optionally specify custom RM for stores. KVMD supports both the synchronous and asynchronous versions of the store, retrieve and delete calls, in addition to the synchronous rebuild device call.

3.1   Hybrid-Mode Operations

We will describe the operations of KVMD in the *hybrid* mode, since *custom* mode is similar and the *standalone* mode is the simpler straightforward version.

RM Determination.   Since KVMD is stateless and can operate without the optional caching layer, when the user issues a KV call, KVMD does not know if the key already exists. The underlying RMs can handle inserts and updates differently. Hence, all Store/Retrieve/Delete operations has to first determine which RM was used to write a KV pair previously, if the KV pair already exist. This information, along with other metadata is stored in the beginning of all values, as shown in Figure 3, the structure of internal values.

Figure 3: **Internal Key and Value Structures.** KVMD Metadata is stored along with user key and values.

'RM ID' identifies the RM used to store the KV pair, 'EC ID' identifies the erasure code used by the RM, 'Total splits' stores the number of splits a huge object was split into (discussed next under 'Huge Object Handling'), and 'Checksum' field stores checksum and ensures that the data read back hasn't been corrupted and is used to detect failure. Individual RMs determines how the checksum is calculated and stored. Other RM specific metadata is also stored with the value, followed by padding.

KVMD reads part/entire KV pair for every operation, to

determine the RM used to write the KV pair and then proceed with the operation, by forwarding the request to the corresponding RM. To aid in this determination by the hybrid manager, all RMs adhere to the below rules:

1. Place the first copy/chunk of the KV pair on the primary device, determined using the same hash function on the key, modulo the number of devices,
2. Store at-least the first copy/chunk/info using the same key as the user key,
3. Store metadata such as RM identifier, EC identifier, at the beginning of the value.

Huge Object Handling. Underlying KV SSD devices may have limits on the max value sizes supported, owing to their internal limitations. For example, the Samsung KV SSD used in this work has an upper size limit of 2 MB. Individual RMs may also have a maximum value size it can support for a KV-pair, based on the underlying device's maximum size minus the metadata size and its own configuration parameters, such as the number of devices a value is split and stored into. If the value size exceeds the maximum size supported by a RM, then KVMD splits the KV pair into multiple KV pairs, where each split's size is determined by maximum size supported by the RM and stores them all using the same RM no matter the residual size of the splits.

As Figure 3 shows, internal keys have additional metadata bytes in addition to the user key field, such as 'split number' and any 'RM specific metadata'. Split number is zero for both the first split of a huge value and a KV that does not have any splits. Thus, the min and max key sizes supported by KVMD are 2 bytes lesser than the underlying KV SSD supported key sizes. During a read, if the metadata in the value indicates that it is part of a huge object, KVMD issues additional IO requests to deal with the huge objects as needed.

Store. After huge objects are split into multiple objects, the RM to use to store the key is determined using the configured size-threshold. The KV pair is then read from the primary device. If the KV already exists, RM_{prev} that was previously used to store the KV pair is extracted from the metadata stored with the value, along with the number of splits stored. Then, RM_{prev}'s *update* (for matching split numbers) and *delete* (for excess splits stored previously) methods are called to let RM_{prev} handle these in a RM specific way. Finally, the new RM's *store* method is called to store all the KV pair splits.

Retrieve. The KV pair is read from the primary device. If it exists, RM_{prev} is determined, along with the user value size and the number of splits for the KV pair. The *retrieve* request could require additional calls to read from multiple splits, or just call RM_{prev}'s *complete_retrieve* method to complete the initial read as the RM sees fit. Finally, the user requested data is assembled to the user value buffer.

Delete. The KV pair is first read from the primary device. If it exists, RM_{prev} and the number of splits are determined from the metadata. Then, RM_{prev}'s *delete* method is called for all the splits.

Rebuild Device. On device failure, KVMD can rebuild all the KVs that would have been present in the failed device to a new device by iterating over all the keys present in the devices adjacent to the failed device, and performing per-KV repairs. Some RMs may require iterating over both the device in front of a failed device and that which is after, while some may require iterating over just one of device. The hybrid manager first obtains the list of drives to iterate from, from all of the underlying RMs, before starting the rebuild process.

3.2 Reliability Mechanisms

This section describes the 4 different reliability mechanisms we implemented and can be plugged into our framework. Table 2 shows the metadata information stored with the different RMs, and will be discussed further below.

3.2.1 Hashing

Hashing does not add any redundancy/data protection. Similar to RAID 0, its purpose is load balancing and request distribution to all underlying devices. It simply hashes the key and stores a single copy in the primary device, and directs all *retrieve* and *delete* calls to the primary device. When a device fails, any recovery attempt fails and user data stored in the device will be lost.

3.2.2 Single Object Replication

Replication is a simple, popular redundancy mechanism in many storage systems that is applied per object (KV pair). The primary device, determined by the key hash, stores the primary copy of the object. As shown in Figure 4, copies of the object are written to $r-1$ consecutive devices when any write happens, in addition to the primary copy, where r is the user-configurable number of replicas, and consecutive devices are determined in a circular fashion. Since 3-way replication is a popular configuration in many systems, including distributed systems, r is set to be 3, by default.
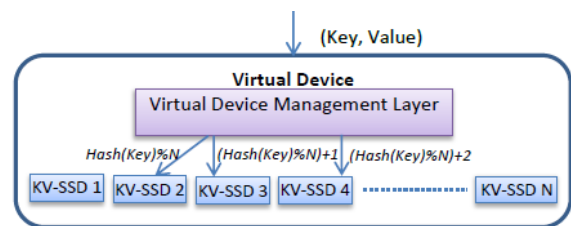

Figure 4: *Replication* stores r copies (3 here) of the data in r consecutive devices.

All copies are identical and stored under the same key in the different devices. r, the number of replicas, is also stored along with other metadata, as shown in Table 2. All RM's

also have a *num_user_key* method to return the number of devices from the primary device that would store the user key as is, without any RM specific key-metadata. Replication's *num_user_key* returns r, for example. The *update* method of all RMs obtain uk_{new}, the return value from the new RM's *num_user_key* method. If uk_{new} is less than r, Replication deletes the final $r - uk_{new}$ KVs. Finally, the *store* is passed on to the new RM, and its *store* method is called.

The *retrieve* method reads the entire value from the primary device, verifies the checksum, strips the metadata from the value and copies the user requested data onto the user buffer and returns it, if the checksum verifies. If checksum error occurs, the value is retrieved from one of the replicas, rewritten to the device that failed, and the correct data is returned to the user. The *delete* method issues delete calls on all r consecutive devices starting from the primary device.

Replication has high storage costs and write overhead, but low read and recovery costs. Since the mechanism works per-object and does not have any dependency on any other KV pair, there is no added update overhead. Replication is a good choice for very small values, where the high storage overhead is not a big strain on the system, and keeping the object intact in one piece and independent of other objects is better for performance.
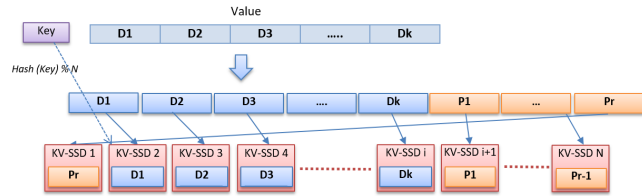

Figure 5: *Splitting* splits the value into k equal-sized objects and add r parity objects.

3.2.3 Single Object Erasure Coding - Splitting

Splitting is a single object erasure coding mechanism, that splits the user object into k equal-sized objects, adds r parity objects using a systemic MDS code and writes the $k + r$ objects to $k + r$ consecutive devices using the same user key. The code is $(4, 2)$ Reed Solomon code, by default, similar to RAID 6, though the code and parameters are configurable by the user.

As shown in Figure 5, the first data shard is placed in the primary device determined by the hash, and the other data and parity shards are placed in consecutive devices in a circular fashion. The size of the shard has to be supported by the erasure code implementation and the underlying devices, and the final shard is zero padded for parity calculation purposes, if shards cannot be evenly divided into k shards of supported size. The *ec* in Table 2 indicates the erasure code implementation to use with splitting. Due to space considerations, we will only describe one ec implementation, our best performing equivalent to RAID 6 that is used for all evaluations in the Evaluation section. The original user value size before splitting is also stored as part of the metadata, to be of use when the last shard is lost or needs recovery, in order to recover the right value content without the zero padding.

Similar to Replication, the *update* method obtains uk_{new} from the new RM and if uk_{new} is less than $k + r$, deletes the final $k + r - uk_{new}$ devices. *Delete* method issues deletes for all $k + r$ KVs. The *retrieve* method reads all splits from the k data devices asynchronously. If all their checksums verify, then metadata is stripped from the splits and they are reassembled and sent to the user. If $f <= r$ checksums fail, the required number of parity shards are read and the failed shards are recovered, rewritten to the failed devices and the user requested value is returned back to the user. If the number of failures, $f > r$, then recovery will fail and an error will be returned back to the user.

Splitting reduces the storage overhead. The read and write overhead and throughput reduction is determined by the erasure coding mechanism, code parameters and the size of the values. Similar to replication, splitting does not have any dependency to any other object; hence, no added update overhead. Splitting is recommended for big value sizes where the multiple request processing overhead does not have a huge impact on overall throughput.

3.2.4 Multi-Object Erasure Coding - Packing

Packing is a multi-object erasure coding mechanism that packs up-to k independent objects from k different devices into a single reliability set. The packing is a logical packing, purely for the sake of parity calculation. The user objects are stored in their own primary devices as determined by their hash values, independent of each other, and thus, do not intro-

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	1	···
RM								Value Metadata									Key Metadata	
Hashing	1	0	Splits		Checksum				Padding								None	
Replication	2	r	Splits		Checksum				Padding								None	
Splitting	3	ec	Splits		Checksum				Value Size			k		r	Padding		None	
Packing	4	ec	Splits		Checksum				k	r		Padding					U/M/P	
								Metadata Value										
Packing		ck	r	Key Size		Var-length Key			Value Size			Repeat ··· $(k + r - 1$ more KVs)						

Table 2: **KVMD Metadata stored per RM.**

duce any privacy concerns. It adds r parity objects to every reliability set and stores them in r devices different from the original k devices. Erasure code can be any implemented using any systemic MDS code. The default, as before, is $(4, 2)$ Reed-Solomon code, similar to RAID6 erasure coding.
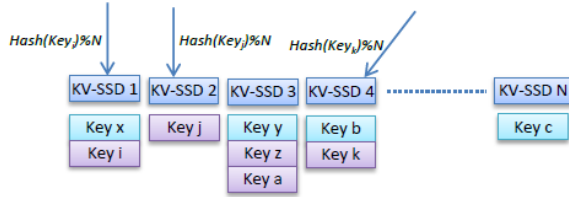

Figure 6: *Packing* packs k different objects into a single reliability set.

Figure 6 shows different keys placed in different KV SSDs based on their hashes, say key i in KV SSD 1, key j in KV-SSD 2 and key k in KV-SSD 4, etc.,. Packing queues recent write requests for each device, and chooses up-to k objects, each from a different device's queue, to be erasure coded in a set (for example, keys x, y, b, and c, the ones marked in blue in the figure, can form a reliability set). Erasure coding of the set of selected objects results in r parity objects, which are written to r different devices.

Erasure coding requires the data sizes to be the same, but there is a difference in size of the objects in a set. This challenge is overcome by virtual zero padding, i.e., the objects are padded with zero's for erasure coding, but the zero padding is not actually written to the device, as shown in Figure 7. The object value buffers and the parity objects are of the same size as the largest object in the set, rounded to a size supported by the erasure code implementation and the underlying devices.
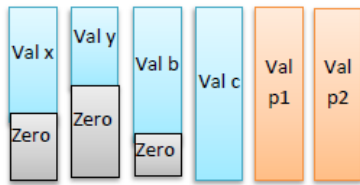

Figure 7: *Packing* pads the different values virtually with zeroes, for the sake of erasure coding.

Retrieve is straightforward; the object is read from the primary device and the checksum is verified to ensure the value isn't corrupted. Object recovery, in case of device failure or checksum failure, and recalculation of parity, in case of updates to an object in a set, requires knowledge of the erasure code set. The RM needs to know which keys were grouped together to calculate parity, and hence are in a set.

Set information together with the actual size of each object (to recreate the objects with actual size without any zero padding) is stored as metadata objects in each of the devices. The metadata objects store the number of user objects, ck and

parity objects, r in the set, along with all keys in the set, and their value sizes, as shown in Table 2. Here, ck stands for current k. We want k objects to be packed every time, but we don't want to wait too long in the queue. Hence, after a wait time threshold, available number of objects, $ck <= k$ is chosen to be packed. The RM specific identifier byte in the key is used to store indicators identifying the key as a user object or parity object or metadata object. The Metadata object for each user key is replicated and the number of replicas is set be r.

The *update* method first regroups the erasure code set the key is part of, without the key, before the write can be passed on to the new RM that will be used to store the key. First, the metadata object is read from the device, followed by the rest of the KV objects in the set and they are rewritten into a new reliability set. Once complete, the metadata object for the key is deleted, following which the store if forwarded to the new RM. Delete happens in a similar manner, except the user object is also deleted along with the deletion of the metadata object. While KVMD supports both synchronous and asynchronous calls, the underlying grouping operation in case of updates and deletes are synchronous in our current implementation, affecting performance. Hence, it is recommended to use packing for objects not expected to be updated much; reads are fast for such objects and storage overhead is also small. Similar to splitting, write throughput degradation is determined by the erasure code implementation, parameters and the size of the objects.

4 Evaluation

In this section, we evaluate both software RAID for block devices and KVMD reliability mechanisms for KV SSDs and present the results. The evaluation is done on a Linux server running CentOS 7.4. The machine has Intel(R) Xeon(R) Gold 6152 CPU @ 2.10GHz with two NUMA nodes and 22 cores per CPU, and 64 GB memory. The machine has 6 NVMe SSDs, and the same SSDs are used in both the RAID and KVMD evaluations. For the RAID evaluation with block devices, enterprise-grade block firmware is used, while a KV firmware is used for the KVMD evaluation with KV-SSDs. The KVMD virtual device is formed with all 6 devices for all tests in all evaluations. Replication is configured with total 3 replicas, while packing and splitting is configured to use 4 data devices and 2 parity devices.

KVMD is implemented as a user-space reliability library that works on top of Samsung's open source KVAPI library to access the KV-SSD devices. Unlike RAID, KVMD has hash calculation and 32-bit checksum calculation and verification overhead for every operation. After a test of couple of different implementations, we settled on the crc32 IEEE checksum calculation function using Intel's ISA-L library [11], since we found it to have the least performance degradation. For erasure coding, we use a Reed Solomon coding implementation for any k and r using the Intel ISA-L library [11]. Ad-

ditional KVMD overhead includes memory allocations/frees and memory copies during external to internal key/value conversions and vice-versa.

The goal of the experiments in this section is to evaluate the performance degradation incurred by the different RMs under different settings and to compare it against the performance degradation incurred by Linux software RAID. But, a RAID vs KVMD comparison is not an apples-to-apples comparison. The device capabilities and internal operations are different. Hence, their absolute throughput numbers are also different. *In more realistic KVS settings, KV SSDs outperform block SSDs with host KV software stack.* To learn more, we refer the readers to the work by Kang et al. [5].

The results in this section are presented with 2 y-axis. The left-hand y-axis and the bar plots show the absolute throughput numbers, and the secondary right-hand y-axis represents the percentage throughput achieved and is the axis for the lollipop plot (the red sticks with the small spheres at the end,

on top of the bars). The lollipops on top of each bar shows the percentage throughput achieved by the RM or RAID scheme represented by the bar, with respect to the first bar in the category, and provides a sense of the performance overhead. Since this work is not about the device implementation, and the numbers are from prototype firmware, and absolute performance numbers of final products are likely to be different from those presented here, **we encourage the readers to focus on the lollipop plots rather than the bar plots**, as we do in the rest of the section.

Our evaluation uses Fio's [12] asynchronous engine for RAID device and kvbench's [13] asynchronous benchmark supplied with the KV SSD host software package for KVMD device. Hence, we use fixed block and value sizes for our experiments. Since, this is a new emerging device, we also do not make any assumptions regarding the popularity of KV sizes based on previous studies, and have chosen 1 KB, 4 KB, 16 KB and 64 KB value and block sizes.
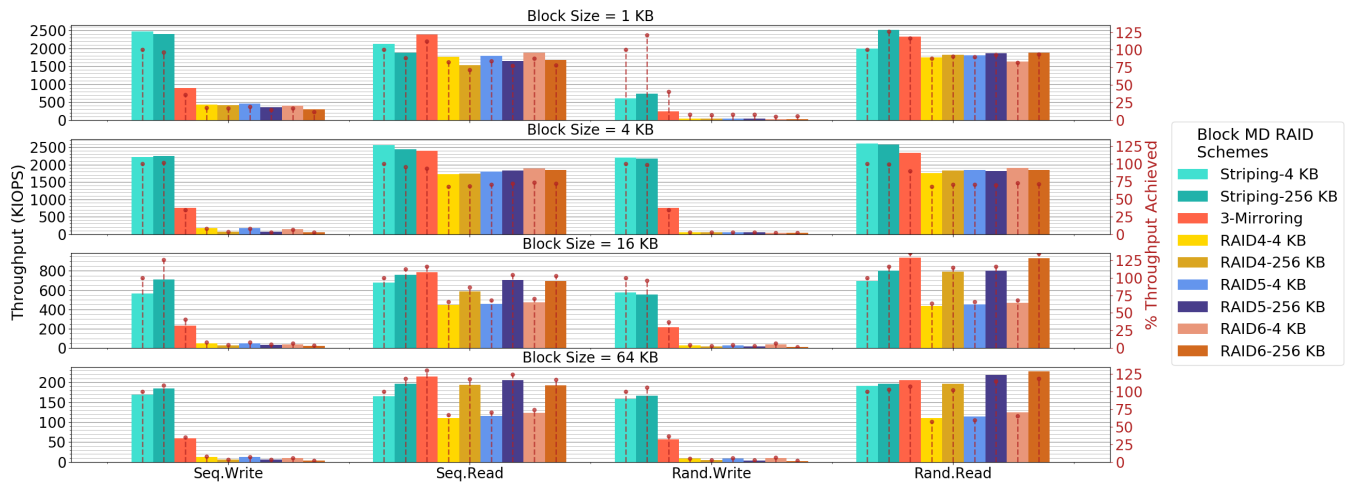

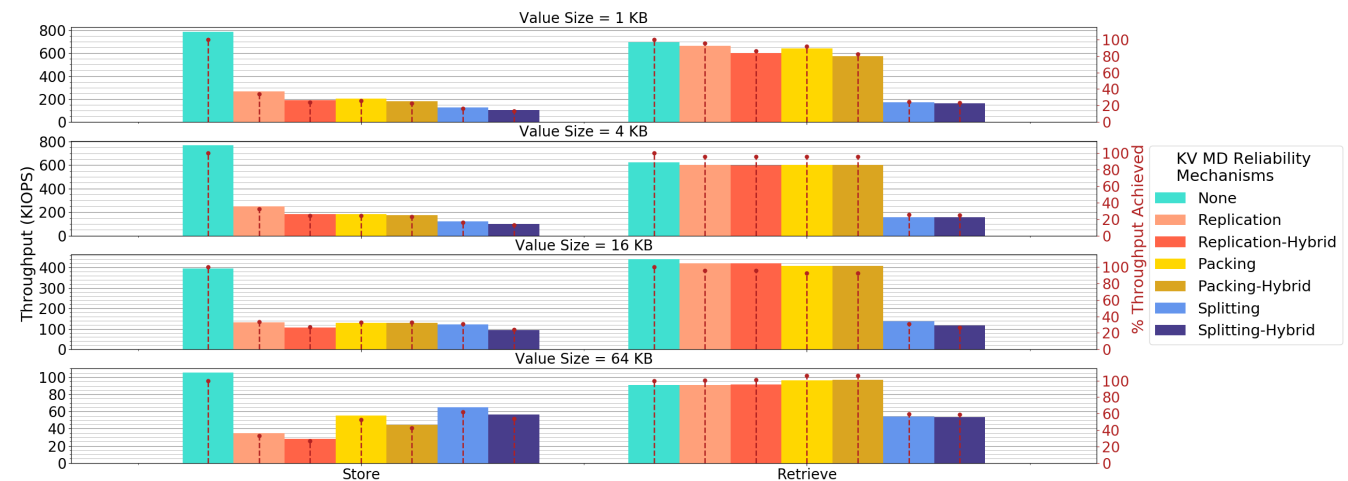Figure 8: RAID Throughput for block sizes from 1 KB to 64 KB.

Figure 9: KVMD RM throughput for various value sizes. The Hybrid mode runs of each RM shows the performance impact due to the extra reads in the mode.

4.1 Block Device RAID Performance

Mdadm software RAID is used to create RAID devices on top of NVMe block devices, in striping, mirroring, raid4, raid5 and raid6 configurations, and were tested with both 4 KB and 256 KB chunk size. Mirroring is configured as 2 virtual devices each with 3 physical devices, for 3-way mirroring similar to 3-way replication, with default settings. We measured performance with a total of 6, 12, 24 and 36 threads and found 24 threads to be lowest number of threads to perform the best. The results shown in the Figure 8 are all with 24 fio threads. Our workload sequence was sequential write, followed by sequential read, followed by random write and finally random reads.

Striping achieves the aggregated throughput of all 6 devices. As can be seen from the figure, most RAID writes incur heavy throughput degradation and perform at a much lower rate, compared to striping. The throughput degradation of mirrored writes is as expected, roughly 1/3rd of the aggregated throughput. But all other writes are unexpectedly worse, due to ready-modify-write operations.

Reads perform much better, though we can see significant performance degradation for small block sizes and for small stripe sizes, even though no additional functionality such as read verification or decoding is performed. Read performance is degraded even for larger block sizes, if the stripe size is smaller than the block size.

We observe mirroring to be the best option for performance, with a constant, understandable performance degradation for writes of all block sizes and best read performance in most cases, but has high storage costs. While erasure coding has better storage costs, very high write throughput degradation and uncalled for read throughput degradation in some configurations makes mdadm RAID erasure coding very undesirable for high performance NVMe SSDs.

4.2 KV SSD KVMD Performance

In the presented results, 'None' signifies pure KV SSD read/write throughput, obtained with 6 threads (one for each device). The same number of threads (6) is used to issue IO to the KVMD device, in all cases.

4.2.1 Fixed Value Sizes

KVMD is evaluated only in the standalone (configured only with the RM tested) and the hybrid (configured with all RMs, but only the RMs being tested are exercised) mode, since custom mode is only a functional extension. Even though only one RM is being exercised in the hybrid mode, the possibility of multiple RMs triggers an additional read request for every operation (to check if it already exists) before continuing with the operation. Hence, we observe a slight throughput degradation in the hybrid mode, compared to the standalone mode.

Store and Retrieve. Figure 9 shows the store and retrieve performance for the RMs. Replication achieves roughly 1/3rd

the aggregated 6 drive throughput, since it writes 3 objects for every object the user writes, similar to RAID 3-way mirroring. As seen in the figure, the write performance degradation is as expected, in spite of the additional hashing, checksum calculation and memory copy operations. Replication-Hybrid issues 4 requests for every write operation and hence incurs a higher, but expected performance degradation. The read throughput is very close to that of the drives without any reliability, and the slight performance degradation observed is due to checksum verification and memory copy operations for every read operation.

Packing issues the additional read request in both the standalone and the hybrid mode. Hence, the write throughput is similar in both modes. In the tested configuration, it groups every 4 user write into 14 total writes to the device - 4 user writes + 2 parity writes + 8 metadata object writes. The metadata writes are of smaller size than user writes; hence, for small value sizes where metadata write throughput is similar to object write throughput, the write throughput is close to 4/14 of the aggregated device throughput, but for larger value sizes where metadata writes are not as significant as object writes, the write throughput is close to 1/2 of the aggregated device throughput. The read throughput is similar to replication read throughput, since both read the user object in a similar fashion without any other dependency and additional IO requests. Hence, performance characteristics become similar to replication in many cases, even though the space amplification is way less.

Splitting splits the objects into 4 equal parts of size 1/4th the user object size and writes 2 more parity objects of same size as the splits. Hence, splitting issues 6 writes 1/4th the size of the user object, and its write throughput can only be 1/6th of the KV-SSD throughput for the smaller value size. As can be seen in the figure, its write throughput is the lowest among all the RMs for small value sizes, but catches up as value size increases and becomes better than others for larger value sizes. As in the case of replication, additional read request in case of hybrid mode results in slightly more throughput degradation. Reads have the same pattern as writes, but the smaller object performance is better than writes, because every user read only issues 4 read requests to the devices (to read all 4 splits), while every user write issues 6 write requests to the device (to write the 4 splits + 2 parity objects).

Updates and Deletes. For updates and deletes, we show the 4 KB value size results only in Figure 10, since other value sizes follow a similar pattern. The normalized throughput degradation of replication and splitting is similar to the read and write pattern observed earlier. This is because there are no other special update and delete handling procedure for both and they are both limited by the underlying device throughput for the workload and the number of IO requests. But packing performs poorly in both cases, as expected, because our current implementation operates synchronously and has to rewrite objects in a group to new groups before the

update/delete could proceed. We believe packing's update and delete performance can be improved further with more engineering effort, but will still be inherently limited.
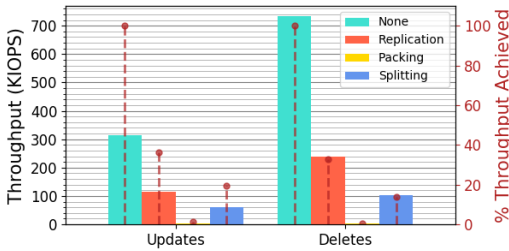

Figure 10: Update and delete throughput for 4 KB values.

4.2.2 Mixed Value Sizes

In this section, we measured the store and retrieve throughput for mixed value sizes, 4 KB, 16 KB and 64 KB in the ratio 30:40:30, and present the results in Figure 11. KVMD was configured in the hybrid mode with value size thresholds configured in such a way that 4 KB objects are handled by replication, 16 KB objects by packing and 64 KB objects by splitting. As can be seen in the figure, the read and write throughput degradation is as expected, retaining the performance characteristics of the underlying RMs exercised by the workload.
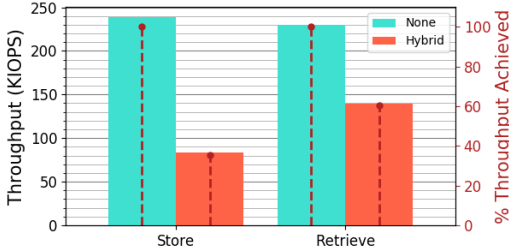

Figure 11: Mixed value size throughput measured in *hybrid* mode configured with all 3 RMs.

4.3 Rebuild Performance

In this section, we present the time taken to rebuild a failed device with very little user data. For this test, we write 1 million 4 KB user objects using the individual RMs/blocks for RAID that is roughly about 4 GB of user data, and then format/fail one of the underlying devices. We present the run time of RAID device repair and KVMD rebuild device functionality in Figure 12. As shown, KVMD reduces repair time drastically compared to RAID, since it is able to and is designed to rebuild only the user data that was written to the failed device as opposed to RAID which traditionally rebuilds the entire failed device. Reduced repair time further increases the reliability of the data stored, as shown in next section.

Time taken, in case of KVMD, is proportional to the RM read/write throughput, decode speed and the number of user
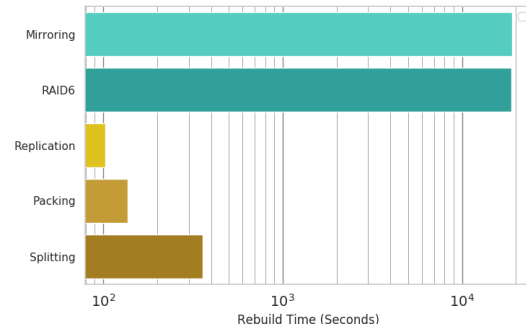

Figure 12: Single device failure rebuild times for RAID and the various KVMD RMs.

objects in the device. Replication has higher write throughput, no decode cost and fewer user objects in the devices and is the quickest. Packing has fewer user objects, but slightly lower write throughput than replication and decode cost, and hence, is slightly slower than replication. Splitting has the lowest write throughput for the workload size, decode size and number of objects, and hence, takes the most time among the RMs. While these KVMD measurements are done using a synchronous, one key at-a-time recovery implementation, it can be improved further with a multi-threaded and/or asynchronous implementations.

5 Analysis

In this section, we provide reliability analysis for KVMD and provide a comparison between the RAID levels and KVMD reliability mechanisms.

5.1 Reliability Analysis

We provide reliability analysis for KVMD, using standard Markov model, and follow the methodology commonly followed by other researchers [14, 15]. As is common in literature, for the sake of simplicity, we are going to assume that data failures are independent and are exponentially distributed, and do not consider correlated failures, even though we are aware that correlated failures are common, and their presence changes the model.

We use the metric mean time to data loss (MTTDL), to compare the reliability of the different mechanisms against each other. The MTTDL of the system is determined by the MTTDL of a reliability set, $MTTDL_{set}$, normalized by the total number of reliability sets in the system, N_{RS}.

$$MTTDL = \frac{MTTDL_{set}}{N_{RS}} \quad (1)$$

Let the average size of a user object be O, and the capacity of the underlying devices be C. Then, under *replication*, $N_{RS} = C/nO$, where n is the number of replicas. Under *splitting*, $N_{RS} = C/n(O/k)$, where n is determined by the code parameters, k and r, and is equal to $k + r$. Under *packing*,

$N_{RS} = C/n(O+M)$, where n is equal to $k+r$ as well and M is the average size of a metadata object.

MTTDL$_{set}$ is a function of mean time to failure (MTTF), mean time to repair (MTTR), the total number of objects in the set (N), and the number of parity/redundant objects in the set (G). MTTF is the average interval of time that an object will be available before failing, and MTTR is the average amount of time needed to repair an object after a failure.

Since MTTF is out of our control and is dependent on the underlying device failure rates, MTTDL$_{set}$ is affected by two factors: $a)$ the number of object failures that can be tolerated before losing user data, and $b)$ the speed at which objects can be repaired. The reliability of the system is also dependent on the number of valid sets stored in the system, unlike RAID which is dependent on the capacity of the system, and not just valid data.
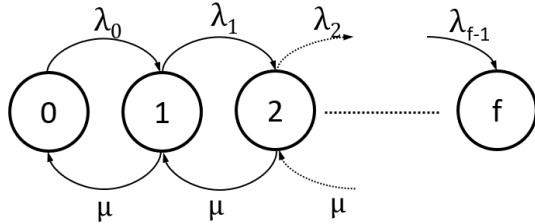

Figure 13: The Markov model used to calculate MTTDL$_{set}$.

We compute MTTDL$_{set}$ using a standard Markov model depicted in Figure 13. The numbers on the states represent the number of objects lost in the set, and f denotes the number of object losses that result in a failure and unrecoverable data loss for data in the set. The number of states for a given system depends on the configuration parameters and characteristics of the reliability mechanism. For *replication*, $f = n$, where n is the number of replicas, and for *splitting* and *packing*, $f = r+1$, where r is the number of parities in the erasure code configuration.

The forward state transitions happen on failures and backward transitions happen on recovery. Failures are assumed to be independent, at the rate $\lambda = 1/MTTF$. Since the objects in a set are distributed to N different devices, when the state is i, there are $N-i$ objects intact in a set, and the rate at which an object is lost, λ_i is equal to $(N-i)\lambda$. For recoveries, we assume a fixed recovery rate, μ for recovering a single object and moving from state i to $i-1$. While it is possible for some RMs to recover chunks in parallel and/or to move from state i to 0 directly with slightly different recovery rates, for the sake of simplicity, we model only serial recovery.

$$MTTDL_{set} \simeq \frac{\mu^{f-1}}{(N)_{(f-1)}\lambda^f} \qquad (2)$$

In equation 2, $(a)_{(b)}$ stands for $(a)(a-1)\cdots(a-b)$.

Table 3 lists the factors affecting the reliability of the system and how. While increasing the MTTF of the underlying

Control Factors	Impact on MTTDL
$\uparrow MTTF/\downarrow \lambda$	$\uparrow$
$\uparrow N$	$\downarrow$
$\uparrow f$	$\Uparrow$
$\downarrow MTTR/\uparrow \mu$	$\Uparrow$
$\downarrow \mu \times N_{RS}$	$\Uparrow$
$\downarrow$ Write Amplification (WA)	$\Uparrow$

Table 3: Factors affecting the reliability of the system.

devices will increase the reliability of the system, and vendors try their best to do the same, for a given hardware type they do not change much and out of user control. But the rest can be controlled by the user. For same number of parity objects, increasing the number of data objects decreases MTTDL, but not so much. But, adding an additional parity/replica to a set increases MTTDL by orders of magnitude.

Reducing the time taken to recover an object has a high positive impact on the reliability system. Because $\mu >> \lambda$, reducing MTTR by half has a much higher impact on MTTDL than doubling MTTF. Similarly, reducing the total time taken to repair and rebuild a device by working only on the reliability sets instead of the entire device as done by RAID, improves reliability tremendously.

Finally, for devices such as SSDs, write amplification has a negative impact on the lifetime of the device and reduces the MTTDL. Even though increased space utilization reduces MTTDL, it has been shown that data protection provided by parity improves data lifetime if the configurations are right [16]. *Replication* has a high space utilization negatively affecting the MTTDL. *Splitting* can be configured to have lower space utilization for the same MTTDL. The space utilization of *packing* can vary based on how many writes are available in the device queues and can be higher than configured. Further, updates on packed objects can increase write amplification even further, as the parity needs to be updated again.

5.2 RAID vs KVMD Comparison

Table 4 provides a comparison between the characteristics of RAID for block SSDs and KVMD for KV SSDs. For the read/write characteristics of RAID, we refer the readers to the original RAID publication [9]. KVMD calculations are given for the *standalone* mode, derived by calculating the number of IO requests issued for a given number of user requests.

Since Replication writes everything r times, its write overhead and space utilization is $1/r$, but reads are straightforward, with no additional overhead. Splitting has N writes for every user write and k to 1 reads for every read based on the whether the read is a partial read or not. Packing can end up packing 1 to k user object in a group, and in case of updates can rewrite the whole group for a single update similar to RAID6. Similar to RAID6 calculation, we do not show additional reads required. While metadata writes are

	Block SSD		KV SSD		
	RAID 1	RAID 6	Replication	Packing	Splitting
Writes	$1/r$	$[1/N, (N-2)/N]$	$1/r$	$[1/(N+m),$ $k/(N+m)]$ where m (metadata) $= [r, rk)]$	$1/N$
Reads	1	1	1	1	$[1/k, 1]$
Rebuild Time	⇑⇑ ($\propto$ Device capacity)	⇑⇑ ($\propto$ Device capacity	↓ ($\propto$ Number of user objects)	↑ ($\propto$ Number of user objects)	↑ ($\propto$ Number of user objects)
Space Utilization	$1/r$	$(N-2)/N$	$1/r$	$[1/(r+1), k/N]$ metadata is additional, but assumed small	k/N
Write Amplification	⇑	[↑ for stripe aligned and sized writes, ⇑⇑ for most writes]	⇑	↑ for inserts ⇑⇑ for updates	↑
Pros & Cons	Similar writes for all sizes. Best reads. Low MTTDL due to WA.	Very poor writes and good reads. Poor, workload-dependent MTTDL due to WA.	Similar to RAID 1. Best for small, hot objects.	Best reads. Best inserts. Very poor updates. Good, workload-dependent MTTDL.	Writes/reads $\propto$ value & request sizes. Best MTTDL. Best for large values.

Table 4: Comparison between RAID levels and KVMD RMs. Here, N is the total number of devices in a group, r is the number of replicas or parity devices, and $k = N - r$.

additional, it is the cost paid for high read performance while keeping space overhead lower than Replication. But as seen from the results, bigger the objects, lesser the metadata impact. Variable sizes complicate the space overhead calculation, but we keep it rounded and simple and ignore metadata space since it is assumed small (but is dependent on the key sizes).

The RM specific factors affecting MTTDL are also shown, for easier comparison and informed selection. Finally, the pros and cons of each and how they compare against each other is given. The comparison shows KVMD can provide for KV-SSDS all that RAID provides for block SSDs and more. While the table provides the characteristics of individual RMs under KVMD, the overall read/write performance and MTTDL in the hybrid mode in the presence of mixed value sizes will be determined by the RM configuration for value size ranges, ratio of the user requests and the average size of the objects served by the different RMs configured.

6 Related Work

Plenty of Maximum Distance Separable (MDS) block erasure codes exist to add data redundancy and failure tolerance, such as Reed-Solomon codes [17], Cauchy Reed-Solomon [18], Blaum-Roth [19], etc.,. Our work presents ways to use them all for variable-length key-value data as well. Qin et al., [20] investigated reliability issues in object-based storage devices, but considers them only as network-attached devices and study mechanisms for very large systems with thousands of nodes. While they provide reliability analysis for replication and object grouping, they do not discuss practical considerations such as variable length handling while grouping, or the impact the various schemes have on read/write performance.

Even though many modern distributed, cloud scale systems are built on top of an object-based model, they still use block storage devices underneath and either rely on the redundancy mechanism the underlying block devices employ, such as RAID [21], or provide redundancy at a higher level such as file-level redundancy rather than at a variable-length object level [15, 22, 23], where, the writes are buffered until a fixed-length block (mostly append-only large blocks) is full and replication/erasure coding is applied to these blocks and the resultant blocks are spread across different storage nodes.

In recent years, researchers have proposed a number of resilient, in-memory, distributed key-value caching solutions. Though they need to maintain key to physical location mappings, which is not required for KV devices, and do not have the same performance characteristics and workloads as that of our target system, they do share commonalities such as variable-length values and addressing scheme. Cocytus [24] uses replication for metadata and keys, and erasure coding for values by splitting the value into k parts, adding m parity parts and storing the resulting $k + m$ parts. EC-Cache [25] erasure codes the variable-length objects by splitting and storing the $k + m$ resulting parts in $k + m$ servers. KVMD also explores both replication and splitting as one of the options.

7 Limitations & Future Directions

While we cover a variety of reliability techniques and a hybrid reliability manager to use the different techniques simultaneously, for different user needs and value sizes, by no means is the work complete. In this section, we will discuss some of the limitations of the current design and implementation, and directions for future enhancements of our work.

Concurrency Control. Currently, KVMD does not implement concurrency control, and assume that the applications will implement concurrency control at their desired level. While the device guarantees consistency in case of concurrent asynchronous operations on the same key, it does not guarantee any ordering. If the application does not implement concurrency control, KVMD can be in an inconsistent state. Though Packing synchronizes all updates and deletes to protect against the concurrent update of two members in a group, it does not protect against concurrent inserts of the same key. Replication might result in different versions of the data in different devices. Splitting might have shards from two different versions in a mingled state, resulting in an inconsistent state. This can be avoided by a lock-based implementation, or through a multi-version implementation.

Crash Consistency. KVMD returns a success only after all replicas/shards (including the parity shards)/entire reliability sets (including the parity objects) are written to the device. It is once again assumed that the user/application can replay the write if it receives a failure. In cases where it cannot do so, such as during a crash, a consistency check module and KV restoration mechanism is required. While implementing a consistency check module similar to device rebuild is simple, an efficient mechanism requires design changes. Once detected, inconsistency in case of Replication can be resolved using a consensus algorithm. In case of Packing, inconsistent groups can be regrouped as long as the KV pair (actual or recovered) checksum can be verified. Partial writes in case of Splitting that has $<= r$ shards in a different version can also be recovered, others can't be. A multi-version based update mechanism can provide crash consistency with some additional impact on performance.

Optional Data/Metadata Caching Layer. The optional data/metadata caching layer shown in Figure 2 has also not yet been implemented. The benefits of a read cache is known. KVMD's value metadata is small in size and caching the metadata can avoid the initial read in case of hybrid mode and reduce the performance gap between the hybrid mode and the standalone mode. While the read of non-existent keys is quick in Samsung KV SSDs, it might not be the case with other devices, and the metadata cache could be very useful in those cases. Packing's metadata object can also be cached in the metadata caching tier, and can help improve the update/delete performance by eliminating the metadata object reads. With the metadata objects in memory and with additional in-memory only metadata per object and group, better regrouping of objects across multiple sets can be performed.

Performance. Current Packing implementation has high update and delete performance penalty, due to inefficient synchronous regrouping. A multi-version based design can enable delaying the regrouping and make room for more efficient regroup operations. Combined with the above metadata caching and in-memory metadata, current Packing inefficiencies can be greatly reduced making it a viable and competing choice. Since the current update performance of the device is roughly half the insert/delete performance, a new version insert and old version delete should have similar performance as the current update, and can solve many of the current limitations.

Picking the Right RM. Picking the right RM can be challenging for users, since the throughput is a function of the device capabilities, the RM parameters, and workload characteristics. Users usually have some intuitive knowledge about the average size of the objects in their system, and their update and delete characteristics. With some performance measurements of the underlying devices, workload information and our evaluation, the right size thresholds can be picked by an informed user. The application/user can also use the custom mode for outliers in a size threshold. Nevertheless, in reality, manual picking is hard due to the changing nature of the workload and/or limited user knowledge. Automatic size threshold determination, size threshold outlier detection and outlier custom mode utilization, to minimize space overhead while maintaining a performance level, are promising future directions for our work.

Capacity Utilization. We have also not considered the value sizes and capacity utilization of the underlying devices. Object distribution that avoids uneven capacity utilization, while maintaining the stateless design is an important future work as well.

8 Conclusion

KVMD, our hybrid reliability manager for multiple key-value storage devices, is configurable per the user needs and workload needs. KVMD can be used in the standalone mode by tiered storage systems that have fixed object size/other workload characteristics, while the hybrid mode enables object-size based configuration for a more general setting. The custom mode can be used to switch RMs for objects with certain characteristics, say hot objects, and is applied per object, giving maximum control to the user. We presented four RMs for KVMD: hashing, replication, packing and splitting, all suitable for variable-length KV objects, with different storage, throughput and reliability trade-offs. We also presented a theoretical analysis and practical evaluations of the RMs using Samsung KV SSD prototypes. Finally, we conclude that KVMD is superior to schemes for block devices in many ways.

References

[1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 2008.

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.*, 2007.

[3] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 2010.

[4] Samsung Key Value SSD enables High Performance Scaling. `http://www.samsung.com/semiconductor/global/file/insight/2017/08/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf`, August 2017.

[5] 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.

[6] Y. Jin, H. W. Tseng, Y. Papakonstantinou, and S. Swanson. KAML: A Flexible, High-Performance Key-Value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384, Feb 2017.

[7] The Seagate Kinetic Open Storage Vision. `https://www.seagate.com/tech-insights/kinetic-vision-how-seagate-new-developer-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/`, 2016.

[8] Rekha Pitchumani, James Hughes, and Ethan L. Miller. SMRDB: Key-value Data Store for Shingled Magnetic Recording Disks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, pages 18:1–18:11, 2015.

[9] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, June 1994.

[10] OpenMPDK. KV SSD host software package. `https://github.com/OpenMPDK/KVSSD`.

[11] Intel(R) Intelligent Storage Acceleration Library. `https://github.com/01org/isa-l`, 2018.

[12] Flexible I/O Tester. `https://github.com/axboe/fio`.

[13] KV Benchmark. `https://github.com/OpenMPDK/KVSSD/tree/master/application/kvbench`.

[14] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment*, March 2013.

[15] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.

[16] Sangwhan Moon and A. L. Narasimha Reddy. Does RAID Improve Lifetime of SSD Arrays? *ACM Transactions on Storage (TOS)*, June 2016.

[17] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 1960.

[18] Johannes Blömer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. Technical Report TR-95-048, International Computer Science Institute, 1995.

[19] M. Blaum and R. M. Roth. On Lowest Density MDS Codes. *IEEE Transactions on Information Theory*, January 1999.

[20] Qin Xin, Ethan L. Miller, Thomas Schwarz, Darrell D. E. Long, Scott A. Brandt, and Witold Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'03)*, MSST '03, 2003.

[21] Introduction to Lustre Architecture. `http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf`, 2017.

[22] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows

---

Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[23] 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)*, November 2006.

[24] Haibo Chen, Heng Zhang, Mingkai Dong, Zhaoguo Wang, Yubin Xia, Haibing Guan, and Binyu Zang. Efficient and Available In-Memory KV-Store with Hybrid Erasure Coding and Replication. *ACM Transactions on Storage*, 2017.

[25] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-cache: Load-balanced, Low-latency Cluster Caching with Online Erasure Coding. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, 2016.

Strong and Efficient Consistency with Consistency-Aware Durability

Aishwarya Ganesan, Ramnatthan Alagappan,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
University of Wisconsin – Madison

Abstract

We introduce *consistency-aware durability* or CAD, a new approach to durability in distributed storage that enables strong consistency while delivering high performance. We demonstrate the efficacy of this approach by designing *cross-client monotonic reads*, a novel and strong consistency property that provides monotonic reads across failures and sessions in leader-based systems. We build ORCA, a modified version of ZooKeeper that implements CAD and cross-client monotonic reads. We experimentally show that ORCA provides strong consistency while closely matching the performance of weakly consistent ZooKeeper. Compared to strongly consistent ZooKeeper, ORCA provides significantly higher throughput (1.8 – 3.3×), and notably reduces latency, sometimes by an order of magnitude in geo-distributed settings.

1 Introduction

A major focus of distributed storage research and practice has been the *consistency model* a system provides. Many models, from linearizability [20] to eventual consistency [16], with several points in-between [27, 29, 30, 48–50] have been proposed, studied, and are fairly well understood.

Despite many years of research, scant attention has been paid to a distributed system's underlying *durability model*, which has strong implications on both consistency and performance. At one extreme, *synchronous durability* requires writes to be replicated and persisted on many nodes before acknowledgment. This model is often employed to achieve strong consistency. For example, to prevent stale reads, a linearizable system (such as LogCabin [28]) synchronously makes writes durable; otherwise, an acknowledged update can be lost, exposing stale values upon subsequent reads. Synchronous durability avoids such cases, but at a high cost: poor performance. Forcing writes to be replicated and persisted, even with performance enhancements such as batching, reduces throughput and increases latency dramatically.

At the other extreme is *asynchronous durability*: each write is only lazily replicated and persisted, perhaps after buffering it in just one node's memory. Asynchronous durability is utilized in systems with weaker consistency models (such as Redis [42]); by acknowledging writes quickly, high performance is realized, but this model leads to weak semantics, exposing stale and out-of-order data to applications.

In this paper, we ask the following question: is it possible for a durability layer to enable strong consistency, yet also deliver high performance? We show this is possible if the dura-

bility layer is carefully designed, specifically by taking the consistency model the system intends to realize into account. We call this approach *consistency-aware durability* or CAD. We show how *cross-client monotonic reads*, a new and strong consistency property, can be realized with high performance by making the durability layer aware of this model. Cross-client monotonicity cannot be realized efficiently without a consistency-aware layer: synchronous durability can enable it but is slow; it simply cannot be realized upon asynchronous durability. In this paper, we implement CAD and cross-client monotonic reads in leader-based replicated systems.

Cross-client monotonic reads guarantees that a read from a client will return a state that is at least as up-to-date as the state returned to a previous read from any client, irrespective of failures and across sessions. To realize this property efficiently, CAD shifts the point of durability from writes to reads: data is replicated and persisted before it is *read*. By delaying durability of writes, CAD achieves high performance; however, by making data durable before it is read, CAD enables monotonic reads across failures. CAD does *not* incur overheads on every read; for many workloads, data can be made durable in the background before applications read it. While enabling strong consistency, CAD does not guarantee complete freedom from data loss; a few recently written items that have not been read yet may be lost if failures arise. However, given that many widely used systems adopt asynchronous durability and thus settle for weaker consistency [32, 43, 44], CAD offers a path for these systems to realize stronger consistency without compromising on performance.

Existing linearizable systems do provide cross-client monotonic reads. However, to do so, in addition to using synchronous durability, most systems restrict reads to the leader [23, 34, 38]. Such restriction limits read throughput and prevents clients from reading from nearby replicas, increasing latency. In contrast, we show how a storage system can realize this property while allowing reads at many replicas. Such a system can achieve low-latency reads from nearby replicas, making it particularly well-suited for geo-distributed settings. Further, such a system can be beneficial in edge-computing use cases, where a client may connect to different servers over the application lifetime (e.g., due to mobility [41]), but still can receive monotonic reads across these sessions.

We implement CAD and cross-client monotonic reads in a system called ORCA by modifying ZooKeeper [3]. ORCA applies many novel techniques to achieve high performance and strong guarantees. For example, a *durability-check* mechanism efficiently separates requests that read non-durable items

from those that access durable ones. Next, a *lease-based active set* technique ensures monotonic reads while allowing reads at many nodes. Finally, a *two-step lease-breaking* mechanism helps correctly manage active-set membership.

Our experiments show that ZooKeeper with CAD is significantly faster than synchronously durable ZooKeeper (optimized with batching) while approximating the performance of asynchronously durable ZooKeeper for many workloads. Even for workloads that mostly read recently written data, CAD's overheads are small (only 8%). By allowing reads at many replicas, ORCA offers significantly higher throughput (1.8 – 3.3×) compared to strongly consistent ZooKeeper (strong-ZK). In a geo-distributed setting, by allowing reads at nearby replicas, ORCA provides 14× lower latency than strong-ZK in many cases while providing strong guarantees. ORCA also closely matches the performance of weakly consistent ZooKeeper (weak-ZK). We show through rigorous tests that ORCA provides cross-client monotonic reads under hundreds of failure sequences generated by a fault-injector; in contrast, weak-ZK returns non-monotonic states in many cases. We also demonstrate how the guarantees provided by ORCA can be useful in two application scenarios.

2 Motivation

In this section, we discuss how strong consistency requires synchronous durability and how only weak consistency can be built upon asynchronous durability.

2.1 Strong Consistency atop Synchronous Durability

Realizing strong consistency requires synchronous durability. For example, consider linearizability, the strongest guarantee a replicated system can provide. A linearizable system offers two properties upon reads. First, it prevents clients from seeing non-monotonic states: the system will not serve a client an updated state at one point and subsequently serve an older state to any client. Second, a read is guaranteed to see the latest update: stale data is never exposed. However, to provide such strong guarantees upon reads, a linearizable system must synchronously replicate and persist a write [25]; otherwise, the system can lose data upon failures and so expose inconsistencies. For example, in majority-based linearizable systems (e.g., LogCabin), the leader synchronously replicates to a majority, and the nodes flush to disk (e.g., by issuing *fsync*). With such synchronous durability, linearizable systems can remain available and provide strong guarantees even when all servers crash and recover.

Unfortunately, such strong guarantees come at the cost of performance. As shown in Table 1, Redis with synchronous majority replication and persistence is 10× slower than the fully asynchronous configuration in which writes are buffered only on the leader's memory. While batching concurrent requests may improve throughput in some systems, synchronous durability fundamentally suffers from high latency.

Replication	Persistence	Throughput (ops/s)	Avg. Latency (μs)
async	async	24215	330
sync	async	9889 (2.4× ↓)	809
sync	sync	2345 (10.3× ↓)	3412

Table 1: **Synchronous Writes Costs.** *The table shows the overheads of synchronous writes in Redis with five replicas and eight clients. The arrows show the throughput drop compared to asynchronous durability. The replicas are connected via 10-Gbps links and use SSDs for persistence.*
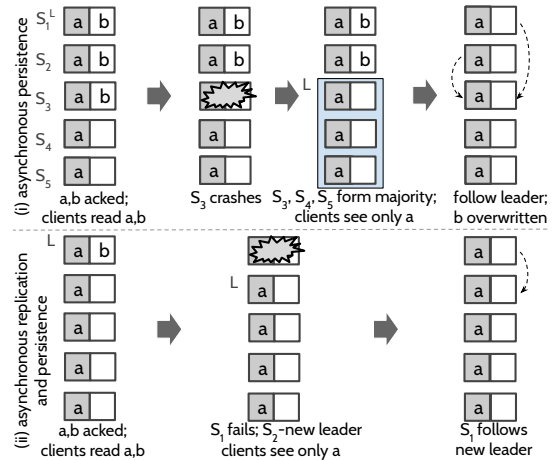

Figure 1: **Poor Consistency atop Asynchronous Durability.** *(i) shows how non-monotonic reads result upon failures with systems that persist asynchronously. (ii) shows the same for systems that replicate and persist asynchronously. Data items shown in grey denote that they are persisted.*

Synchronous durability, while necessary, is not sufficient to prevent non-monotonic and stale reads; additional mechanisms are required. For example, in addition to using synchronous durability, many practical linearizable systems restrict reads to the leader [23, 28, 34, 38]. However, such a restriction severely limits read throughput; further, it prevents clients from reading from their nearest replica, increasing read latencies (especially in geo-distributed settings).

2.2 Weak Consistency atop Asynchronous Durability

Given the cost of synchronous durability, many systems prefer asynchronous durability in which writes are replicated and persisted lazily. In fact, such asynchronous configurations are the default [32, 44] in widely used systems (e.g., Redis, MongoDB). However, by adopting asynchronous durability, as we discuss next, these systems settle for weaker consistency.

Most systems use two kinds of asynchronous-durability configurations. In the first kind, the system synchronously replicates, but persists data lazily (e.g., ZooKeeper with *forceSync* [4] disabled). In the second, the system performs both replication and persistence asynchronously (e.g., default Redis, which buffers updates only on the leader's memory).

With asynchronous persistence, the system can lose data, leading to poor consistency. Surprisingly, such cases can occur although data is replicated in memory of many nodes and when just one node crashes. Consider ZooKeeper with asynchronous persistence as shown in Figure 1(i). At first, a

majority of nodes (S_1, S_2, and S_3) have committed an item b, buffering it in memory; two nodes (S_4 and S_5) are operating slowly and so have not seen b. When a node in the majority (S_3) crashes and recovers, it loses b. S_3 then forms a majority with nodes that have not seen b yet and gets elected the leader[†]. The system has thus silently lost the committed item b and so a client that previously read a state containing items a and b may now notice an older state containing only a, exposing non-monotonic reads. The intact copies on S_1 and S_2 are also replaced by the new leader. Similar cases arise with fully asynchronous systems too as shown in Figure 1(ii).

In essence, systems built upon asynchronous durability cannot realize strong consistency properties in the presence of failures. Such systems can serve a newer state before the failure but an older one after recovery, exposing non-monotonic reads. Only models weaker than linearizability such as causal consistency can be built atop asynchronous durability; such models offer monotonic reads only in the absence of failures and within a single client session. If the server to which the client is connected crashes and recovers, the client has to establish a new session in which it may see a state older than what it saw in its previous session [30].

Weakly consistent systems can expose non-monotonic states also because they usually allow reads at many nodes [14]. For example, a client can reconnect to a different server after a disconnection, and may read an older state in the new session if a few updates have not been replicated to this server yet. For the same reason, two sessions to two different servers from a single application may receive non-monotonic states. While the above cases do not violate causal consistency by definition (because it is a different session), they lead to poor semantics for applications.

To summarize our discussion thus far, synchronous durability enables strong consistency but is prohibitively expensive. Asynchronous durability offers high performance, but only weak consistency can be built upon it. We next discuss how the seemingly conflicting goals of strong consistency and high performance can be realized together in a storage system by carefully designing its durability layer.

3   Strong, Efficient Consistency with CAD

Our goal in this paper is to design a durability primitive that enables strong consistency while delivering high performance. To this end, we first observe that asynchronous durability can lose data arbitrarily upon failures, and so prevents the realization of both non-stale and monotonic reads together. While preventing staleness requires expensive synchronous durability upon every write, we note that monotonic reads across failures can be useful in many scenarios and can be realized efficiently. We design *consistency-aware durability*

[†]While a node that has lost its data can be precluded from joining the cluster (like in Viewstamped Replication [26]), such solutions affect availability and practical systems do not employ such a strategy.

or CAD, a new durability primitive that enables this strong property with high performance.

The main idea underlying CAD is to allow writes to be completed asynchronously but enforce durability upon reads: data is replicated and persisted before it is read by clients. By delaying the durability of writes, CAD achieves high performance. However, by ensuring that the data is durable before it is read, CAD enables monotonic reads even across failures. CAD does not always incur overheads when data is read. First, for many workloads, CAD can make the data durable in the background well before applications read it. Further, only the first read to non-durable data triggers synchronous replication and persistence; subsequent reads are fast. Thus, if clients do not read data immediately after writing (which is natural for many workloads), CAD can realize the high performance of asynchronous durability but enable stronger consistency. In the case where clients do read data immediately after writing, CAD incurs overheads but ensures strong consistency.

Upon CAD, we realize cross-client monotonic reads, a strong consistency property. This property guarantees that a read from a client will always return a state that is at least as up-to-date as the state returned to a previous read from *any* client, irrespective of server and client failures, and across sessions. Linearizability provides this property but not with high performance. Weaker consistency models built atop asynchronous durability cannot provide this property. Note that cross-client monotonicity is a stronger guarantee than the traditional monotonic reads that ensures monotonicity only within a session and in the absence of failures [10, 30, 49].

Cross-client monotonic reads can be useful in many scenarios. As a simple example, consider the view count of a video hosted by a service; such a counter should only increase monotonically. However, in a system that can lose data that has been read, clients can notice counter values that may seem to go backward. As another example, in a location-sharing service, it might be possible for a user to incorrectly notice that another user went backwards on the route, while in reality, the discrepancy is caused by the underlying storage system that served the updated location, lost it, and thus later reverted to an older one. A system that offers cross-client monotonic reads avoids such cases, providing better semantics.

To ensure cross-client monotonic reads, most existing linearizable systems restrict reads to the leader, affecting scalability and increasing latency. In contrast, a system that provides this property while allowing reads at multiple replicas offers attractive performance and consistency characteristics in many use cases. First, it distributes the load across replicas and enables clients to read from nearby replicas, offering low-latency reads in geo-distributed settings. Second, similar to linearizable systems, it provides monotonic reads, irrespective of failures, and across clients and sessions which can be useful for applications at the edge [36]. Clients at the edge may often get disconnected and connect to different servers, but still can get monotonic reads across these sessions.

4 ORCA Design

We now describe ORCA, a leader-based majority system that implements consistency-aware durability and cross-client monotonic reads. We first provide a brief overview of leader-based systems (§4.1) and outline ORCA's guarantees (§4.2). We then describe the mechanisms underlying CAD (§4.3). Next, we explain how we realize cross-client monotonic reads while allowing reads at many nodes (§4.4). Finally, we explain how ORCA correctly ensures cross-client monotonic reads (§4.5) and describe our implementation (§4.6).

4.1 Leader-based Majority Systems

In leader-based systems (such as ZooKeeper), all updates flow through the leader which establishes a single order of updates by storing them in a log and then replicating them to the followers [21, 39]. The leader is associated with an epoch: a slice of time, in which at most one leader can exist [6, 39]. Each update is uniquely identified by the epoch in which it was appended and its position in the log. The leader constantly sends heartbeats to the followers; if the followers do not hear from the leader for a while, they elect a new leader. With synchronous durability, the leader acknowledges an update only after a majority of replicas (i.e., $\lfloor n/2 \rfloor + 1$ nodes in an n-node system) have persisted the update. With asynchronous durability, updates are either buffered in memory on just the leader (asynchronous replication and persistence) or a majority of nodes (asynchronous persistence) before acknowledgment.

When using synchronous durability and restricting reads to the leader, the system provides linearizability: a read is guaranteed to see the latest update and receive monotonic states. With asynchronous durability and when allowing reads at all nodes, these systems only provide sequential consistency [8], i.e., a global order of operations exists but if servers crash and recover, or if clients read from different servers, reads may be stale and non-monotonic [8, 38].

4.2 Failure Model and Guarantees

Similar to many majority-based systems, ORCA intends to tolerate only fail-recover failures, not Byzantine failures [24]. In the fail-recover model, nodes may fail at any time and recover at a later point. Nodes fail in two ways; first, they could crash (e.g., due to power failures); second, they may get partitioned due to network failures. When a node recovers from a crash, it loses its volatile state and is left only with its on-disk state. During partitions, a node's volatile state remains intact, but it may not have seen data that the other nodes have. **Guarantees.** ORCA preserves the properties of a leader-based system that uses asynchronous durability, i.e., it provides sequential consistency. However, in addition, it also provides cross-client monotonic reads under all failure scenarios (e.g., even if all replicas crash and recover), and across sessions. ORCA is different from linearizable systems in that it does not guarantee that reads will never see stale data. For example, if failures arise after writing the data but before reading it, ORCA
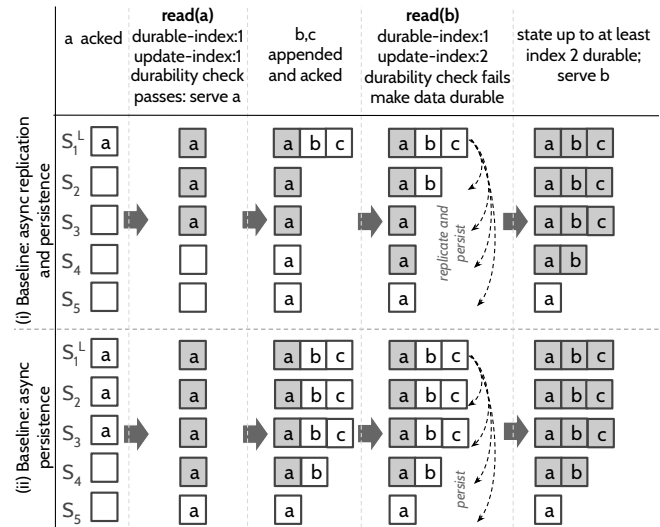

Figure 2: **CAD Durability Check.** *The figure shows how* CAD *works. Data items shown in grey are durable. In (i), the baseline is fully asynchronous; in (ii), the baseline synchronously replicates but asynchronously persists. At first, when item a is durable, read(a) passes the durability check. Items b and c are appended then. The check for read(b) fails; hence, the leader makes the state durable after which it serves b.*

may lose a few recent updates and thus subsequent reads can get an older state. Majority-based systems remain available as long as a majority of nodes are functional [7, 39]; ORCA ensures the same level of availability.

4.3 CAD Durability Layer

In the rest of this section, we use asynchronous durability as the baseline to highlight how CAD is different from it. CAD aims to perform similarly to this baseline but enable stronger consistency. We now provide intuition about how CAD works and explain its mechanisms; we use Figure 2 to do so.

4.3.1 Updates

CAD preserves the update path of the baseline asynchronous system as it aims to provide the same performance during writes. Thus, if the baseline employs asynchronous replication and persistence, then CAD also performs both replication and persistence asynchronously, buffering the data in the memory of the leader as shown in Figure 2(i). Similarly, if the baseline synchronously replicates but asynchronously persists, then CAD also does the same upon writes as shown in Figure 2(ii). While preserving the update path, in CAD, the leader keeps replicating updates in the background and the nodes flush to disk periodically. We next discuss how CAD handles reads.

4.3.2 State Durability Guarantee

When a read for an item i is served, CAD guarantees that the *entire state* (i.e., writes even to other items) up to the last update that modifies i are durable. For example, consider a log such as $[a, b_1, c, b_2, d]$; each entry denotes a (non-durable) update to an item, and the subscript shows how many updates are done to a particular item. When item b is read, CAD

guarantees that all updates at least up to b_2 are made durable before serving b. CAD makes the entire state durable instead of just the item because it aims to preserve the update order established by the leader (as done by the base system).

CAD considers the state to be durable when it can recover the data after any failures including cases where all replicas crash and recover and in all successive views of the cluster. Majority-based systems require at least a majority of nodes to form a new view (i.e., elect a leader) and provide service to clients. Thus, if CAD safely persists data on at least a majority of nodes, then at least one node in any majority even after failures will have all the data that has been made durable (i.e., that was read by the clients) and thus will survive into the new view. Therefore, CAD considers data to be durable when it is persisted on the disks of at least a majority of nodes.

4.3.3 Handling Reads: Durability Check

When a read request for an item i arrives at a node, the node can immediately serve i from its memory if all updates to i are already durable (e.g., Figure 2, read of item a); otherwise, the node must take additional steps to make the data durable. As a result, the node first needs to be able to determine if all updates to i have been made durable or not.

A naive way to perform this check would be to maintain for each item how many nodes have persisted the item; if at least a majority of nodes have persisted an item, then the system can serve it. A shortcoming of this approach is that the followers must inform the leader the set of items they have persisted in each response, and the leader must update the counts for all items in the set on every acknowledgment.

CAD simplifies this procedure by exploiting the ordering of updates established by the leader. Such ordering is an attribute common to many majority-based systems; for example, the ZooKeeper leader stamps each update with a monotonically increasing epoch-counter pair before appending it to the log [5]. In CAD, with every response, the followers send the leader only a single index called the *persisted-index* which is the epoch-counter of the last update they have written to disk. The leader also maintains only a single index called the *durable-index* which is the index up to which at least a majority of nodes have persisted; the leader calculates the durable-index by finding the highest persisted-index among at least a majority (including self).

When a read for an item i arrives at the leader, it compares the *update-index* of i (the epoch-counter of the latest update that modifies i) against the system's durable-index. If the durable-index is greater‡ than the update-index, then all updates to i are already durable and so the leader serves i immediately; otherwise, the leader takes additional steps (described next) to make the data durable. If the read arrives at a follower, it performs the same check (using the durable-index sent by the leader in the heartbeats). If the check passes, it

---
‡An index a is greater than index b if (a.epoch > b.epoch) or (a.epoch == b.epoch and a.counter > b.counter).

serves the read; otherwise, it redirects the request to the leader which then makes the data durable.

4.3.4 Making the Data Durable

If the durability check fails, CAD needs to make the state (up to the latest update to the item being read) synchronously durable before serving the read. The leader treats the read for which the check fails specially. First, the leader synchronously replicates all updates upto the update-index of the item being read if these updates have not yet been replicated. The leader also informs the followers that they must flush their logs to disk before responding to this request.

When the followers receive such a request, they synchronously append the updates and flush the log to disk and respond. During such a flush, all previous writes buffered are also written to disk, ensuring that the entire state up to the latest update to the item being read is durable. Fortunately, the periodic background flushes reduce the amount of data that needs to be written during such foreground flushes. The persisted-index reported by a node as a response to this request is at least as high as the update-index of the item. When the flush finishes on a majority, the durable-index will be updated, and thus the data item can be served. The fourth column of Figure 2 shows how this procedure works. As shown, the durability check fails when item b is read; the nodes thus flush all updates upto index 2 and so the durability-index advances; the item is then served.

As an optimization, ORCA also persists writes that are after the last update to the item being read. Consider the log $[a,b,c]$ in Figure 2; when a client reads b, the durability check fails. Now, although it is enough to persist entries up to b, CAD also flushes update c, obviating future synchronous flushes when c is read as shown in the last column of the figure.

To summarize, CAD makes data durable upon reads and so guarantees that state that has been read will never be lost even if servers crash and recover. We next discuss how upon this durability primitive we build cross-client monotonic reads.

4.4 Cross-Client Monotonic Reads

If reads are restricted only to the leader, a design that many linearizable systems adopt, then cross-client monotonic reads is readily provided by CAD; no additional mechanisms are needed. Given that updates go only through the leader, the leader will have the latest data, which it will serve on reads (if necessary, making it durable before serving). Further, if the current leader fails, the new view will contain the state that was read. Thus, monotonic reads are ensured across failures.

However, restricting reads only to the leader limits read scalability and prevents clients from reading at nearby replicas. Most practical systems (e.g., MongoDB, Redis), for this reason, allow reads at many nodes [31, 33, 45]. However, when allowing reads at the followers, CAD alone cannot ensure cross-client monotonic reads. Consider the scenario in Figure 3. The leader S_1 has served versions a_1 and a_2 after
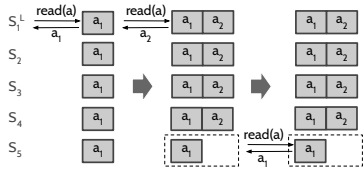
Figure 3: **Non-monotonic Reads.** *The figure shows how non-monotonic states can be exposed atop* CAD *when reading at the followers.*

Figure 4: **Active Set and Leases.** *(i) shows how removing a follower hastily can expose non-monotonic states; (ii) shows how* ORCA *breaks leases.*

making them durable on a majority. However, follower S_5 is partitioned and so has not seen a_2. When a read later arrives at S_5, it is possible for S_5 to serve a_1; although S_5 checks that a_1 is durable, it does not know that a has been updated and served by others, exposing non-monotonic states. Thus, additional mechanisms are needed which we describe next.

4.4.1 Scalable Reads with Active Set

A naive way to solve the problem shown in Figure 3 is to make the data durable on all the followers before serving reads from the leader. However, such an approach would lead to poor performance and, more importantly, decreased availability: reads cannot be served unless all nodes are available. Instead, ORCA solves this problem using an *active set*. The active set contains *at least* a majority of nodes. ORCA enforces the following rules with respect to the active set.

R1: When the leader intends to make a data item durable (before serving a read), it ensures that the data is persisted and applied by *all* the members in the active set.

R2: Only nodes in the active set are allowed to serve reads.

The above two rules together ensure that clients never see non-monotonic states. **R1** ensures that all nodes in the active set contain all data that has been read by clients. **R2** ensures that only such nodes that contain data that has been previously read can serve reads; other nodes that do not contain the data that has been served (e.g., S_5 in Figure 3) are precluded from serving reads, preventing non-monotonic reads. The key challenge now is to maintain the active set correctly.

4.4.2 Membership using Leases

The leader constantly (via heartbeats and requests) informs the followers whether they are part of the active set or not. The active-set membership message is a lease [12, 18] provided by the leader to the followers: if a follower F believes that it is part of the active set, it is guaranteed that no data will be served to clients without F persisting and applying the data. The lease breaks when a follower does not hear from the leader for a while. Once the lease breaks, the follower cannot serve reads anymore. The leader also removes the follower from the active set, allowing the leader to serve reads by making data durable on the updated (reduced) active set.

To ensure correctness, a follower must mark itself out *before* the leader removes it from the active set. Consider the scenario in Figure 4(i), which shows how non-monotonic states can be exposed if the leader removes a disconnected follower from the active set hastily. Initially, the active set contains all the nodes, and so upon a read, the leader tries to make
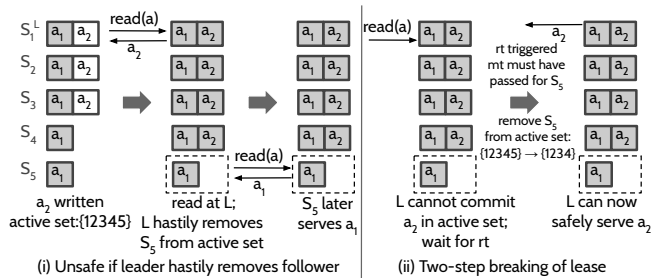
a_2 durable on all nodes; however, follower S_5 is partitioned. Now, if the leader removes S_5 (before S_5 marks itself out) and serves a_2, it is possible for S_5 to serve a_1 later, exposing out-of-order states. Thus, for safety, the leader must wait for S_5 to mark itself out and then only remove S_5 from the active set, allowing the read to succeed.

ORCA breaks leases using a two-step mechanism: first, a disconnected follower marks itself out of the active set; the leader then removes the follower from the active-set. ORCA realizes the two-step mechanism using two timeouts: a mark-out timeout (*mt*) and a removal timeout (*rt*); once *mt* passes, the follower marks itself out; once *rt* passes, the leader removes the follower from the active set. ORCA sets *rt* significantly greater than *mt* (e.g., $rt >= 5 * mt$) and *mt* is set to the same value as the heartbeat interval. Figure 4(ii) illustrates how the two-step mechanism works in ORCA. The performance impact is minimal when the leader waits to remove a failed follower from the active set. Specifically, only reads that access (recently written) items that are not durable yet must wait for the active set to be updated; the other vast majority of reads can be completed without any delays.

Like any lease-based system, ORCA requires non-faulty clocks with a bounded drift [18]. By the time *rt* passes for the leader, *mt* must have passed for the follower; otherwise, non-monotonic states may be returned. However, this is highly unlikely because we set *rt* to a multiple of *mt*; it is unlikely for the follower's clock to run too slowly or the leader's clock to run too quickly that *rt* has passed for the leader but *mt* has not for the follower. In many deployments, the worst-case clock drift between two servers is as low as 30 *μs*/sec [17] which is far less than what ORCA expects. Note that ORCA requires only a bounded drift, not synchronized clocks.

When a failed follower recovers (from a crash or a partition), the leader adds the follower to the active set. However, the leader ensures that the recovered node has persisted and applied all entries up to the durable-index before adding the node to the active set. Sometimes, a leader may break the lease for a follower G even when it is constantly hearing from G, but G is operating slowly (perhaps due to a slow link or disk), increasing the latency to flush when a durability check fails. In such cases, the leader may inform the follower that it needs to mark itself out and then the leader also removes the follower from the active set.

The size of the active set presents a tradeoff between scalability and latency. If many nodes are in the active set, reads can be served from them all, improving scalability; however, reads that access recently written non-durable data can incur more latency because data has to be replicated and persisted on many nodes. In contrast, if the active set contains a bare majority, then data can be made durable quickly, but reads can be served only by a majority.

Deposed leaders. A subtle case that needs to be handled is when a leader is deposed by a new one, but the old leader does not know about it yet. The old leader may serve some old data that was updated and served by the other partition, causing clients to see non-monotonic states. ORCA solves this problem with the same lease-based mechanism described above. When followers do not hear from the current leader, they elect a new leader but do so after waiting for a certain timeout. By this time, the old leader realizes that it is not the leader anymore, steps down, and stops serving reads.

4.5 Correctness

ORCA never returns non-monotonic states, i.e., a read from a client always returns at least the latest state that was previously read by any client. We now provide a proof sketch for how ORCA ensures correctness under all scenarios.

First, when the current leader is functional, if a non-durable item (whose update-index is L) is read, ORCA ensures that the state at least up to L is persisted on all the nodes in the active set before serving the read. Thus, reads performed at any node in the active set will return at least the latest state that was previously read (i.e., up to L). Followers not present in the active set may be lagging but reads are not allowed on them, preventing them from serving an older state. When a follower is added to the active set, ORCA ensures that the follower contains state at least up to L; thus any subsequent reads on the added follower will return at least the latest state that was previously read, ensuring correctness. When the leader removes a follower, ORCA ensures that the follower marks itself out before the leader returns any data by committing it on the new reduced set, which prevents the follower from returning any older state.

When the current leader fails, ORCA must ensure that latest state that was read by clients survives into the new view. We argue that this is ensured by how elections work in ORCA (and in many majority-based systems). Let us suppose that the latest read has seen state up to index L. When the leader fails and subsequently a new view is formed, the system must recover all entries at least up to L for correctness; if not, an older state may be returned in the new view. The followers, on a leader failure, become candidates and compete to become the next leader. A candidate must get votes from at least a majority (may include self) to become the leader. When requesting votes, a candidate specifies the index of the last entry in its log. A responding node compares the incoming index (P) against the index of the last entry in its own log (Q). If the node has more up-to-date data in its log than the candidate (i.e., $Q > P$), then the node does *not* give its vote to the candidate. This is a property ensured by many majority-based systems [2, 6, 39] which ORCA preserves.

Because ORCA persists the data on all the nodes in the active set and given that the active set contains at least a majority of nodes, at least one node in any majority will contain state up to L on its disk. Thus, only a candidate that has state at least up to L can get votes from a majority and become the leader. In the new view, the nodes follow the new leader's state. Given that the leader is guaranteed to have state at least up to L, all data that have been served so far will survive into the new view, ensuring correctness.

4.6 Implementation

We have built ORCA by modifying ZooKeeper (v3.4.12). We have two baselines. First, ZooKeeper with synchronous replication but asynchronous persistence (i.e., ZooKeeper with *forceSync* disabled). Second, ZooKeeper with asynchronous replication; we modified ZooKeeper to obtain this baseline.

In ZooKeeper, write operations either create new key-value pairs or update existing ones. As we discussed, ORCA follows the same code path of the baseline for these operations. In addition, ORCA replicates and persists updates constantly in the background. Read operations return the value for a given key. On a read, ORCA performs the durability check (by comparing the key's update-index against the system's durable-index) and enforces durability if required.

ORCA incurs little metadata overheads compared to unmodified ZooKeeper to perform the durability check. Specifically, ZooKeeper already maintains the last-updated index for every item (as part of the item itself [9]) which ORCA reuses. Thus, ORCA needs to additionally maintain only the durable-index, which is 8 bytes in size. However, some systems may not maintain the update indexes; in such cases, CAD needs eight additional bytes for every item compared to the unmodified system, a small price to pay for the performance benefits.

Performing the durability check is simple in ZooKeeper because what item a request will read is explicitly specified in the request. However, doing this check in a system that supports range queries or queries such as "get all users at a particular location" may require a small additional step. The system would need to first tentatively execute the query and determine what all items will be returned; then, it would enforce durability if one or more items are not durable yet.

We modified the replication requests and responses as follows. The followers include the persisted-index in their response and the leader sends the followers the durable-index in the requests or heartbeats. These messages are also used to maintain the active-set lease. We set the durable-index as the maximum index that has been persisted and applied by all nodes in the active set. We set the follower mark-out timeout to the same value as the heartbeat interval (100 ms in our implementation). We set the removal timeout to 500 ms.
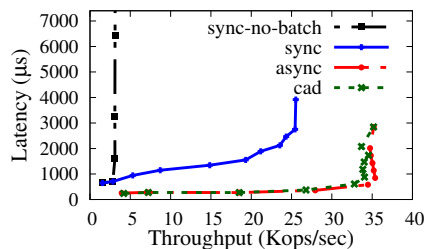
Figure 5: **Write-only Workload: Latency vs. Throughput.** *The figure plots the average latency against throughput by varying the number of clients for a write-only workload for different durability layers.*

5   Evaluation

In our evaluation, we ask the following questions:

- How does CAD perform compared to synchronous and asynchronous durability?
- How does ORCA perform compared to weakly consistent ZooKeeper and strongly consistent ZooKeeper?
- Does ORCA ensure cross-client monotonic reads in the presence of failures?
- Does ORCA provide better guarantees for applications?

We conduct a set of experiments to answer these questions. We run our performance experiments with five replicas. Each replica is a 20-core Intel Xeon CPU E5-2660 machine with 256 GB memory running Linux 4.4 and uses a 480-GB SSD to store data. The replicas are connected via a 10-Gbps network. We use six YCSB workloads [15] that have different read-write ratios and access patterns: W (write-only), A (w:50%, r:50%), B (w:5%, r:95%), C (read-only), D (read latest, w:5%, r:95%), F (read-modify-write:50%, r:50%). We do not run YCSB-E because ZooKeeper does not support range queries. Numbers reported are the average over five runs.

5.1   CAD Performance

We first evaluate the performance of the durability layer in isolation; we compare CAD against synchronous and asynchronous durability. With asynchronous durability and CAD, the system performs both replication and persistence asynchronously. With synchronous durability, the system replicates and persists writes (using *fsync*) on a majority in the critical path; it employs batching to improve performance.

5.1.1   Write-only Micro-benchmark

We first compare the performance for a write-only workload. Intuitively, CAD should outperform synchronous durability and match the performance of asynchronous durability for such a workload. Figure 5 shows the result: we plot the average latency seen by clients against the throughput obtained when varying the number of closed-loop clients from 1 to 100. We show two variants of synchronous durability: one with batching and the other without. We show the no-batch variant only to illustrate that it is too slow and we do *not* use this variant for comparison; throughout our evaluation, we compare only against the optimized synchronous-durability

Workload	Throughput (Kops/s)			% of reads triggering durability in CAD
	sync	async	CAD	
A	10.2	35.3 (3.5×)	33.7 (3.3 ×)	5.1 (of 50% reads)
B	23.1	39.4 (1.7×)	38.7 (1.7 ×)	0.83 (of 95% reads)
D	23.3	40.1 (1.7×)	36.9 (1.6 ×)	4.32 (of 95% reads)
F	11.8	35.7 (3.0×)	34.6 (2.9 ×)	4.07 (of 67% reads)

Table 2: **CAD Performance.** *The table compares the throughput of the three durability layers; the numbers in parenthesis in columns 3 and 4 are the factor of improvement over synchronous durability. The last column shows the percentage of reads that trigger synchronous durability in CAD.*

variant that employs batching.

We make the following three observations from the figure. First, synchronous durability with batching offers better throughput than the no-batch variant; however, even with aggressive batching across 100 clients, it cannot achieve the high throughput levels of CAD. Second, writes incur significantly lower latencies in CAD compared to synchronous durability; for instance, at about 25 Kops/s (the maximum throughput achieved by synchronous durability), CAD's latency is 7× lower. Finally, CAD's throughput and latency characteristics are very similar to that of asynchronous durability.

5.1.2   YCSB Macro-benchmarks

We now compare the performance across four YCSB workloads that have a mix of reads and writes. A, B, and F have a zipfian access pattern (most operations access popular items); D has a latest access pattern (most reads are to recently modified data). We run this experiment with 10 clients. We restrict the reads only to the leader for all three systems as we are evaluating only the durability layers. Table 2 shows the result.

Compared to synchronous durability with batching, CAD's performance is significantly better. CAD is about 1.6× and 3× faster than synchronous durability for read-heavy workloads (B and D) and write-heavy workloads (A and F), respectively.

CAD must ideally match the performance of asynchronous durability. First, performance of writes in CAD should be identical to asynchronous durability; making data durable on reads should not affect writes. Figure 6(a) shows this aspect for YCSB-A; results are similar for other workloads too.

Second, most read operations in CAD must experience latencies similar to reads in asynchronous durability. However, reads that access non-durable items may trigger synchronous replication and persistence, causing a reduction in performance. This effect can be seen in the read latency distributions shown in Figure 6(b) and 6(c). As shown, a fraction of reads (depending upon the workload) trigger synchronous durability and thus incur higher latencies. However, as shown in Table 2, for the variety of workloads in YCSB, this fraction is small. Therefore, the drop in performance for CAD compared to asynchronous durability is little (2% – 8%).

A bad workload for CAD is one that predominantly reads recently written items. Even for such a workload, the percentage of reads that actually trigger synchronous durability is small due to prior reads that make state durable and periodic background flushes in CAD. For example, with YCSB-D,
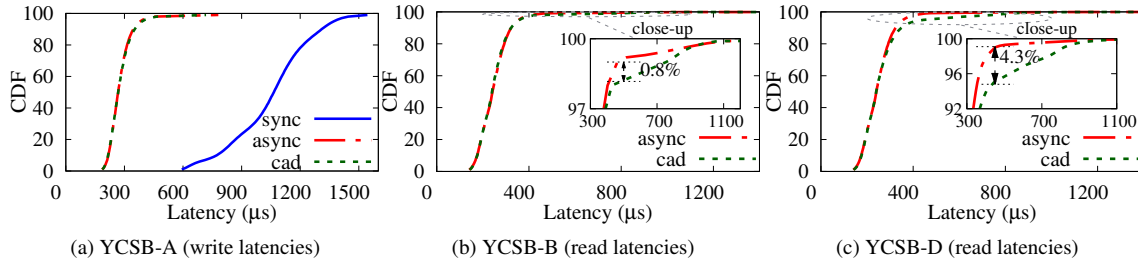
(a) YCSB-A (write latencies)   (b) YCSB-B (read latencies)   (c) YCSB-D (read latencies)

Figure 6: **Operation Latencies.** *(a) shows the latency distribution of writes in YCSB-A for the three durability layers. (b) and (c) show read latencies for for async and CAD in YCSB-B and YCSB-D; the annotation within a close-up shows the percentage of reads that trigger synchronous durability in CAD.*
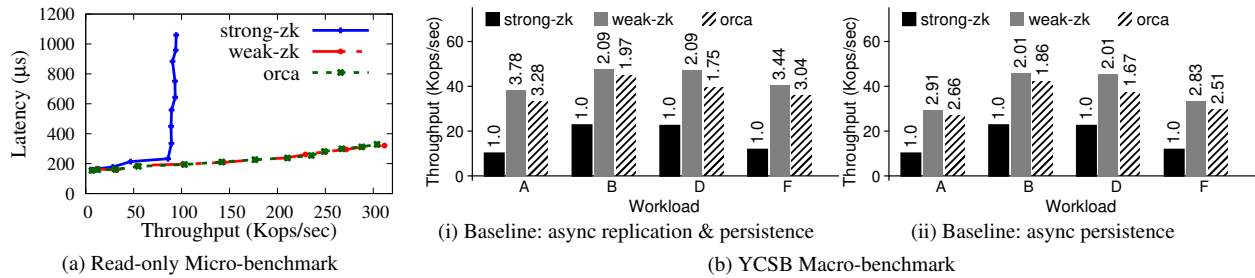

(a) Read-only Micro-benchmark   (i) Baseline: async replication & persistence   (ii) Baseline: async persistence

(b) YCSB Macro-benchmark

Figure 7: **ORCA Performance.** *(a) plots the average latency against throughput by varying the number of clients for a read-only workload for the three systems. (b) compares the throughput of the three systems across different YCSB workloads. In (b)(i), weak-ZK and ORCA asynchronously replicate and persist; in (b)(ii), they replicate synchronously but persist data lazily. The number on top of each bar shows the performance normalized to that of strong-ZK.*

although 90% of reads access recently written items, only 4.32% of these requests trigger synchronous replication and persistence; thus, CAD's overhead compared to asynchronous durability is little (only 8%).

CAD performance summary. CAD is significantly faster than synchronous durability (that is optimized with batching) while matching the performance of asynchronous durability for many workloads. Even for workloads that mostly read recently modified items, CAD's overheads are small.

5.2 ORCA System Performance

We now evaluate the performance of ORCA against two versions of ZooKeeper: strong-ZK and weak-ZK. Strong-ZK is ZooKeeper with synchronous durability (with batching), and with reads restricted to the leader; strong-ZK provides linearizability and thus cross-client monotonic reads. Weak-ZK replicates and persists writes asynchronously, and allows reads at all replicas; weak-ZK does not ensure cross-client monotonic reads. ORCA uses the CAD durability layer and reads can be served by all replicas in the active set; we configure the active set to contain four replicas in our experiments.

5.2.1 Read-only Micro-benchmark

We first demonstrate the benefit of allowing reads at many replicas using a read-only benchmark. Figure 7(a) plots the average latency against the read throughput for the three systems when varying the number of clients from 1 to 100. Strong-ZK restricts reads to the leader to provide strong guarantees, and so its throughput saturates after a point; with many concurrent clients, reads incur high latencies. Weak-ZK allows reads at

many replicas and so can support many concurrent clients, leading to high throughput and low latency; however, the cost is weaker guarantees as we show soon (§5.3). In contrast, ORCA provides strong guarantees while allowing reads at many replicas and thus achieving high throughput and low latency. The throughput of weak-ZK and ORCA could scale beyond 100 clients, but we do not show that in the graph.

5.2.2 YCSB Macro-benchmarks

We now compare the performance of ORCA against weak-ZK and strong-ZK across different YCSB workloads with 10 clients. Figure 7(b) shows the results.

In Figure 7(b)(i), weak-ZK and ORCA carry out both replication and persistence lazily; whereas, in 7(b)(ii), weak-ZK and ORCA replicate synchronously but persist to storage lazily, i.e., they issue *fsync*-s in the background. As shown in Figure 7(b)(i), ORCA is notably faster than strong-ZK (3.04 – 3.28× for write-heavy workloads, and 1.75 – 1.97× for read-heavy workloads). ORCA performs well due to two reasons. First, it avoids the cost of synchronous replication and persistence during writes. Second, it allows reads at many replicas, enabling better read throughput. ORCA also closely approximates the performance of weak-ZK: ORCA is only about 11% slower on an average. This reduction arises because reads that access non-durable items must persist data on all the nodes in the active set (in contrast to only a majority as done in CAD); further, reads at the followers that access non-durable data incur an additional round trip because they are redirected to the leader. Similar results and trends can be seen for the asynchronous-persistence baseline in Figure 7(b)(ii).
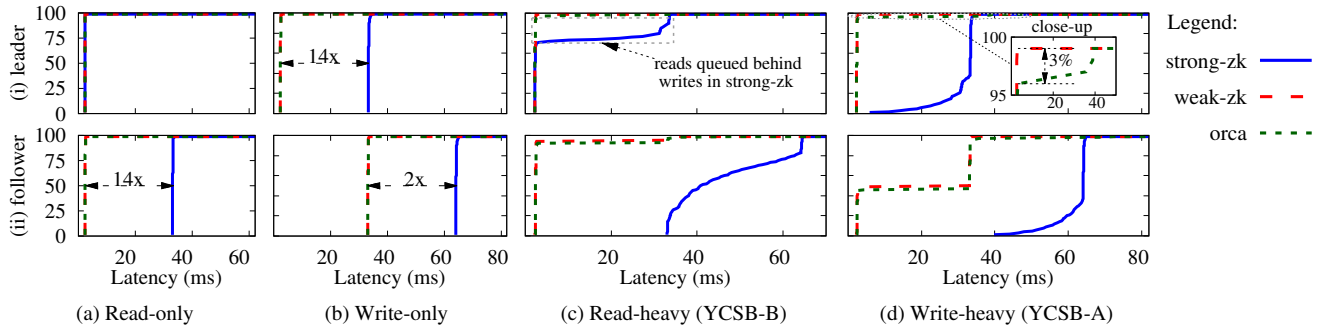
Figure 8: **Geo-distributed Latencies.** *(i) shows the distribution of latencies for operations originating near the leader; (ii) shows the same for requests originating near the followers. The ping latency between a client and its nearest replica is <2ms; the same between the client and a replica over WAN is ∼35 ms.*

5.2.3 Performance in Geo-Replicated Settings

We now analyze the performance of ORCA in a geo-replicated setting by placing the replicas in three data centers (across the US), with no data center having a majority of replicas. The replicas across the data center are connected over WAN. We run the experiments with 24 clients, with roughly five clients near each replica. In weak-ZK and ORCA, reads are served at the closest replica; in strong-ZK, reads go only to the leader. In all three systems, writes are performed only at the leader.

Figure 8 shows the distribution of operation latencies across different workloads. We differentiate two kinds of requests: ones originating near the leader (the top row in the figure) and ones originating near the followers (the bottom row). As shown in Figure 8(a)(i), for a read-only workload, in all systems, reads originating near the leader are completed locally and thus experience low latencies (∼2 ms). Requests originating near the followers, as shown in 8(a)(ii), incur one WAN RTT (∼33 ms) to reach the leader in strong-ZK; in contrast, weak-ZK and ORCA can serve such requests from the nearest replica and thus incur 14× lower latencies.

For a write-only workload, in strong-ZK, writes originating near the leader must incur one WAN RTT (to replicate to a majority) and disk writes, in addition to the one local RTT to reach the leader. In contrast, in weak-ZK and ORCA, such updates can be satisfied after buffering them in the leader's memory, reducing latency by ∼14×. Writes originating near the followers in strong-ZK incur two WAN RTTs (one to reach the leader and other for majority replication) and disk latencies; such requests, in contrast, can be completed in one WAN RTT in weak-ZK and ORCA, reducing latency by ∼2×.

Figure 8(c) and 8(d) show the results for workloads with a read-write mix. As shown, in strong-ZK, most operations incur high latencies; even reads originating near the leader sometimes experience high latencies because they are queued behind slow synchronous writes as shown in 8(c)(i). In contrast, most requests in ORCA and weak-ZK can be completed locally and thus experience low latencies, except for writes originating near the followers that require one WAN RTT, an inherent cost in leader-based systems (e.g., 50% of operations in Figure 8(d)(ii)). Some requests in ORCA incur higher la-
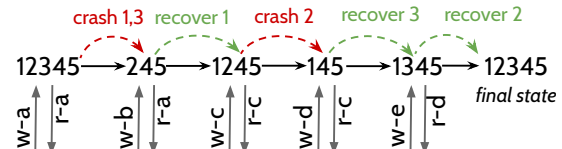

Figure 9: **An Example Failure Sequence.** *The figure shows an example sequence generated by our test framework.*

tencies because they read recently modified data. However, only a small percentage of requests experience such higher latencies as shown in Figure 8(d)(i).

ORCA performance summary. By avoiding the cost of synchronous replication and persistence during writes, and allowing reads at many replicas, ORCA provides higher throughput (1.8 − 3.3×) and lower latency than strong-ZK. In the geo-distributed setting, ORCA significantly reduces latency (14×) for most operations by allowing reads at nearby replicas and hiding WAN latencies with asynchronous writes. ORCA also approximates the performance of weak-ZK. However, as we show next, ORCA does so while enabling strong consistency guarantees that weak-ZK cannot offer.

5.3 ORCA Consistency

We now check if ORCA's implementation correctly ensures cross-client monotonic reads in the presence of failures and also test the guarantees of weak-ZK and strong-ZK under failures. To do so, we developed a framework that can drive the cluster to different states by injecting crash and recovery events. Figure 9 shows an example sequence. At first, all nodes are alive; then nodes 1, 3 crash; 1 recovers; 2 crashes; 3 recovers; finally, 2 recovers. In addition to crashing, we also randomly choose a node and introduce delays to it; such a lagging node may not have seen a few updates. For example, $\boxed{1}2345 \rightarrow 245 \rightarrow 1\boxed{2}45 \rightarrow 145 \rightarrow 134\boxed{5} \rightarrow 12345$ shows how nodes 1, 2, and 5 experience delays in a few states.

We insert new items at each stage and perform reads on the non-delayed nodes. Then, we perform a read on the delayed node, triggering the node to return old data, thus exposing non-monotonic states. Every time we perform a read, we check whether the returned result is at least as latest as the result of

System	Outcomes (%)		System	Outcomes (%)	
	Correct	Non-monotonic		Correct	Non-monotonic
weak-ZK	17	83	weak-ZK	4	96
strong-ZK	100	0	strong-ZK	100	0
sync-ZK-all	63	37	sync-ZK-all	63	37
ORCA	100	0	ORCA	100	0
(a) Async persistence			(b) Async replication & persistence		

Table 3: **ORCA Correctness.** *The tables show how ORCA provides cross-client monotonic reads. In (a), weak-ZK and ORCA use asynchronous persistence; in (b), both replication and persistence are asynchronous.*

any previous read. Using the framework, we generated 500 random sequences similar to the one in Figure 9. We subject weak-ZK, strong-ZK, and ORCA to the generated sequences.

Table 3(a) shows results when weak-ZK and ORCA synchronously replicate but asynchronously persist. With weak-ZK, non-monotonic reads arise in 83% of sequences due to two reasons. First, read data is lost in many cases due to crash failures, exposing non-monotonic reads. Second, delayed followers obliviously serve old data after other nodes have served newer state. Strong-ZK, by using synchronous durability and restricting reads to the leader, avoids non-monotonic reads in all cases. Note that while synchronous durability can avoid non-monotonic reads caused due to data loss, it is not sufficient to guarantee cross-client monotonic reads. Specifically, as shown in the table, sync-ZK-all, a configuration that uses synchronous durability but allows reads at all nodes, does not prevent lagging followers from serving older data, exposing non-monotonic states. In contrast to weak-ZK, ORCA does not return non-monotonic states. In most cases, a read performed on the non-delayed nodes persists the data on the delayed follower too, returning up-to-date data from the delayed follower. In a few cases (about 13%), the leader removed the follower from the active set (because the follower is experiencing delays). In such cases, the delayed follower rejects the read (because it is not in the active set); however, retrying after a while returns the latest data because the leader adds the follower back to the active set. Similar results can be seen in Table 3(b) when weak-ZK and ORCA asynchronously replicate and persist writes.

5.4 Application Case Studies

We now show how the guarantees provided by ORCA can be useful in two application scenarios. The first one is a location-sharing application in which an user updates their location (e.g., a → b → c) and another user tracks the location. To provide meaningful semantics, the storage system must ensure monotonic states for the reader; otherwise, the reader might incorrectly see that the user went backwards. While systems that provide session-level guarantees can ensure this property within a session, they cannot do so across sessions (e.g., when the reader closes the application and re-opens, or when the reader disconnects and reconnects). Cross-client monotonic reads, on the other hand, provides this guarantee irrespective of sessions and failures.

Outcome(%)	Location-tracking			Retwis		
	weak-ZK	strong-ZK	ORCA	weak-ZK	strong-ZK	ORCA
Inconsistent	13	0	0	8	0	0
Consistent (old)	39	0	7	20	0	12
Consistent (latest)	48	100	93	72	100	88

Table 4: **Case Study: Location-tracking and Retwis.** *The table shows how applications can see inconsistent (non-monotonic), and consistent (old or latest) states with weak-ZK, strong-ZK, and ORCA.*

We test this scenario by building a simple location-tracking application. A set of users update their locations on the storage system, while another set of users reads those locations. Clients may connect to different servers over the lifetime of the application. Table 4 shows result. As shown, weak-ZK exposes inconsistent (non-monotonic) locations in 13% of reads and consistent but old (stale) locations in 39% of reads. In contrast to weak-ZK, ORCA prevents non-monotonic locations, providing better semantics. Further, it also reduces staleness because of prior reads that make state durable. As expected, strong-ZK never exposes non-monotonic or old locations.

The second application is similar to Retwis, an open-source Twitter clone [46]. Users can either post tweets or read their timeline (i.e., read tweets from users they follow). If the timeline is not monotonic, then users may see some posts that may disappear later from the timeline, providing confusing semantics [14]. Cross-client monotonic reads avoids this problem, providing stronger semantics for this application.

The workload in this application is read-dominated: most requests retrieve the timeline, while a few requests post new content. We thus use the following workload mix: 70% get-timeline and 30% posts, leading to a total of 95% reads and 5% writes for the storage system. Results are similar to the previous case study. Weak-ZK returns non-monotonic and stale timelines in 8% and 20% of get-timeline operations, respectively. ORCA completely avoids non-monotonic timelines and reduces staleness, providing better semantics for clients.

6 Discussion

In this section, we discuss how CAD can be beneficial for many current systems and deployments, and how it can be implemented in other classes of systems (e.g., leaderless ones).
Application usage. As we discussed, most widely used systems lean towards performance and thus adopt asynchronous durability. CAD's primary goal is to improve the guarantees of such systems. By using CAD, these systems and applications atop them can realize stronger semantics without forgoing the performance benefits of asynchrony. Further, little or no modifications in application code are needed to reap the benefits that CAD offers.

A few applications such as configuration stores [19] cannot tolerate any data loss and so require immediate synchronous durability upon every write. While CAD may not be suitable for this use case, a storage system that implements CAD can support such applications. For example, in ORCA, applications

can optionally request immediate durability by specifying a flag in the write request (of course, at the cost of performance).

CAD for other classes of systems. While we apply CAD to leader-based systems in this paper, the idea also applies to other systems that establish no or only a causal order of updates. However, a few changes compared to our implementation for leader-based systems may be required. First, given that there is no single update order, the system may need to maintain metadata for each item denoting whether it is durable or not (instead of a single durable-index). Further, when a non-durable item x is read, instead of making the entire state durable, the system may make only updates to x or ones causally related to x durable. We leave such extension as an avenue for future work.

7 Related Work

Consistency models. Prior work has proposed an array of consistency models and studied their guarantees, availability, and performance [10, 19, 27, 29, 30, 48–50]. Our work, in contrast, focuses on how consistency is affected by the underlying durability model. Lee et al., identify and describe the durability requirements to realize linearizability [25]. In contrast, we explore how to design a durability primitive that enables strong consistency with high performance.

Durability semantics. CAD's durability semantic has a similar flavor to that of a few local file systems. Xsyncfs [37] delays writes to disk until the written data is externalized, realizing high performance while providing strong guarantees. Similarly, file-system developers have proposed the O_RSYNC flag [22] that provides similar guarantees to CAD. Although not implemented by many kernels [22], when specified in open, this flag blocks read calls until the data being read has been persisted to the disk. BarrierFS' fbarrier [52] and OptFS' osync [13] provide delayed durability semantics similar to CAD; however, unlike CAD, these file systems do not guarantee that data read by applications will remain durable after crashes. Most of the prior work resolves the tension between durability and performance in a much simpler single-node setting and within the file system. To the best of our knowledge, our work is the first to do so in replicated systems and in the presence of complex failures (e.g., partitions).

Improving distributed system performance. Several approaches to improving the performance of replicated systems using speculation [19, 51], exploiting commutativity [35], and network ordering [40] have been proposed. However, these prior approaches do not focus on addressing the overheads of durability, an important concern in storage systems. ORCA avoids durability overheads by separating consistency from freshness: reads can be stale but never out-of-order. Lazy-Base [14] applies a similar idea to analytical processing systems in which reads access only older versions that have been fully ingested and indexed. However, such an approach often returns staler results than a weakly consistent system. In contrast, ORCA never returns staler data than a weakly consistent system; further, ORCA reduces staleness compared to weak systems by persisting data on many nodes upon reads (as shown by our experiments). SAUCR reduces durability overheads in the common case but compromises on availability for strong durability in rare situations (e.g., in the presence of many simultaneous failures) [1]. ORCA makes the opposite tradeoff: it provides better availability but could lose a few recent updates upon failures.

Cross-client monotonic reads. To the best of our knowledge, cross-client monotonic reads is provided only by linearizability [25, 38]. However, linearizable systems require synchronous durability and most prevent reads at the followers. ORCA offers this property without synchronous durability while allowing reads at many nodes. Gaios [11] offers strong consistency while allowing reads from many replicas. Although Gaios distributes reads across replicas, requests are still bounced through the leader and thus incur an additional delay to reach the leader. The leader also requires one additional round trip to check if it is indeed the leader, increasing latency further. In contrast, ORCA allows clients to directly read from the nearest replica, enabling both load distribution and low latency. ORCA avoids the extra round trip (to verify leadership) by using leases. ORCA's use of leases to provide strong consistency is not new; for example, early work on cache consistency in distributed file systems has done so [18].

8 Conclusion

In this paper, we show how the underlying durability model of a distributed system has strong implications for its consistency and performance. We present consistency-aware durability (CAD), a new approach to durability that enables both strong consistency and high performance. We show how cross-client monotonic reads, a strong consistency guarantee can be realized efficiently upon CAD. While enabling stronger consistency, CAD may not be suitable for a few applications that cannot tolerate any data loss. However, it offers a new, useful middle ground for many systems that currently use asynchronous durability to realize stronger semantics without compromising on performance.

Acknowledgments

References

[1] Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.

[2] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[3] Apache. ZooKeeper. https://zookeeper.apache.org/.

[4] Apache. ZooKeeper Configuration Parameters. https://zookeeper.apache.org/doc/r3.1.2/zookeeperAdmin.html#sc_configuration.

[5] Apache. ZooKeeper Guarantees, Properties, and Definitions. https://zookeeper.apache.org/doc/r3.2.2/zookeeperInternals.html#sc_guaranteesPropertiesDefinitions.

[6] Apache. ZooKeeper Leader Activation. https://zookeeper.apache.org/doc/r3.2.2/zookeeperInternals.html#sc_leaderElection.

[7] Apache. ZooKeeper Overview. https://zookeeper.apache.org/doc/r3.5.1-alpha/zookeeperOver.html.

[8] Apache ZooKeeper. ZooKeeper Consistency Guarantees. https://zookeeper.apache.org/doc/r3.3.3/zookeeperProgrammers.html#ch_zkGuarantees.

[9] Apache ZooKeeper. ZooKeeper Programmer's Guide - ZooKeeper Stat Structure. https://zookeeper.apache.org/doc/r3.1.2/zookeeperProgrammers.html#sc_zkStatStructure.

[10] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, New York, NY, June 2013.

[11] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, April 2011.

[12] Randal C Burns, Robert M Rees, and Darrell DE Long. An Analytical Study of Opportunistic Lease Renewal. In *International Conference on Distributed Computing Systems (ICDCS '01)*, Phoenix, AZ, April 2001.

[13] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[14] James Cipar, Greg Ganger, Kimberly Keeton, Charles B Morrey III, Craig AN Soules, and Alistair Veitch. LazyBase: Trading Freshness for Performance in a Scalable Database. In *Proceedings of the EuroSys Conference (EuroSys '12)*, Bern, Switzerland, April 2012.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IA, June 2010.

[16] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, August 1987.

[17] 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 *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, April 2018.

[18] Cary G. Gray and David Cheriton. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, Litchfield Park, Arizona, December 1989.

[19] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[20] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), July 1990.

[21] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '10)*, Boston, MA, June 2010.

[22] Jonathan Corbet. O_*SYNC. https://lwn.net/Articles/350219/.

[23] Karthik Ranganathan. Low Latency Reads in Geo-Distributed SQL with Raft Leader Leases. https://blog.yugabyte.com/low-latency-reads-in-geo-distributed-sql-with-raft-leader-leases/.

[24] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[25] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[26] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT CSAIL, 2012.

[27] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[28] LogCabin. LogCabin. https://github.com/logcabin/logcabin.

[29] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[30] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI '17)*, Boston, MA, March 2017.

[31] MongoDB. MongoDB Read Preference. https://docs.mongodb.com/manual/core/read-preference/.

[32] MongoDB. MongoDB Replication. https://docs.mongodb.org/manual/replication/.

[33] MongoDB. Non-Blocking Secondary Reads. https://www.mongodb.com/blog/post/mongodb-40-nonblocking-secondary-reads.

[34] MongoDB. Read Concern Linearizable. https://docs.mongodb.com/manual/reference/read-concern-linearizable/.

[35] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[36] Seyed Hossein Mortazavi, Bharath Balasubramanian, Eyal de Lara, and Shankaranarayanan Puzhavakath Narayanan. Toward Session Consistency for the Edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, Boston, MA, July 2018.

[37] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[38] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014.

[39] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, June 2014.

[40] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, March 2015.

[41] David Ratner, Peter Reiher, Gerald J Popek, and Geoffrey H Kuenning. Replication Requirements in Mobile Environments. *Mobile Networks and Applications*, 6(6):525–533, 2001.

[42] Redis. Redis. http://redis.io/.

[43] Redis. Redis Persistence. https://redis.io/topics/persistence.

[44] Redis. Redis Replication. http://redis.io/topics/replication.

[45] Redis. Scaling Reads. https://redislabs.com/ebook/part-3-next-steps/chapter-10-scaling-redis/10-1-scaling-reads/.

[46] Retwis. Retwis. https://github.com/antirez/retwis.

---

[47] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[48] Doug Terry. Replicated Data Consistency Explained Through Baseball. *Communications of the ACM*, 56(12):82–89, 2013.

[49] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems (PDIS '94)*, Autin, TX, September 1994.

[50] Paolo Viotti and Marko Vukolić. Consistency in nontransactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, June 2016.

[51] Benjamin Wester, James Cowling, Edmund B Nightingale, Peter M Chen, Jason Flinn, and Barbara Liskov. Tolerating Latency in Replicated State Machines through Client Speculation. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, MA, April 2009.

[52] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO Stack for Flash Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, CA, February 2018.