PolyStore: Exploiting Combined Capabilities of Heterogeneous Storage

Yujie Ren, *Rutgers University and EPFL;* David Domingo, Jian Zhang, and Paul John, *Rutgers University;* Rekha Pitchumani, *Samsung Semiconductor Inc.;* Sanidhya Kashyap, *EPFL;* Sudarsun Kannan, *Rutgers University*

https://www.usenix.org/conference/fast25/presentation/ren

This paper is included in the Proceedings of the 23rd USENIX Conference on File and Storage Technologies.

February 25–27, 2025 • Santa Clara, CA, USA

ISBN 978-1-939133-45-8

PolyStore: Exploiting Combined Capabilities of Heterogeneous Storage

Yujie Ren*† David Domingo* Jian Zhang* Paul John* Rekha Pitchumani‡
Sanidhya Kashyap† Sudarsun Kannan*

*Rutgers University †EPFL ‡Samsung Semiconductor Inc.

Abstract

With the "non-hierarchical" trend in emerging storage media, *the philosophy of hierarchy* inevitably falls short in fully leveraging the combined bandwidth of multiple devices. In this paper, we propose a *horizontally structured* storage architecture that leverages the combined capabilities of heterogeneous devices. We introduce **PolyStore**, a meta layer atop storage medium-optimized file systems that spans userspace and the OS, allowing applications to access multiple storage devices concurrently with transparent, fine-grained data placement. PolyStore maximizes cumulative storage bandwidth and reduces hardware and software bottlenecks without compromising important properties such as sharing and security. Our evaluations show that PolyStore achieves 1.11x- 9.38x performance gains for micro-benchmarks and 1.52x- 2.02x for real-world applications across various device configurations.

1 Introduction

Advancements in storage media have led to a diverse range of performance characteristics, from byte-addressable low-latency persistent memory (PM) [6] and NVMe SSDs [63] with high bandwidth [47], to CXL-based SSDs with low latency and high bandwidth [68], and massive capacity devices like SATA SSDs. As hardware evolves, applications such as large databases [11], data streaming [54], and graph processing [39, 60], require high bandwidth from storage devices. At this critical juncture of hardware evolution and application demands, harnessing the collective capabilities of these *heterogeneous storage devices* (henceforth referred to as **HSDs**)is critical for performance and scalability potential.

In exploring storage heterogeneity within a machine, prior works follow the *philosophy of hierarchy*. We categorize them into three broad categories: (1) *caching*, (2) *tiering*, and (3) *application-directed* approaches. Caching solutions [1, 43, 44, 64] stack storage devices into a hierarchy, where updates and frequently accessed data are cached in faster storage, backed by slower storage on the lower level. Tiering solutions [22, 23, 38, 42, 50, 56, 62, 71] also adopt a hierarchical storage design, but the data is maintained exclusively on faster and slower storage tiers. They periodically assess data usage metrics (e.g., hotness) for deciding on data placement across tiers. Finally, application-directed approaches [32, 42, 57, 70] use dedicated placement policies for HSDs for application-specific logics and requirements. Although such design philosophy is central and common to the design of computer systems [17, 53], it suffers from three major limitations when we consider the "non-hierarchical" trend with modern HSDs [64].

First, the performance of those hierarchical approaches places faster devices on top of the hierarchy, thereby inherently preventing applications utilizing the combined performance of all devices. Although state-of-the-art caching design supports concurrent reads after saturating the faster storage layer [64], it still performs writes in a hierarchical manner, resulting in under-utilization of the write bandwidth.

Second, the hierarchical architecture prioritizes faster storage over slower storage, which incurs contention on a single device for multi-threaded I/O-intensive applications [4, 11] at both the software [64] and hardware levels [67]. Moreover, placing data eagerly on faster storage can cause excessive cache eviction and data migration between faster and slower devices, consuming part of the storage bandwidth.

Finally, using DRAM to cache and buffer data for hiding storage latency is still important, even for faster storage devices like PM [43, 74]. However, in the context of HSDs, traditional DRAM caching (e.g., Linux page cache) and DRAM+PM approaches [43, 74] are static as they do not adapt to varied performance gaps across HSDs.

To address these limitations, we argue that a *horizontally structured storage architecture* for HSDs has several benefits in (1) exploring cumulative storage capabilities with device bandwidth and capacity, (2) utilizing mature hardware-optimized file systems for individual device, and (3) enriching the DRAM cache to address the diversity of characteristics of HSDs. To that end, we propose **PolyStore**, a novel meta layer between applications and the existing OS storage stack that arranges HSDs in a non-hierarchical layout transparently and exploits the capability of their hardware-optimized file systems. PolyStore distributes large and bandwidth-intensive files across multiple devices at a fine-grained level to exploit cumulative storage bandwidth, while maintaining small and latency-sensitive files within a single storage. PolyStore splits the responsibility for managing HSDs between user and kernel space. The user-level runtime is responsible for mapping data across HSDs, while the OS component handles access protection, data sharing, and leverages mature hardware-optimized file systems.

However, realizing a horizontal architecture for HSDs with diverse performance and durability characteristics pose the following challenges: The first one is squeezing the cumulative bandwidth of devices while adapting to hardware and workload changes without modifying applications. The second challenge stems from diverse performance characteristics across HSDs that complicate effective DRAM buffering mechanisms and policies. Finally, ensuring crash consistency and recovery across HSDs is not straightforward due to the varying atomicity and durability guarantees of storage media.

PolyStore addresses these challenges with the following contributions: (1) **A heterogeneity-aware runtime for scaling horizontally** with support for concurrent indexing, file system operations, and dynamic data placement mechanism. The placement mechanism maps application threads and their updates across storage devices in a dynamic way to maximize cumulative bandwidth and minimize contention and data movement costs to adapt to workload changes in the long run. (2) **A heterogeneity-aware DRAM caching mechanism** that buffers data in DRAM on top of the horizontal layout of HSDs to hide performance variations, reducing kernel traps, lowering latency, and improving data plane throughput. (3) **A coordinated persistence mechanism** that ensures data durability and crash consistency by integrating runtime and OS components with hardware-optimized file systems.

Our evaluation shows PolyStore outperforms state-of-the-art caching [64] and tiering [38, 43, 62] solutions by up to 9.38x on PM/NVMe and 1.87x over NVMe/SATA configuration. Similarly, PolyStore's heterogeneity-aware DRAM caching improves throughput by up to 3.18x. Additionally, PolyStore improves throughput by up to 3.12x for metadata-intensive Filebench [59], and up to 2.94x on three applications over state-of-the-art systems.

2 Background and Related Work

We first present an overview of hardware trends and then discuss state-of-the-art software solutions.

2.1 Storage Hardware Trend

Modern storage devices have been evolving steadily with varying characteristics. For instance, modern non-volatile memory technologies [5, 6] offer lower access latency, while the evolution of PCIe [47] and the NVMe protocol [63] have enabled block-based SSDs to achieve high throughput.

Data growth necessitates heterogeneous storage for performance and cost efficiency [64]. However, managing devices

HSD Properties	Caching [43, 44, 64]	Tiering [38, 62, 71]	App-specific [32, 57, 69]	PolyStore
Cumulative read bandwidth utilization	*Orthus* ★	✗	✗	✓
Cumulative write bandwidth utilization	✗	✗	✗	✓
Heterogeneity-aware DRAM caching	*P2CACHE*	✗	✗	✓
Cross-device durability & crash consistency	✗	✓	✓	✓

Table 1: **Existing solutions vs. PolyStore.** ★*Orthus* [64] partially support cumulative read bandwidth utilization for HSDs.

with varied bandwidth, asymmetric read/write speeds, and diverse capacities complicates storage software [6, 63, 66].

2.2 Managing Storage Heterogeneity

To manage HSDs in a single system, state-of-the-art approaches follow *the philosophy of hierarchy* by placing faster devices above slower ones.

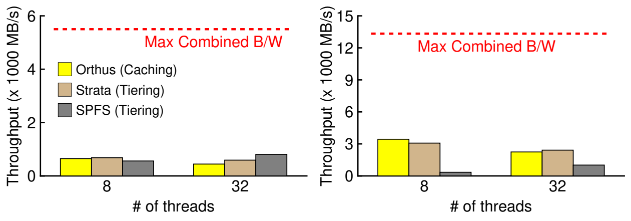
Caching uses faster storage to absorb writes and accelerate reads by storing frequently accessed data on faster storage backed by larger but slower storage [1, 13, 15, 20, 24, 27, 33, 35, 37, 45, 65]. For example, some systems [43, 44] use PM as a caching layer over SSD. Meanwhile, Orthus [64] considers non-hierarchical caching for HSDs. It proposes dynamic cache admission control that concurrently reads from slower storage and bypasses the fast caching device, but only when the bandwidth of fast storage is saturated, limiting the benefits of cumulative bandwidth for only read operations.

Tiering stacks HSDs as tiers, and maintain data exclusively at one layer and only migrate it across tiers based on certain policies (e.g., hotness). The designs include file systems [38, 42, 50, 62, 71] and user-level runtimes [22, 23, 56]. For instance, Strata [38] uses a combination of PM and SSD, first buffering data on PM as a log and then asynchronously digesting the logs to slower SSDs. In contrast, Ziggurat [71] writes across PM and SSD based on the access pattern, and SPFS [62] uses a lightweight file system for PM and stacks it on top of file systems for SSDs, such as ext4.

Application-directed data placement uses application-level knowledge to place data across HSDs [32, 42, 57, 70]. For example, key-value stores like *NoveLSM* [32] first write data to PM, then use background threads to compact data to SSDs. Similarly, DBMSes place write-ahead logs for faster storage and database files to slower storage [25, 58, 74].

3 Analyzing Hierarchy Pitfalls

We analyze the limitations of hierarchical approaches in managing HSDs. As summarized in Table 1, these approaches often over-prioritize or restrict application threads to the fast device at the top of the hierarchy, leaving unrealized device bandwidth on the table. To illustrate the limitation in terms of cumulative bandwidth utilization, we use a multi-threaded micro-benchmark on a multi-core system with PM and NVMe storage (*Config I* in Table 3), with the OS page cache disabled. We examine three state-of-the-art systems: (1) Orthus [64], a



(a) Random Write (b) Random Read

Figure 1: **Inefficient hardware utilization and software design.** Throughput for 32 threads (Direct I/O without DRAM cache) with each thread accessing its private 2GB file. The red line shows the combined device bandwidth of PM and NVMe.

non-hierarchical caching solution; (2) Strata [38], a full stack tiering solution with an user-level file system optimized for PM and NVMe; and (3) SPFS [62], a stackable PM file system that could be stacked on top of ext4 for NVMe. Figure 1 shows the extent of the failure of current approaches for utilizing cumulative bandwidth of HSDs.

For **write operations**, existing caching and tiering solutions force application threads to write to PM first. As the number of threads increases, hardware contention also increases, as studied in prior work [67]. For **read operations**, caching approaches move data from NVMe to PM when the data resides in NVMe. This approach not only limits application I/O throughput but also incurs additional write to PM. As mentioned above, writing to PM becomes a bottleneck with increasing threads. Hybrid caching solutions like Orthus only allow concurrent reads from PM and NVMe, but only when PM bandwidth is saturated. This saturation typically occurs during high cache hits to PM [64]. The tiering solution Strata also faces the same issue. Meanwhile, SPFS does not move frequently accessed data from NVMe to PM for read operations until a write operation executes [62]. Hence, for the random read workload, SPFS cannot take advantage of the PM read bandwidth due to the absence of the OS page cache in our configuration. Note that extending Orthus to support non-hierarchical writes across HSDs presents fundamental challenges, primarily due to its block-layer implementation, which cannot guarantee atomicity or durability [19, 36, 49].

Some caching and tiering systems have attempted to enhance performance of HSDs by incorporating DRAM caching, which mitigates asymmetrical read/write characteristics in PM [43, 71, 74]. However, these systems neglect a unified DRAM cache atop of HSDs, which could implement device-specific cache policies by considering hardware contention, latency, and available bandwidth across the HSDs.

4 Design and Implementation of PolyStore

We design PolyStore—a meta-layer atop device-optimized file systems. PolyStore manages data across one or more storage devices to utilize cumulative bandwidth without compromising important storage system properties.

4.1 PolyStore Design Goals

Goal 1: Attaining combined bandwidth of heterogeneous storage. To extract cumulative bandwidth, PolyStore distributes data within large files and between different files across HSDs and adapts to different workloads and storage hardware. Two principles drive this goal:

P1: *Provide heterogeneity-aware runtime to support fine-grained data indexing and file system operations.* For fine-grained distribution of data (blocks) of large and bandwidth-intensive files [39, 60] across multiple storage devices, a concurrent indexing mechanism and multi-storage file system operations must support parallel read and update operations to a file with minimal synchronization overhead.

P2: *Dynamic bandwidth-aware data placement across HSDs.* Exploiting cumulative storage bandwidth requires a dynamic data placement mechanism to the underlying HSDs that adapts to hardware and application workload changes.

Goal 2: Achieving heterogeneity-aware DRAM caching to lower latency. Unlike the OS page cache or existing solutions that do not consider storage heterogeneity, PolyStore should provide a unified DRAM cache support to address such asymmetrical bandwidth and latency gaps across HSDs.

P3: *Heterogeneity-aware DRAM caching with abstracted HSD storage layer.* The horizontal HSDs layout enables exposing the right set of abstractions to incorporate DRAM caching with storage-specific admission and eviction policies.

Goal 3: Providing unified durability, crash consistency, and sharing guarantees. Spreading data across HSDs non-hierarchically through fine-grained placement should not compromise atomicity, durability, crash consistency, and security guarantees. Two principles achieve this:

P4: *Coordinated persistence and recovery.* Atomicity, durability, and crash-consistency guarantees require coordinated data and metadata persistence across device-optimized file systems and HSD runtime with minimal overhead.

P5: *Secure metadata update delegation to the OS.* Sharing files across applications introduces the possibility of metadata corruption, which requires a trusted entity (OS).

4.2 Overview

For our first goal of attaining combined storage bandwidth across HSDs, we propose the following two components for ***P1*** and ***P2***: **(a)** A scalable data indexing (**Poly-index**) component utilizes a range-based tree structure *range-tree* to enable the ability to split data across underlying physical files in their respective storage. *Poly-index* enables concurrent access across storage devices with fine-grained range-level locks. **(b)** A dynamic data placement (**Poly-placement**) component that maps I/O requests from application threads dynamically to different storage, thereby preventing the over-saturation of any single storage device. For the second goal with ***P3***, we introduce **(c)** a heterogeneity-aware DRAM caching (**Poly-cache**) in user-space with versatile storage-specific cache admission and eviction control policies. For ensuring durability, crash-

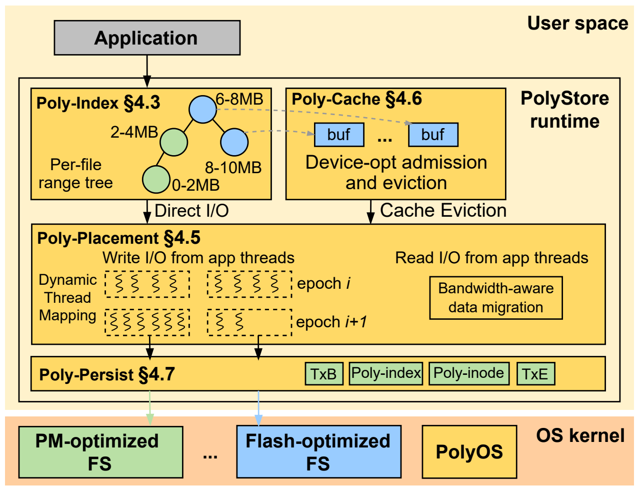


Figure 2: **PolyStore High-level Design.** Figure shows the overall flow of PolyStore when application threads perform I/O. Poly-index specifies the location of data blocks on physical devices; each index entry points to a DRAM buffer in Poly-cache. Poly-placement determines the device for appended blocks by mapping I/O threads to underlying devices to maximize combined bandwidth utilization. Poly-persist handles crash consistency across underlying devices. PolyOS is responsible for sharing, fairness, and security.

consistency, and sharing guarantees toward our third goal with ***P4*** and ***P5***, we propose **(d)** a coordinated persistence and consistency technique (**Poly-persist**), which spans across HSDs with differing atomicity and durability guarantees. **(e)** A thin OS component ensures secure sharing across applications.

We prototype PolyStore with a *cross-layered* design with a user-level runtime that intercepts POSIX I/O operations and a thin kernel component in the VFS layer (PolyOS) for enabling file sharing across processes. PolyStore accommodates a variety of storage types, along with their respective file systems [41, 66]. For multiple devices within the same type, PolyStore utilize its file system to scale (e.g., RAID [7, 46]). Figure 2 illustrates a high-level flow of operations across the components built from the techniques introduced above.

4.3 Structures for Fine-grained Placement

For utilizing the cumulative bandwidth of HSDs, PolyStore needs to distribute data across multiple storage media with low overhead. This requires a fine-grained placement mechanism that abstracts the complexity of data distribution across small files or within large bandwidth sensitive files [39, 60]. To achieve this goal, PolyStore introduces the following key structures with minimal locking overheads, low memory and storage footprints, and efficient namespace management.

Poly-inode. PolyStore exposes a logical file interface to applications, abstracting the complexity of data distribution. Each logical file is represented by a *Poly-inode*, which maintains a mapping between the logical file and one or more physical files in the underlying file systems. The Poly-inode contains essential information such as open file descriptors for per-device physical inodes and reference counters, enabling seamless management of data across HSDs.

Poly-index. To enable fine-grained block placement and efficient utilization of cumulative bandwidth, particularly crucial for large files like databases or streaming files, we introduce *Poly-index*. This scalable data indexing structure employs a *range-tree*, an augmented red-black tree indexed by interval ranges (low, high)[9], which allows for efficient look-ups and updates of non-overlapping block ranges [50].

Each Poly-index node is a unit of locking granularity, and covers a 2MB range by default (but configurable). We use 2MB range (1) to balance between memory capacity and concurrency; and (2) 2MB size aligns with Linux's maximum block I/O size for a single request, allowing the entire range to be evicted using a single request [21]. Each Poly-index node has its own readers-writer lock, promoting high concurrency and minimal locking overheads, in line with state-of-the-art file systems [40, 50, 73]. Each Poly-index node has a 128B memory and 96B disk footprint, resulting in only 64MB memory usage for a 1TB file (0.0064%).

As illustrated in Figure 2, Poly-index manages the logical blocks across physical files in HSDs. Additionally, Poly-index nodes maintain pointers to user-level DRAM cache buffers (§4.6). Both Poly-inode and Poly-index are stored in memory-mapped files on faster storage with the durability and crash consistency guarantee by Poly-persist (§4.7), enabling sharing across multiple processes with access permissions (§4.8).

Hybrid namespace management. To efficiently manage namespaces across multiple storage devices, PolyStore implements a dual-path name resolution strategy. For data-intensive files spread across HSDs, which require less frequent name resolution, PolyStore uses a directory hierarchy in fast storage and a flat-name structure on slower devices. This approach minimizes random accesses and expedites resolution for files spanning multiple devices. For example, a multi-storage logical file ($/polyfs/d1/f1$) is mapped to a physical file on faster storage using a conventional directory hierarchy (e.g., $/fast/d1/f1$ on PM), while the physical file on slower storage adopts a flat name path on the root using the hash value of the logical path (e.g., $/slow/hash(”polyfs/d1/f1”)$).

4.4 Heterogeneity-aware File System Ops

PolyStore 's horizontal scaling for exploiting the combined bandwidth of HSDs for data plane operations introduces design challenges that, when addressed, reduce overhead and improve performance for metadata-heavy operations (e.g., file creation), ensure ordering (e.g., file appends), and enable concurrent reads while preserving POSIX semantics.

4.4.1 Adaptive File Creation and Opening

Since small files rarely benefit from cumulative storage bandwidth, PolyStore implements an adaptive, on-demand approach to multi-storage file creation. Initially, PolyStore creates a physical file on one device with its file system, along with a Poly-inode and Poly-index for the logical file. As file size grows and storage bandwidth saturates, PolyStore dynamically creates new physical files on different devices, mapping them to the same Poly-inode and dynamically places additional blocks to the other devices. This strategy utilizes cumulative bandwidth for large files while minimizing overhead for small files (default to 2MB matching Poly-index node size but configurable), reducing metadata overhead across file systems. When applications create or open a file, PolyStore returns a logical file descriptor on which the application performs subsequent data plane operations.

4.4.2 File Writes to Exploit Cumulative Bandwidth

To exploit cumulative storage bandwidth, PolyStore efficiently manages write operations across HSDs under two main scenarios: when application threads perform writes concurrently on a large shared file or when each application thread writes to its own file. These scenarios require both concurrent write management and the flexibility to map application threads to different storage devices.

Write indexing: When an application thread performs a write, it traverses the Poly-index to locate nodes covering the relevant block ranges based on offset and size. This process addresses three key scenarios: (1) For ranges not indexed, PolyStore creates new nodes on demand and issues I/O to the appropriate file systems of the target device. (2) For partially existing ranges, PolyStore performs I/O for the entire range, adding missing nodes as necessary. (3) For fully indexed ranges, PolyStore updates only the modified blocks in place to reduce write amplification and improve efficiency.

Mapping write threads to HSDs: To map application threads performing writes and distribute file blocks across HSDs, PolyStore begins with a static approach that determines offline the maximum number of threads required to saturate the bandwidth of each storage type using widely-used benchmark tools [14]. PolyStore then prioritizes the assignment of the maximum number of threads that can exploit the bandwidth of the fastest storage, with any remaining application threads assigned to slower devices, to exploit the cumulative bandwidth. For bandwidth-intensive large files, this results in block ranges of a file being spread across multiple storage devices, whereas for smaller files, the writes generally end up on a single storage device. Finally, in PolyStore, a single write operation's blocks are always written to a single storage for ordering and crash consistency. In §4.5, we describe our dynamic placement design for handling changes in I/O behavior and access patterns.

4.4.3 Preserving POSIX Semantics

Appends. PolyStore tackles the challenge of maintaining POSIX-compliant appends [8] across HSDs with a dual-pronged approach. First, it ensures serialized appends by opening all relevant physical files in append mode when a logical file is accessed, enforcing ordered writes within each device. Second, PolyStore leverages Poly-index to coordinate appends across multiple threads and physical files, maintaining global order. Timestamps in the Poly-index range tree further optimize DRAM caching (discussed in §4.6) by ensuring that when block blk_N is evicted, all preceding blocks (blk_0 to blk_{N-1}) are also evicted, preserving data integrity. In case of failure, Poly-index and timestamps are used to recover the correct order and truncate updates following an unrecoverable block. This ensures PolyStore provides the ordering and consistency of traditional file systems while capitalizing on the performance advantages of HSD.

Concurrent Reads. To efficiently utilize bandwidth and concurrency for reads, application threads use Poly-index to quickly identify block ranges spread across one or more files on multiple devices, enabling concurrent read I/Os on HSDs even during ongoing writes.

4.4.4 Other Metadata Operations

PolyStore tackles complex metadata operations, such as `rename` and `unlink`, by ensuring the same correctness and crash-consistency guarantees as traditional OS file systems [34, 75]. For files spanning multiple devices, PolyStore locks the Poly-inode to ensure atomicity, updates metadata for each physical file via the underlying file system, and commits PolyStore metadata to faster storage upon success. For small files on a single device, PolyStore minimizes overhead by directly updating the physical file's metadata using the file system. Our evaluation shows that PolyStore 's bandwidth gains outweigh the overhead of additional PolyStore metadata updates for data-heavy files. Even for metadata-heavy workloads, PolyStore offers gains through its concurrent design.

4.5 Bandwidth-aware Placement & Migration

PolyStore must adapt to the dynamic and fluctuating I/O behavior and changing access patterns of modern applications. For write/update operations, as described in §4.4, the initial simple static mapping of I/Os from application threads to HSDs is insufficient to maximize cumulative bandwidth under workload changes. However, finding the optimal solution for this mapping across HSDs is challenging, comparable to solving an NP-Complete knapsack problem [30, 48, 52].

Similarly, for read operations, timely migration of frequently accessed "hot" data to faster devices is essential to prevent performance degradation. PolyStore aims to achieve these goals without compromising cumulative data placement objectives or incurring excessive data movement.

4.5.1 Dynamic Thread Mapping for Writes

We design Poly-placement, which uses a greedy strategy inspired by previous research [30, 48, 52]. At the beginning of the application (first epoch), Poly-placement starts with static placement as discussed in §4.4, mapping enough threads to saturate each storage device bandwidth starting with the fastest device (D_{high}), and assigning other threads to slower storage as shown in Algorithm 1.

Algorithm 1: Thread Mapping and Remapping in HSDs

```
Input: T[] is per-device thread set, M is total thread count
Input: D[] denotes device list, N is number of devices
Param: curr_state<$D_{low}$, $D_{high}$> denotes thread mapping state
       between a low and high bandwidth device
  low_to_high: remap thread from $D_{low}$ to $D_{high}$ (initial state)
  high_to_low: remap thread from $D_{high}$ to $D_{low}$
  STABLE: stabilized throughput yielding highest throughput
Param: $B_{dev/global}$ denotes per-device/global bandwidth usage
Param: $X_{dev/global}$ denotes a threshold (configurable and default to
       50%) of bandwidth change for thread remapping

1  Function PolyPlacement($T$[], $D$[], $N$):
2      for $i = 0$; $i < N - 1$; $i = i + 1$ do
3          DynamicPlacement($T$[], $D$[], $i$)
4      end
5  End Function
6  Function DynamicPlacement($T$[],$D$[],$i$):
7      $high = i$, $low = i + 1$
8      if curr_state<$D_{low}$, $D_{high}$> == low_to_high then
9          if $B_{high}$ improved by $X_{high}$ from last epoch then
10             move a thread from $T[low]$ to $T[high]$
11         else if $B_{high}$ dropped by $X_{high}$ from last epoch then
12             move a thread from $T[high]$ to $T[low]$
13             curr_state<$D_{low}$, $D_{high}$> = high_to_low
14         else
15             curr_state<$D_{low}$, $D_{high}$> = STABLE
16     else if curr_state<$D_{low}$, $D_{high}$> == high_to_low then
17         if $B_{high}$ improved by $X_{high}$ from last epoch then
18             move a thread from $T[high]$ to $T[low]$
19         else if $B_{high}$ dropped by $X_{high}$ from last epoch then
20             move a thread from $T[low]$ to $T[high]$
21             curr_state<$D_{low}$, $D_{high}$> = low_to_high
22         else
23             curr_state = STABLE
24     else if curr_state<$D_{low}$, $D_{high}$> == STABLE then
25         if $B_{high}$ improved by $X_{high}$ from last epoch then
26             return
27         else if $B_{high}$ dropped by $X_{high}$ from last epoch then
28             curr_state<$D_{low}$, $D_{high}$> = low_to_high
29         else if $B_{global}$ dropped by $X_{global}$ from last epoch then
30             reset to default/initial mapping
31 End Function
```

Algorithm 2: Bandwidth-Aware Data Migration

```
Input: D[] denotes device list, N is number of devices
Input: More[] and Less[] represent per-device hot and cold list
Input: Blk represents the data block being accessed
Input: devidx represents the device id of the data block Blk resides
Parameter: Blk.state denotes the access frequency of the
           Poly-index node (2-bit states)
MORE_FREQUENT, LESS_FREQUENT (initial value)
1  Function BW_Move($D$[], devidx, $N$):
2      if Blk.state == LESS_FREQUENT then
3          Move Blk from the Less[devidx] to More[devidx]
4      if Blk.state == MORE_FREQUENT then
5          for $i = 0$; $i < devidx$; $i = i + 1$ do
6              if $BW_{write}[i]$ AND $BW_{read}[i]$ are not saturated AND
                D[i].freeSpace > low_water_mark then
7                  Migrate from $D_{devidx}$ to $D_i$
8                  Move Blk from the More[devidx] to Less[i]
9                  return
10         end
11     if D[devidx].freeSpace < low_water_mark then
12         Launch a BG thread executing BG_Move($D$[], devidx, $N$)
13 End Function
14 Function CAP_Move($D$[], devidx, $N$):
15     while D[devidx].freeSpace < high_water_mark do
16         for $i = devidx + 1$; $i < N$; $i = i + 1$ do
17             if $BWwrite_i$ AND $BWread_i$ are not saturated then
18                 Migrate from $D_{devidx}$ to $D_i$
19                 Move Blk from the Less[devidx] to Less[i]
20                 return
21         end
22     end
23 End Function
```

Poly-placement then continuously profiles I/O activities in configurable epochs (200ms by default), tracking global and per-device bandwidth and capacity utilization, application's read and write I/O throughput, and thread-level metrics like I/O sizes. Poly-placement also uses Poly-index to keep track of access frequencies of blocks.

After the initial epoch, Poly-placement begins in a `low_to_high` remapping state, migrating threads to saturate the faster device (D_{high}) with higher bandwidth (line 10). If the faster device's throughput (B_{high}) increases beyond a threshold X_{high}, more threads are mapped to continue saturation (lines 10, 20). However, if moving threads to D_{high} decreases throughput significantly, thread remapping moves to a `high_to_low` state, indicating threads be remapped to slower storage to try and alleviate potential contention (lines 12, 18). When remapping yields minimal improvement, the system enters a `STABLE` state (lines 15, 23) where no remapping occurs unless application bandwidth significantly decreases or falls below the cumulative HSD bandwidth (lines 29-30).

4.5.2 Bandwidth and Capacity-aware Migration

Application access patterns continuously evolve and storage hardware performance and capacity vary, making data migration inevitable. To balance migration costs and read performance, PolyStore employs two forms of migration: a bandwidth-aware and a capacity-aware migration.

Mechanism: The data migration includes the following steps. (1) Append data to the target device. (2) Update the Poly-index to point to the new data location. (3) Truncate the source file and mark the data block migrated as garbage. Migrating data across HSDs potentially suffers from data inconsistency during power loss. To protect PolyStore from inconsistency issues, PolyStore wraps the data migration into a transaction (described shortly in §4.7). Data migration leaves garbage data across HSDs. PolyStore performs background garbage collection by either punching holes using `fallocate` (if the file system supports) or by compacting a fragmented file and updating the Poly-index.

Bandwidth-aware Migration (`BW_Move`): Thread mapping and remapping in Poly-placement for cumulative bandwidth utilization may result in scenarios where threads frequently access hot data from physical files stored on slower storage. In such cases, if the data is not stored in the DRAM cache of the slower device, the bandwidth of the slower device could become saturated and contended, incurring the under-utilization of read bandwidth of faster storage (the case of SPFS analyzed in §3). This necessitates moving hot data from slower to faster storage. However, migration should only occur if the device bandwidth is not already saturated.

PolyStore utilizes its Poly-index to track hot ranges and blocks, and Poly-cache (to be discussed shortly in §4.6) to determine if blocks are already cached before initiating migration. As shown in Algorithm 2, each block range must meet a minimum access count threshold before moving (50% by default empirically but configurable). Frequently accessed(*MORE_FREQUENT*) blocks are first moved to the fastest storage, followed by other storage devices (lines 4-10). As an added optimization, with a unified DRAM cache across all fast and slow devices (§4.6), data movement can be avoided for frequently accessed data already in the cache.

Capacity-aware Migration (`CAP_Move`): When the capacity of faster storage exceeds a watermark (a configurable 90% of a storage capacity similar to Linux [29]), PolyStore triggers a background migration, progressively evicting blocks from faster to slower storage (lines 14-20). As showcased in §5, `BW_Move` and `CAP_Move` are effective for applications that are bandwidth and capacity-intensive applications (e.g., GraphWalker [60]) and scales when using multiple storage devices (e.g., PM, NVMe, and SATA SSD). To further reduce migrations, an effective optimization is to place cold blocks below a certain access threshold to the slowest storage.

4.6 Heterogeneity-aware DRAM Caching

Existing solutions of adapting DRAM caching to the storage hierarchy in HSDs mainly focus on mitigating the read/write performance asymmetry for PM [43, 71, 74]. However, with the horizontal layout for HSDs in PolyStore, these approaches lack the ability of providing device-specific cache admission and eviction policies in a holistic way.

To address these limitations, we propose Poly-cache, a heterogeneity-aware DRAM caching component for HSDs. Poly-cache improves scalability through parallel cache admission and eviction, leveraging cumulative storage bandwidth, and hardware-specific cache admission and eviction policies. We outline the Poly-cache mechanisms, followed by its flexibility to support hardware-specific caching policies.

4.6.1 Userspace Caching with Concurrent Poly-index

Poly-cache adopts a user-level design for performance and flexibility [28]. It bypasses the OS page cache, eliminating redundant caching and reducing kernel trap overhead for I/O operations when cache hits. It implements a unified DRAM cache for logical files placed across one or more devices with fine-grained concurrency control. Furthermore, the user-level cache makes it easier to employ device-specific cache admission and eviction policies, allowing dynamic optimization based on storage characteristics and workload patterns.

To ensure Poly-cache is scalable and highly concurrent, we utilize Poly-index, where each tree node contains per-range metadata and links to a DRAM buffer of same size to the configurable range size, which is configurable but set to 2MB by default, using a maximum of 512 DRAM pages of 4KB. In memory-rich systems, to reduce allocation costs, Poly-cache uses huge pages (e.g., 2MB page) for each range. For file writes, using Poly-index, the DRAM cache buffers corresponding to one or more ranges are updated, followed by updating the node's metadata with dirty block information. For reads and overwrite operations spanning multiple ranges, on a cache miss, missing blocks are first loaded into Poly-cache, and then updated as necessary for overwrites.

4.6.2 Cache Evictions Exploiting Combined Bandwidth

Efficient cache eviction is critical given the limited memory available for caching. PolyStore uses *cgroups* to limit the maximum cache budget per application, preventing impact on other applications. Poly-cache employs a timestamp-based, two-level least-recently-used (LRU) eviction policy. Each Poly-index node is marked with a timestamp when its blocks are added to the cache. When flushing the cache during file commits (`fsync`), these timestamps dictate the order in which data ranges are committed to the disk. The two-level LRU consists of a file-level LRU that evicts inactive files, and a per-file range LRU that evicts LRU blocks within each file.

Cache evictions are performed by background threads maintained in a thread pool. Initially, Poly-cache allocates one eviction thread for each storage type. For files with blocks stored on multiple storage devices, multiple threads concurrently write back (for dirty blocks) and evict blocks to their respective underlying files. To further accelerate cache eviction and flushing, and utilize the cumulative bandwidth of multiple storage devices, our bandwidth monitoring mechanism profiles the bandwidth use and increases the number of eviction threads based on available CPUs and bandwidth.

Finally, the horizontal layout of HSDs in PolyStore with Poly-cache opens the opportunity of evicting (i.e., migrating) blocks to different storage devices regardless of their physical file location, further enabling dynamic optimization of data placement as described in §4.5.

4.6.3 Hardware-specific Poly-cache Policies

The user-level design of Poly-cache offers flexibility in implementing new caching policies. We present two simple-but-effective policies that address hardware performance variations across different storage types:

Bypassing Cache for PM Reads: PM reads provide substantially higher bandwidth and lower latency compared to writes. Introducing a DRAM cache layer for reads, unlike for writes, increases data copy overheads without yielding performance gains [18, 32, 66] (§3). To address this performance disparity, we implement a policy where blocks/ranges are exclusively read from PM, bypassing Poly-cache, and directly memory-mapped until a write modifies the range. This approach reduces PM to DRAM data copy overheads for reads while accelerating slower PM writes with DRAM caching.

Dynamic Adjustment of Cache Ratio for SSDs: Considerable bandwidth and latency variations exist among similar storage media with the same access granularity but different interfaces and vendors (e.g., NVMe SSD vs. SATA SSD). To narrow this performance gap when using HSDs with horizontal scaling, we implement a flexible admission control policy that dynamically adjusts cache ratios between devices, assigning a higher cache ratio to slower storage based on hardware and software metrics (e.g., device bandwidth and latency) and the ratio of data access between faster and slower storage.

4.7 Coordinated Persistence

Spreading data blocks of a logical files to HSDs in PolyStore introduces challenges of maintaining atomicity, crash-consistency and durability across HSDs with their file systems. Varying atomicity and durability guarantees across file systems [31] complicate satisfying these properties.

To address the challenges, PolyStore develops **Poly-persist** with a split design: it ensures the atomicity and durability of its runtime states, such as Poly-index nodes and Poly-inode by leveraging the durability guarantee provided by the underlying OS file systems in PolyStore.

Atomicity: To ensure atomicity for complex file system operations like rename, link, and truncate, PolyStore augments the underlying physical file systems' atomicity capabilities with its own journaling mechanism. This journal comprises of a global log of uncommitted files, per-file metadata, and an operational log detailing operations and their commit status. A commit operation involves a transaction that first serially commits to the storage-specific file system, then updates PolyStore metadata in Poly-inode and Poly-index nodes. For example, during an atomic file `rename`, PolyStore initiates a transaction and executes the operation on the underlying file system, updating the operational log. In case of a crash or transaction abort, PolyStore uses the operational log for recovery, attempting to retry the transaction. If recovery fails (e.g., file system corruption), PolyStore marks the affected files as inaccessible. These files can then be repaired using tools like *fsck* [2], similar to OS file system.

Crash-consistency and Recovery: PolyStore journals its runtime metadata (Poly-inode and Poly-index) to a dedicated partition on faster storage, similar to state-of-the-art multi-device storage systems [38, 71]. When using Persistent Memory (PM) as fast storage, we optimize the Poly-inode log entries to fit in one cache line, committing them through atomic PM writes. Poly-index's range nodes are optimized to fit in two cache lines, committed in a transaction using `clflush` and memory barriers [31, 66]. Without PM, PolyStore stores metadata in memory-mapped files on block storage and uses `msync` to persist log entries. During recovery, PolyStore first waits for the underlying file systems to complete their recovery during mount, and then uses the global log to identify all uncommitted files. Finally, it examines the metadata journal and operational log to checkpoint committed updates to the underlying file systems, excluding uncommitted ones.

Common Minimal Multi-Storage Durability: File systems and storage devices have different levels of durability. For example, PM file systems like NOVA offer robust metadata and data durability, while block-based ext4 disables data journaling by default to reduce overhead. Due to the precise placement of blocks and files in PolyStore, durability levels may vary within the same file or across files. To address this variability, PolyStore establishes a uniform baseline durability level across file systems, even if it is less stringent than the strongest level offered by any individual file system. For example, in configurations using multiple file systems such as NOVA for PM and ext4 for NVMe, PolyStore maintains metadata-only durability.

4.8 Sharing and Security

PolyStore's split design ensures security and multi-process file sharing guarantees similar to state-of-the-art user-level file systems [31, 38, 50, 72]. When multiple applications share the devices, PolyOS utilizes Linux's I/O throttling mechanism in Linux [51] to ensure the fairness of I/O bandwidth. The access control guaranteed by the OS kernel ensures PolyStore updates data and metadata (e.g., Poly-index) in a secure manner. However, PolyStore must address a potential vulnerability with the user-level Poly-cache design. If a malicious or buggy process with read-only permission for the physical file attempts to update Poly-inode by adding a cache buffer in user space, it could corrupt Poly-index.

To mitigate potential corruption and security risks inherent in most user-level designs, especially when sharing files, PolyStore leverages its split design to delegate Poly-index updates from user space to the PolyStore OS component. For instance, when a shared file is opened with read permissions by multiple processes, the PolyStore OS component implemented in the VFS layer is notified and revokes permission to the memory-mapped Poly-index file for all processes by raising an interrupt like state-of-the-art approach [72]. Consequently, all subsequent accesses and updates to Poly-index are exclusively handled by the OS component of PolyStore.

5 Evaluation

We evaluate PolyStore by addressing the following.

- How effective is Poly-placement in utilizing cumulative storage bandwidth of HSDs and its consequent impact in reducing I/O latency?
- How does Poly-cache compare to existing DRAM caches for HSDs and traditional OS page caches?
- Are PolyStore techniques beneficial in improving performance for macrobenchmarks and real applications?
- Can PolyStore quickly recover from failures?

Methodology: We evaluate PolyStore using three configurations described in Table 2. *Config I* uses PM and NVMe

Storage	Config I	Config II	Config III
Faster	256GB Optane PM	ITB NVMe SSD	256GB Optane PM
Slower	1TB NVMe SSD	2TB SATA SSD	1TB NVMe SSD
Slowest	-	-	2TB SATA SSD

Table 2: **Experimental Configurations.**

Approach	Description
PM-only (NOVA [66])	PM-only that uses mature and scalable NOVA [66]
NVMe-only (ext4)	uses NVMe SSD only with deployed ext4
SATA-only (ext4)	uses SATA SSD only with ext4
Orthus (Caching) [64]	fast storage as cache and slow as backend device
Strata (Tiering) [38]	PM as fast tier and NVMe as backend device
SPFS (Tiering) [62]	PM as fast tier and NVMe as backend device
P2CACHE [43]	uses DRAM to mitigate performance degrade in PM
PolyStore-static	PolyStore using naive static data placement
PolyStore-dynamic	PolyStore w/ dynamic data placement
PolyStore	PolyStore w/ dynamic placement and Poly-cache

Table 3: **PolyStore and alternatives evaluated.**

SSD, while *Config II* uses NVMe SSD and SATA SSD. We use the state-of-the-art NOVA [66] as the file system for PM and ext4 for both SATA and NVMe SSDs. The maximum write and read bandwidths are 4.6GB/s and 13.2GB/s for PM [67], 1.2GB/s and 1.2GB/s for NVMe [26], and 560MB/s and 530MB/s for SATA SSD [55], respectively.

We compare PolyStore against **single storage configurations** (PM-only, NVMe-only, and SATA-only), state-of-the-art **caching** (Orthus [64]), **tiering** (Strata [38], SPFS [62]), and **DRAM+PM** designs (P2CACHE [43]) as listed in Table 3. For PolyStore, we use the mature and well-tested NOVA [66] for PM and ext4 for the rest. Strata, SPFS, and P2CACHE are explicitly designed for using PM; hence, we excluded them from *Config II*. None of those approaches support more than two types of devices; therefore, we only use PolyStore to showcase horizontal scaling in *Config III* with three devices. We evaluate PolyStore on data-intensive microbenchmarks, metadata-heavy macrobenchmarks with smaller files (Filebench [59]), and real-world applications (RocksDB [11], Redis [10], and GraphWalker [60]).

5.1 Cumulative Storage Bandwidth

We evaluate PolyStore's efficacy in maximizing cumulative bandwidth of HSDs via a microbenchmark that bypasses the OS page cache with `O_DIRECT` flag. We contrast two methods: PolyStore-static, initially distributing threads evenly and statically across faster and slower storage (§4.4.2), and PolyStore-dynamic, featuring dynamic data placement (§4.5). We evaluate various access patterns and in-depth analyses, employing 4KB I/O sizes for a total workload size of 64GB on both *Config I* (Figure 3) and *Config II* (Figure 6).

Write througput and latency. Figure 3a shows the throughput for sequential write using *append* operation. First, single storage designs like PM-only and NVMe-only show poor throughput. PM-only's bandwidth saturates at 8 threads after which it degrades due to contention (discussed in §3). NVMe-only shows only marginal gains with higher threads from SSD's internal parallelism. State-of-the-art Orthus, a hybrid caching approach, and tiering solutions Strata and SPFS all fail to utilize multi-storage bandwidth for writes.

	Sequential Write	Sequential Read
Orthus	150.4	54.3
Strata	144.1	73.2
SPFS	123.2	121.4
PolyStore-static	85.2	84.1
PolyStore-dynamic	23.5	22.9

Table 4: **Microbenchmark with direct I/O on PM/NVMe (*Config I*)** average latency (μs) of 32 threads.

PolyStore-static employs fine-grained static placement of I/O requests, maximizing cumulative HSD bandwidth. It outperforms PM-only by 2.16x, NVMe-only by 3.25x, and Orthus by 4.23x. PolyStore-dynamic further optimizes bandwidth use through I/O remapping, saturating 92.3% of combined bandwidth with 32 threads. It achieves up to 1.48x gains over PolyStore-static and 9.38x over other approaches.

For random writes in *Config I* (Figure 3c), PolyStore-dynamic gains are marginal over PolyStore-static, for which the thread mapping is already configured to maximize bandwidth, which is also reflected in *Config II* (Figure 6a). As shown in Table 4, PolyStore also reduces the write latency by up to 6.38x through the effective use of HSDs. Existing approaches, which place PM at the top of the hierarchy, suffer from high write latency spikes at higher thread counts due to contention, aligning with observations in prior studies [67].

Dynamic data placement analysis. To gain additional insights into the benefits of dynamic data placement (PolyStore-dynamic) compared with baseline (PolyStore-static), in Figure 4a, we show per-epoch (200ms) I/O bandwidth (y-axis) analysis for sequential write with 32 threads in Figure 3a. The x-axis shows the increasing epoch tics. PolyStore-dynamic converges around 8–10 threads that saturate PM bandwidth, while the rest use NVMe. Unlike PolyStore-static, as some PM threads complete (at epochs 80 and 124, marked with blue dotted lines), PolyStore-dynamic is able to remap some or all running threads from NVMe to PM to maintain PM bandwidth saturation, leading to faster completion.

Read performance. Figure 3b shows sequential read results. PM-only shows stable throughput after 16 threads due to the saturation of PM's bandwidth. Orthus is 1.26x better than Strata before 8 threads, however it is not able to scale beyond because Orthus needs to admit data from NVMe to PM in random read workload, incurring concurrent PM writes as analyzed in §3. The breakdown shown in Figure 4b also indicates such data movement in Orthus. In contrast, PolyStore-dynamic outperforms PM-only at 32 threads by 1.11x, benefiting from the cumulative read bandwidth of both PM and NVMe. In addition, the performance breakdown shown in Figure 4b indicates the extra cost from using multiple file systems in PolyStore is balanced out by its benefits. In Table 4, PolyStore shows slightly higher read latency over PM-only because some reads require NVMe access. However, it still outperforms other approaches.

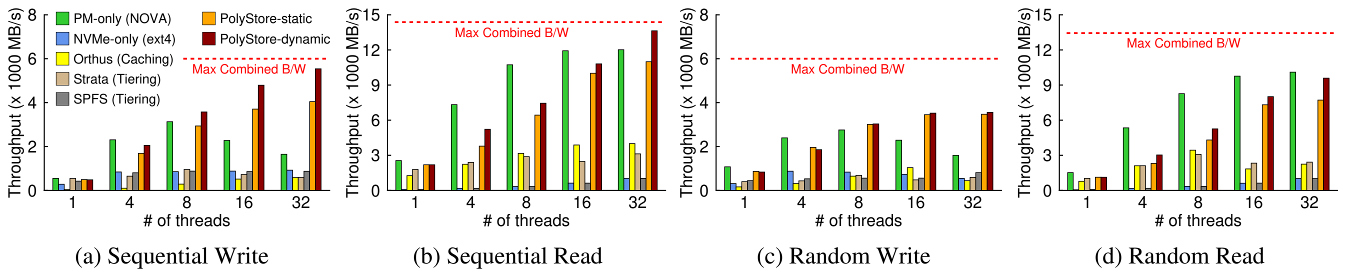


Figure 3: **Microbenchmark with direct I/O (without DRAM cache) on PM/NVMe (*Config I*).** Red dotted lines show maximum combined PM + NVMe write and read bandwidth. Threads access their individual files.

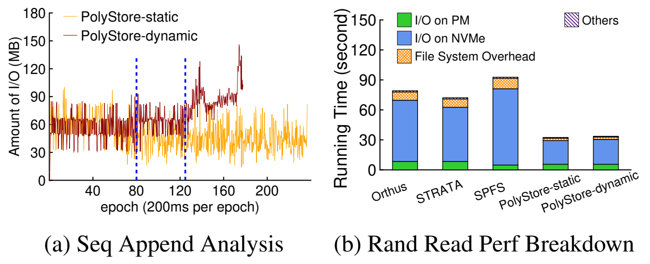


Figure 4: **Microbenchmark Analyses (*Config I*).** (a) The per-epoch I/O stat analyses for sequential write in Figure 3a. (b) The breakdown analyses for random read in Figure 3d.

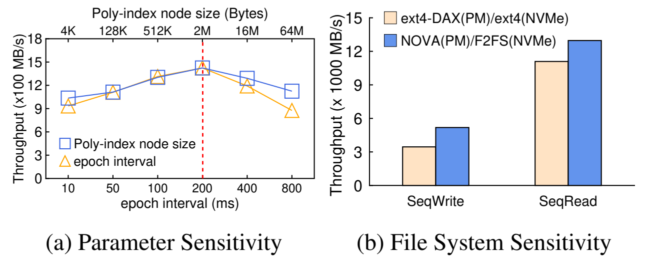


Figure 5: **Parameters and File Systems Sensitivity Study. (a) Epoch interval and Poly-index node size study in *Config II*. (b) File systems sensitivity study in *Config I*.**

Performance on systems without PM. To understand the reusability and effectiveness of PolyStore beyond a PM-based system, we evaluate PolyStore on *Config II*. The faster NVMe SSD is used for persisting PolyStore metadata on a memory-mapped file. As shown in Figure 6a, the aggregate use of NVMe and SATA SSDs in PolyStore provides 1.87x gains over NVMe-only. For reads, shown in Figure 6b, PolyStore outperforms Orthus by up to 2.23x.

Shared file access. We now study how PolyStore performs when threads share a file. 32 threads (16 readers and 16 writers) perform random 4KB I/O operations on a shared 64GB file. Figure 6c shows that Poly-index mitigates per-inode `rw-lock` overhead by distributing logical files across multiple physical files. This boosts performance by up to 2.95x for writers and 3.04x for readers compared to NVMe-only.

Scaling beyond 2 Devices. We use three devices (*Config III*) to showcase the ability of PolyStore to scale beyond two devices. As shown in Figure 6d, PolyStore utilizes up to 91.7% of the combined bandwidth of three devices.

5.2 Parameter Sensitivity

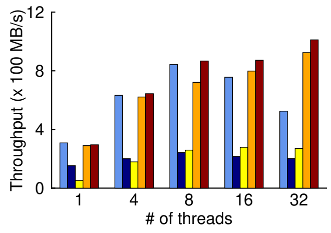
We study the parameter sensitivity in Poly-placement shown in Figure 5a to justify the default selection in PolyStore described in design sections. For the Poly-index node size, the smaller it is, the better bandwidth utilization of HSDs, but with the higher the Poly-index memory footprint. For the epoch interval, the trade-off lies in the bandwidth utilization efficiency and the profiling overhead in Poly-placement.

We also study the impact of file systems selected for PolyStore with different combinations for PM and NVMe in *Config I*. We use the following two configurations. (1) NOVA [66] for PM, F2FS [41] for NVMe. (2) ext4-DAX for PM, ext4 for NVMe. Figure 5b shows that using storage-optimized file systems gives better performance in all workloads. NOVA's per-CPU data structures and scalable log-structured design enables higher concurrency [66], thereby outperforms ext4-DAX. On the other hand, F2FS's zone-aware concurrent logging and flash-friendly file system layout makes it a superior choice over ext4 for flash-based storage [41]. As a result, PolyStore-dynamic using NOVA/F2FS provides up to 1.63x compared with using ext4-DAX/ext4.

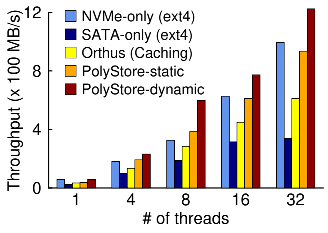
5.3 Scalable DRAM Cache atop HSDs

We evaluate the benefits of scalable user-level DRAM Poly-cache and compare against PM-only without any DRAM caching [61], and NVMe-only with OS page cache. Orthus enables OS page cache for slower storage, and Strata with only metadata cache in DRAM [38]. Lastly, P2CACHE adopts a PM+DRAM design to serve legacy storage, using PM as a cache for write durability with DRAM cache for reads.

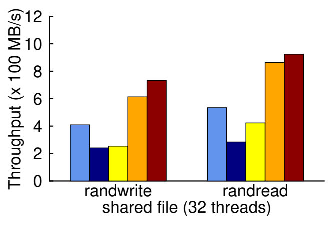
In Figure 7a and Figure 7b, we show the throughput for random write and read for PM/NVMe, with working set size fitting in the DRAM cache. At higher thread count, P2CACHE's write performance faces PM's scalability bottlenecks. In contrast, Poly-cache, with flexible cache admission that only buffers write but not read requests, shows up to 6.31x gains against baseline systems. In Figure 7c and Figure 7d, we assess *PolyStore*'s performance across different DRAM availability scenarios by varying the working set to memory size ratio on the x-axis. For example, a 16:1 ratio indicates a workload size 16 times larger than the available cache memory. In this configuration (16:1), when the cache is full, unlike



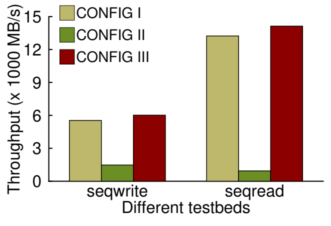
(a) Random Write (*Config II*)



(b) Random Read (*Config II*)

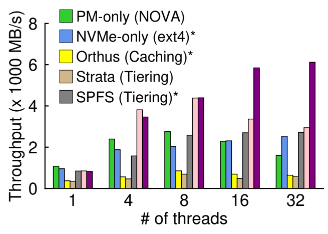


(c) File Sharing (*Config II*)

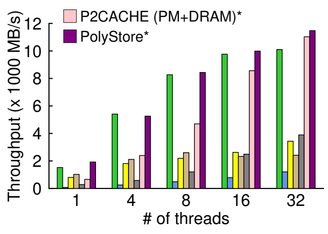


(d) Impact on three devices

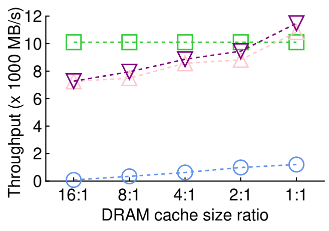
Figure 6: **Microbenchmark with direct-I/O on NVMe/SATA (*Config II*) and three devices (*Config III*).**



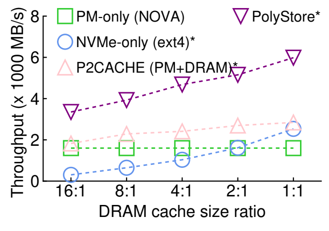
(a) Random Write



(b) Random Read

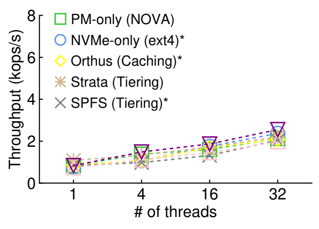


(c) Random Read

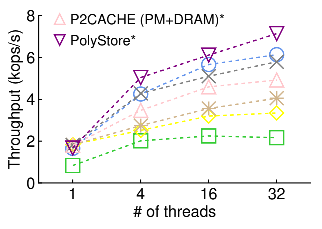


(d) Random Write

Figure 7: **DRAM Caching Impact and Sensitivity on PM/NVMe *(Config I)*.** (a) and (b) show random write and read performance with Poly-cache. (c) and (d) show the sensitivity to DRAM cache size for 32 threads. (* denotes systems using the DRAM cache).



(a) Varmail



(b) Fileserver

Figure 8: **Metadata-heavy Filebench Performance.** X-axis shows the thread count and the y-axis shows the throughput.

P2CACHE which uses PM as a write cache that leads to contention to PM, PolyStore's concurrent evictions to HSDs alleviates memory pressure, reducing stalls when waiting for available DRAM space. Consequently, for random write workloads, PolyStore improves 3.18x over PM-only and 2.21x over NVMe-only with OS page cache.

5.4 Impact on Metadata-heavy Workloads

To evaluate PolyStore with *metadata-heavy workloads and small files,* we use the widely-used **Filebench** [59] with the *Varmail* and *Fileserver* workloads. These workloads are metadata-heavy (e.g., file create, delete), constituting 69% (*Varmail*) and 63% (*Fileserver*) of the overall I/O. File sizes are small (512KB and 1MB), with data read-to-write ratios of 1:1 and 1:2 in *Varmail* and *Fileserver*, respectively.

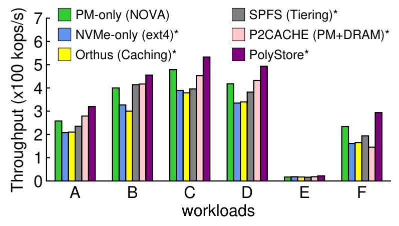
In Figure 8, even for metadata-heavy workloads, PolyStore outperforms single-storage file systems, achieving 3.12x gains over PM-only (without DRAM cache) for write-heavy *Fileserver*. Note that PolyStore's on-demand physical file creation (§4.4.1) allows small files to be placed on a single storage device without the file creation cost of two file systems. The marginal throughput gains over other approaches stem from PolyStore's user space DRAM cache (Poly-cache) that reduces system calls for data-plane operations [50].

5.5 Real-world Applications

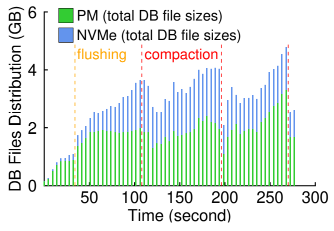
We select three applications: RocksDB [11], Redis [10], and GraphWalker [60] to evaluate PolyStore. RocksDB is an LSM-based key-value store with diverse read and write access patterns, including compaction, file commits, and rename operations. Redis is a storage-backed, in-memory key-value database that performs I/O for logging state to append-only-files (AOF) and asynchronously checkpoints in-memory key-values to backup files. GraphWalker is a state-of-the-art graph processing framework designed for fast and scalable random walks, targeting large graphs on storage in a single-machine. We first evaluate RocksDB as stand-alone for the horizontal scaling impact on HSD. Next, we analyze PolyStore' performance when handling multiple applications: RocksDB and Redis. Then, we investigate PolyStore's resilience in failure recovery. Finally, we study the performance of GraphWalker and insights of data migration footprints.

Horizontal scaling benefits with *YCSB*. We utilize *YCSB*, a set of real-world access patterns widely used in evaluations [16]. *YCSB* encompasses six access patterns, A-F, with varying read/write ratios following a Zipfian distribution. Our setup includes a value size of 512B, 10M keys, and 32 threads. RocksDB stores data in 128MB SST files.

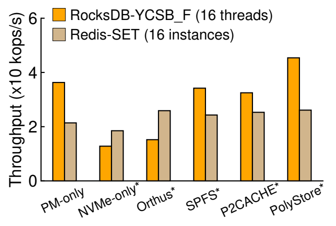
As shown in Figure 9a, PolyStore harnesses cumulative PM and NVMe bandwidth in write-heavy workloads A and F, achieving significant gains over state-of-the-art approaches: up to 1.52x and 2.02x, respectively. In workload F, with a 50% *read-modify-write* pattern, Poly-cache, as discussed in §4.6, plays a crucial role. It evicts blocks in DRAM cache initially placed on slower NVMe to faster PM (space permitting),



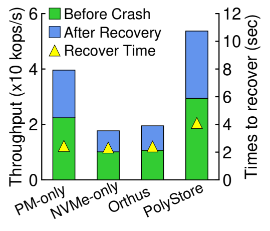
(a) RocksDB *YCSB*



(b) PolyStore's Placement Trace

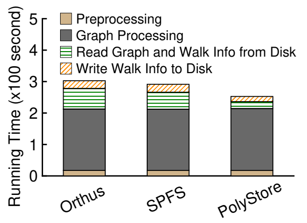


(c) Multi-Application

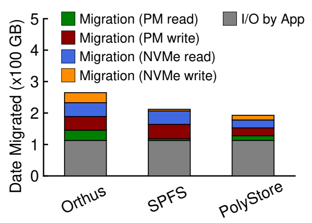


(d) Failure Recovery

Figure 9: **RocksDB and Redis on PM/NVMe (*Config I*).** Figures show (a) throughput of *YCSB* workloads with 32 threads, 10M keys with 512B; (b) HSD data placement trace for *YCSB-A*; (c) throughput for running RocksDB and Redis at the same time; (d) random-write throughput after recovering from failure, with right y-axis showing the recovery time. (* denotes systems using DRAM for caching.)



(a) Run Result



(b) Data Migration

Figure 10: **GraphWalker *MS-PPR*.** Total Graph size of 64GB, with DRAM size limited to 16GB, PMEM size to 32GB. (a) shows the breakdown of running result; (b) shows the migration footprints.

speeding up their reaccess. This allows PolyStore to achieve its best throughput gains in *YCSB F* workload compared to others. Figure 9b illustrates the DB files' footprint as a time series on PM and NVMe, showcasing PolyStore's ability to utilize the combined bandwidth of both storage types.

Multi-application performance. We study the performance impact in running multiple applications simultaneously with RocksDB and 16-instance multi-instance Redis. We do not restrict the bandwidth of these applications using the PolyStore OS component. As shown in Figure 9c, PolyStore achieves throughput gains over P2CACHE by 1.96x without degrading the performance for Redis. PolyStore OS layer, uses Linux's fair I/O sharing to dynamically adjust and throttle the bandwidths utilization across different applications (§4.8) .

Performance under failure. To demonstrate the correctness and efficient recovery under failures, we use the *fillrandom* workload in RocksDB's *db_bench* [12]. We randomly inject failures and measure recovery time and total throughput (Figure 9d). Leveraging Poly-persist's coordinated runtime and OS durability, RocksDB successfully recovers and continues execution. PolyStore prioritizes physical file recovery across multiple storages before Poly-inodes and Poly-trees, slightly extending recovery time. However, by utilizing cumulative bandwidth, it outperforms other systems by up to 2.91x.

Graph Processing with Large Graph. To study the data migration across HSDs in a long run with a large real-world dataset [3] that cannot fit into DRAM, we use the state-of-the-art graph processing framework GraphWalker with *MS-PPR* (multi-source personalize page rank) algorithm. We limit the DRAM size to 16GB and PM size to 32GB to fully trigger read/write to the graph data going through storage devices.

Orthus follows the hierarchy with traditional caching, and it can only reap the cumulative bandwidth of both PM and NVMe when the hit rate of PM is high. So, when the hit rate of PM is low, it needs to migrate data from NVMe to PM, thereby incurring concurrent PM write. SPFS, in the shortage of DRAM capacity for buffering data from NVMe, cannot proactively migrate hot data from NVMe to PM because SPFS can only migrate data from NVMe to PM when a write access happens [62]. Hence, its read performance is limited to using NVMe though with less data migration overheads compared with Orthus. In contrast, PolyStore can dynamically monitor whether the PM read and write bandwidths are saturated or not; to be specific, when the PM write bandwidth is saturated, PolyStore does not migrate data from NVMe to PM, and performs a direct NVMe read. Therefore, PolyStore incurs 2.46x less PM writes than Orthus, and presenting 1.84x and 1.62x shorter time to load the graph and intermediate walk info from disk compared to Orthus and SPFS respectively.

6 Conclusion

To conclude, we propose PolyStore, a horizontally arranged heterogeneous storage design to exploit the cumulative bandwidth of multiple storage devices without throwing away the virtues of mature and well-tested hardware-optimized file systems. PolyStore's techniques of scalable data indexing, dynamic thread and data placement, storage heterogeneity-aware DRAM caching, and durability contribute to high-performance gains, improving performance by up to 9.38x against state-of-the-art caching and tiering systems.

Acknowledgements

We thank our shepherd, Yu Liang, and anonymous reviewers for the insightful comments to improve the quality of this paper. We also thank Changwoo Min and the members of RSRL for their valuable feedback. This research was supported by funding from NSF CNS grants 1910593, 2231724, 2443438, and Samsung Memory Solutions Lab. This work was carried out on experimental platforms funded by the NSF II-EN grant 1730043.

References

[1] bcache. https://bcache.evilpiepirate.org/.

[2] e2fsck: fsck for ext4. https://linux.die.net/man/8/e2fsck.

[3] Friendster social network and ground-truth communities. https://snap.stanford.edu/data/com-Friendster.html.

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

[5] Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.

[6] Intel Optane Persistent Memory Product Brief. https://intel.ly/3c1IGH0.

[7] Intel Optane RAID. https://www.intel.com/content/www/us/en/developer/articles/technical/storage-redundancy-with-intel-optane-persistent-memory-modules.html.

[8] POSIX O_APPEND. https://man7.org/linux/man-pages/man2/open.2.html.

[9] RBtree based Interval Tree as a Priority Tree Replacement. https://lwn.net/Articles/509994/.

[10] Redis. http://redis.io/.

[11] RocksDB. http://rocksdb.org/.

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

[13] Saba Ahmadian, Reza Salkhordeh, and Hossein Asadi. Lbica: A load balancer for i/o cache architectures. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1196–1201. IEEE, 2019.

[14] Jens Axboe. FIO. https://fio.readthedocs.io/en/latest/fio$_d$oc.html.

[15] Nathan C Burnett, John Bent, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Exploiting gray-box knowledge of buffer-cache management. In *USENIX Annual Technical Conference, General Track*, pages 29–44, 2002.

[16] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[17] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in multics. *Commun. ACM*, 11(5):306–312, may 1968.

[18] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[19] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, Boston, MA, February 2019. USENIX Association.

[20] Brian C Forney and Andrea C Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *Conference on File and Storage Technologies (FAST 02)*, 2002.

[21] Shaleen Garg, Jian Zhang, Rekha Pitchumani, Manish Parashar, Bing Xie, and Sudarsun Kannan. Crossprefetch: Accelerating i/o prefetching for modern storage. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 102–116, New York, NY, USA, 2024. Association for Computing Machinery.

[22] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 20, USA, 2011. USENIX Association.

[23] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[24] David A Holland, Elaine Angelino, Gideon Wald, and Margo I Seltzer. Flash caching on the storage client. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 127–138, 2013.

[25] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-Aware Logging in Transaction Systems. *Proc. VLDB Endow.*, 8(4):389–400, December 2014.

[26] Intel. Intel nvme ssd 750 data sheet. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-750-spec.pdf.

[27] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. Dulo: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, volume 4, pages 8–8, 2005.

[28] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-mt: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, page 24–35, New York, NY, USA, 2009. Association for Computing Machinery.

[29] Crobett Jonathan. Linux Swapping. https://lwn.net/Articles/495543/.

[30] Bryant A. Julstrom. Greedy, genetic, and greedy genetic algorithms for the quadratic knapsack problem. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, page 607–614, New York, NY, USA, 2005. Association for Computing Machinery.

[31] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

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

[33] Jaehyung Kim, Hongchan Roh, and Sanghyun Park. Selective i/o bypass and load balancing method for write-through ssd caching in big data analytics. *IEEE Transactions on Computers*, 67(4):589–595, 2017.

[34] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 147–161, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 45–58, 2013.

[36] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write Policies for Host-side Flash Caches. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 45–58, San Jose, CA, February 2013. USENIX Association.

[37] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. Centaur: Host-side ssd caching for storage performance control. In *2015 IEEE International Conference on Autonomic Computing*, pages 51–60. IEEE, 2015.

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

[39] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, Hollywood, CA, USA, 2012.

[40] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write Optimization of Log-structured Flash File System for Parallel I/O on Manycore Servers. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 21–32, New York, NY, USA, 2019. ACM.

[41] 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, Santa Clara, CA, 2015.

[42] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. HiLSM: An LSM-Based Key-Value Store for Hybrid NVM-SSD Storage Systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, CF '20, pages 208–216, New York, NY, USA, 2020. Association for Computing Machinery.

[43] Zhen Lin, Lingfeng Xiang, Jia Rao, and Hui Lu. P2CACHE: Exploring tiered memory for In-Kernel file systems caching. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 801–815, Boston, MA, July 2023. USENIX Association.

[44] Iyswarya Narayanan, Aishwarya Ganesan, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Anand Sivasubramaniam. Getting More Performance with Polymorphism from Emerging Memory Technologies. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 8–20, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, volume 12, 2012.

[46] 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, pages 109–116, New York, NY, USA, 1988. Association for Computing Machinery.

[47] PCISIG. PCI Express 6.0 Specification. https://pcisig.com/pci-express-6.0-specification.

[48] Ulrich Pferschy and Joachim Schauer. The knapsack problem with conflict graphs. *J. Graph Algorithms Appl.*, 13(2):233–249, 2009.

[49] Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD lifetimes with Disk-Based write caches. In *8th USENIX Conference on File and Storage Technologies (FAST 10)*, San Jose, CA, February 2010. USENIX Association.

[50] Yujie Ren, Changwoo Min, and Sudarsun Kannan. CrossFS: A Cross-layered Direct-Access File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154. USENIX Association, November 2020.

[51] Andrea Righi. Per-task I/O throttling. https://lwn.net/Articles/264770/.

[52] A.H.G. Rinnooy Kan, L. Stougie, and C. Vercellis. A class of generalized greedy algorithms for the multi-knapsack problem. *Discrete Applied Mathematics*, 42(2):279–290, 1993.

[53] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, jul 1974.

[54] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[55] Samsung. Samsung sata ssd 870 evo pro data sheet. https://semiconductor.samsung.com/us/consumer-storage/internal-ssd/870evo.

[56] Woong Shin, Christopher D. Brumgard, Bing Xie, Sudharshan S. Vazhkudai, Devarshi Ghoshal, Sarp Oral, and Lavanya Ramakrishnan. Data jockey: Automatic data management for hpc multi-tiered storage systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 511–522, 2019.

[57] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Prism: Optimizing key-value store for modern heterogeneous storage devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, pages 588–602, New York, NY, USA, 2023. Association for Computing Machinery.

[58] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1541–1555, New York, NY, USA, 2018. Association for Computing Machinery.

[59] Tarasov Vasily. Filebench. https://github.com/filebench/filebench.

[60] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John C. S. Lui. GraphWalker: An I/O-Efficient and Resource-Friendly graph analytic system for fast and scalable random walks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 559–571. USENIX Association, July 2020.

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

[62] Hobin Woo, Daegyu Han, Seungjoon Ha, Sam H. Noh, and Beomseok Nam. On stacking a persistent memory file system on legacy file systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 281–296, Santa Clara, CA, February 2023. USENIX Association.

[63] NVM Express Workgroup. NVMExpress Specification. https://nvmexpress.org/resources/specifications/.

[64] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307–323. USENIX Association, February 2021.

[65] Xiaojian Wu and AL Narasimha Reddy. A novel approach to manage a hybrid storage system. *J. Commun.*, 7(7):473–483, 2012.

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

[67] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[68] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. Overcoming the memory wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 601–617, Boston, MA, July 2023. USENIX Association.

[69] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31. USENIX Association, July 2020.

[70] Jian Zhang, Tao Xie, Yuzhuo Jing, Yanjie Song, Guanzhou Hu, Si Chen, and Shu Yin. Bora: a bag optimizer for robotic analysis. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[71] 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, February 2019. USENIX Association.

[72] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. Enabling high-performance and secure userspace nvm file systems with the trio architecture. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 150–165, New York, NY, USA, 2023. Association for Computing Machinery.

[73] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM performance with opportunistic delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, Carlsbad, CA, July 2022. USENIX Association.

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

[75] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. SOSP '19, page 259–274, New York, NY, USA, 2019. Association for Computing Machinery.

A Artifact Appendix

Abstract

PolyStore artifact is the practical implementation of the whole system design presented in this paper aimed at capitalizing the cumulative storage bandwidth of HSDs with their mature and hardware-optimized OS-level file systems with different hardware configurations. To use our artifact and reproduce the main experiments, we provide instructions to build the PolyStore runtime and PolyOS kernel module, and run experiments Appendix A. Though the scripts to run the experiments in the *Testbed I* (Table 2), readers can change them for other configurations of HSDs on other hardware platforms.

Scope

The artifact of PolyStore aims to demonstrate the methodologies to utilize the cumulative storage bandwidth of HSDs proposed, designed, and implemented in this paper. This artifact is intended not only to validate the main claims in this paper but also to enable researchers and open-source developers to extend and explore new ideas, methodologies, and practices of managing HSDs. The artifact of PolyStore is licensed under *Apache License 2.0*, and the copyright is held by Rutgers University. There is no warranty and merchantability for commercial purposes.

Contents

The PolyStore artifact comprises user-level PolyStore runtime, the PolyOS kernel module, and the necessary *bash* and *python* scripts to set up the environment, systems, and experiments. Here is the complete list:

- Source code of PolyStore runtime and PolyOS kernel module.
- Source code of necessary third-party libraries.
- Source code of benchmark and application workloads.
- Scripts for setting up environment and compilation.
- Scripts for running experiments.

Hosting

The artifact of PolyStore is hosted on Github with the *README* file for documentation in the following public link: https://github.com/RutgersCSSystems/PolyStore.

Requirements

The artifact of PolyStore is based on *Linux kernel 5.1.0* with the NOVA file system. The hardware platform is specified in Table 2 and §5. The current scripts for setting up the environment and desired libraries are developed for *Ubuntu 20.04.5 LTS* and the specific storage devices on the testbed in evaluating PolyStore. Porting to other Linux distributions and hardware platforms would require some script modifications.

Evaluation

We provide comprehensive step-by-step instructions on GitHub to reproduce the major experiments in the paper. Using different hardware platforms and storage devices may output similar or divergent results observed from the paper. Hence, we welcome researchers and open-source developers to contact us on our Github repository for any issues and suggestions. As a brief overview of the evaluation, we illustrate how to execute the "Hello world" example with PolyStore.

Before starting, we assume the current work directory is in the root directory of the Github repository of PolyStore.

Build and install the desired Linux kernel

```
$ cd $BASE
$ source ./scripts/setvars.sh
$ ./scripts/compile_kernel.sh
$ sudo reboot
```

Mount the file systems

After reboot, go to the work directory is in the root directory of the Github repository of PolyStore.

```
$ cd $BASE
$ ./scripts/mount_pmem_nova.sh
$ ./scripts/mount_nvme_ext4.sh
```

Please modify the above two scripts with the actual devices in the target testbed.

Compile and build PolyStore runtime and PolyOS kernel module

```
$ source ./scripts/setvars.sh
$ cd $BASE/polylib
$ make clean && make
$ cd $BASE/polylib/src/polyos
$ make clean && make
$ cd $BASE/tools
$ make clean && make
```

Compile and build the microbenchmark

```
$ cd $BASE/benchmarks/microbench
$ make clean && make
```

Run a quick test with the microbenchmark

```
$ cd $BASE/experiments/microbench
$ ./run_polystore_dynamic.sh
```

Expect output will be similar to "aggregated thruput 7072.90 MB/s, average latency 32.08 us".

Acknowledgments

We appreciate the valuable time and patience of the anonymous artifact evaluation reviewers and their comments and suggestions to improve the accessibility and quality of our artifact. We thank again to Rutgers Panic Lab for the infrastructure support during the artifact evaluation process. We also thank the software's developers and contributors of the tools and libraries used in the artifact of PolyStore.