USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Getting the MOST out of your Storage Hierarchy with Mirror-Optimized Storage Tiering

Kaiwei Tu, *University of Wisconsin–Madison;* Kan Wu, *Google;*
Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau,
University of Wisconsin–Madison

https://www.usenix.org/conference/fast26/presentation/tu

This paper is included in the Proceedings of the
24th USENIX Conference on File and Storage Technologies.

February 24–26, 2026 • Santa Clara, CA, USA

ISBN 978-1-939133-53-3

Open access to the Proceedings of the
24th USENIX Conference on File and Storage Technologies
is sponsored by

Getting the MOST out of your Storage Hierarchy with Mirror-Optimized Storage Tiering

Kaiwei Tu, Kan Wu[†], Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin–Madison, [†]Google

Abstract

We present *Mirror-Optimized Storage Tiering* (MOST), a novel tiering-based approach optimized for modern storage hierarchies. The key idea of MOST is to combine the load-balancing advantages of mirroring with the space-efficiency advantages of tiering. Specifically, MOST dynamically mirrors a small amount of hot data across storage tiers to efficiently balance load, avoiding costly migrations. As a result, MOST is as space-efficient as classic tiering while achieving better bandwidth utilization under I/O-intensive workloads. We implement MOST in Cerberus, a user-level storage management layer based on CacheLib. We show the efficacy of Cerberus through a comprehensive empirical study: across a range of static and dynamic workloads, Cerberus achieves better throughput than competing approaches on modern storage hierarchies especially under I/O-intensive and dynamic workloads.

1 Introduction

Storage hierarchies have been essential to computer system design since its inception [20]. In persistent storage systems, such hierarchies are commonplace; for example, SSDs are commonly used as a caching layer above hard drives [47]; AutoRAID uses a mirrored set of disks for performance on top of a log-structured RAID-5 layout for capacity [66].

However, as recent work has shown [32, 69], new device types pose challenges in the construction of storage hierarchies. Specifically, new non-volatile memories [7] and low-latency SSDs [8, 9] have performance and capacity profiles that overlap. As such, arranging devices into a strict hierarchy is difficult [32, 69]; doing so ensures that the peak potential of the storage system is not realized.

We are thus left with a significant challenge: how should we manage a collection of modern devices so as to maximize performance while minimizing space overheads? As we discuss in this paper, classic techniques fall short.

To understand the issues better, consider a simplified two-device system, with a faster/smaller "performance" device and a slower/larger "capacity" device. The first family of approaches we call "single copy", as they keep only a single copy of each data block in storage. One well-known single-copy approach is tiering [16, 25, 30, 38, 39, 44, 54, 56, 60, 61, 64, 67, 73, 75, 79, 80], which (slowly) migrates blocks between tiers based on the hotness or some other optimization goal. Tiering is space-efficient, as there is one copy of each block; however, a tiering-based system cannot efficiently adapt to changes in load because data need to be migrated around the tiers. Striping [22, 59] is another single-copy approach that statically allocates blocks across devices but struggles to manage heterogeneous hierarchies or changes in workload [24].

The second family of approaches we call "multiple copy", as they replicate some (or all) blocks. A well-known multiple-copy approach is caching [6, 18, 27, 46, 48, 49, 53, 69, 74], which makes copies of popular blocks in the performance device. As a result, caching wastes capacity (as all copy-based approaches do); in addition, caching does not utilize the bandwidth of the capacity layer, thus falling short of peak performance. Mirroring [19, 22], a technique that replicates data across two devices, effectively balances the load for read requests by directing them to either device, yet also incurs a significant capacity cost.

The state-of-the-art approaches for handling heterogeneous memory and storage devices focus on tiering [23, 64]; at a high level, these previous works attempt to maximize the total throughput delivered by all devices by dynamically placing and migrating data across devices such that accesses to each device are proportional to their bandwidth or equalize access latency. In this paper, we introduce *Mirror-Optimized Storage Tiering* (MOST) and show that mirroring a small amount of hot data across devices, in combination with tiering, greatly improves performance and reduces device writes. Given these mirrored data, the host can dynamically route requests to different devices to effectively balance load instead of performing costly data migrations across devices like other approaches.

We find that adding a small amount of mirroring is advantageous to pure tiering for three primary reasons. First, mirrors enable the system to quickly react to workload changes by adjusting routing; pure tiering, on the other hand, must slowly migrate a significant amount of data to adjust load distribution. Second, mirrors reduce overall device writes when handling dynamic workloads; migration-based approaches require extensive device writes to move data across tiers. Third, mirroring is more robust to fluctuations in device performance and prevents overreacting with unnecessary migrations. This effect is particularly pronounced for storage devices that suffer from poor tail latency or write-heavy workloads that frequently trigger background activity inside the SSD.

MOST is a novel approach optimized for the modern storage hierarchy. MOST mirrors a small amount of hot data within the classic tiering system, enabling flexible load dis-

tribution control. Under low load, MOST functions similarly to a classic tiering system, routing requests for mirrored data to the performance device to maximize its utilization. Under high load, MOST dynamically routes a portion of requests to the capacity copy of the mirrored data, thereby utilizing the bandwidth of the capacity device more effectively. Key to the success of MOST are the algorithms that manage its behavior. MOST is based on a simple, robust, and self-adjusting optimization mechanism that requires no prior workload knowledge and is capable of handling the varied performance characteristics of different storage hierarchies.

We implement MOST as a storage management layer named Cerberus within the widely-used flash cache library, CacheLib [2, 18, 46]. Our experiments are conducted on two storage hierarchies using three different real device types [5, 8, 12–14]. For static cache workloads, MOST achieves up to 2.34× higher throughput and reduces P99 latency by up to 75% compared to state-of-the-art systems (Colloid [64], HeMem [56], and Orthus [69]). Across four production cache workloads, MOST increases throughput by up to 1.86× and reduces P99 GET latency by up to 90% compared to these systems. Overall, Cerberus enhances throughput under intensive workloads and reduces writes by up to 84% compared to Colloid under dynamic workloads through its judicious use of mirroring across storage tiers.

2 Motivation and Background

In this section, we discuss some of the trends in modern memory and storage hierarchies containing multiple heterogeneous devices. We then describe existing management techniques – such as striping, tiering, mirroring, and caching– and analyze their limitations. We conclude by discussing some of the extra challenges of managing storage hierarchies, as opposed to memory hierarchies.

2.1 Multi-Device Hierarchies

Multiple devices [23, 39, 56, 58, 64, 69, 79] are commonly used to serve memory and storage load. Since new devices may be added incrementally over time to increase capacity or performance, these new devices are likely to exhibit different performance characteristics compared to older devices in the system. One natural approach has been to organize memory or storage devices into a *hierarchy* of tiers, with newer, faster devices placed as a cache above slower, possibly larger devices; such hierarchies have been essential to systems since their inception [20].

However, as recent work has shown [32, 69], new device types pose challenges to ordered hierarchies. To simplify our discussion, we focus on a two-layered hierarchy containing two types of devices: a *performance* device and a *capacity* device, where the performance device is more expensive, smaller in capacity, and faster, whereas the capacity device is cheaper, larger, and slower. In a traditional hierarchy (e.g., DRAM on HDD), the performance gap between the two devices is substantial and the performance of the capacity device can be ignored. However, with emerging technologies such as non-volatile memory [7], low-latency SSDs [8], NVMe Flash SSDs [5, 14], SATA Flash SSDs [13], and remote storage with NVMeoF [12], disaggregated SSDs [35, 36], and EBS [1, 78], the performance and capacity devices have overlapping characteristics. As such, arranging these devices into a strict hierarchy [32, 69] limits their potential.

Storage Device	Latency (µs)		Read (GB/s)		Write (GB/s)	
	4K	16K	4K	16K	4K	16K
Optane SSD	11	18	2.2	2.4	2.2	2.2
PCIe 4.0 NVMe Flash SSD	66	86	1.5	3.3	1.9	2.3
PCIe 3.0 NVMe Flash SSD	82	90	1.0	1.6	1.5	1.6
PCIe 4.0 NVMe Flash SSD over RDMA	88	114	1.2	2.7	1.7	2.3
SATA Flash SSD	104	146	0.38	0.5	0.38	0.5

Table 1: **Device Performance.** *Latency measured for a single-thread read workload; bandwidth for a 32-thread workload. Remote device is connected via a 25 Gbps link.*

Table 1 shows that latency and bandwidth can be similar for the performance and capacity devices given current technology. Memory and storage systems can be composed of any pair of these devices. Importantly, the performance ratio across the two tiers may be close: the bandwidth ratio for 16KB reads is only 1.5:1 between Optane and PCIe 3.0 NVMe devices and only 1.25:1 between local and remote PCIe 4.0 NVMe devices. More subtly, the performance ratios are not static and strongly depend on workload; for example, given 4KB reads, the ratio between Optane and PCIe 3.0 NVMe devices increases to approximately 2.2:1. These performance ratios depend on access sizes, percentage of writes, and concurrency [68].

2.2 Existing Approaches

At a high level, classic approaches for managing multi-device memory and storage can be grouped into two categories: approaches that maintain a single copy of each block (i.e., striping, tiering, and exclusive caching), and approaches that may maintain multiple copies of each block (i.e., mirroring, caching). We describe these approaches and highlight their inefficiencies, as summarized in Table 2.

Single Copy. *Striping* [22, 24, 28, 51, 52, 59] allocates a single copy of each block in a predetermined static pattern across devices. Striping does not handle heterogeneous devices well: if data is striped evenly across devices, throughput is bottlenecked by the slowest device; if data is striped in a weighted pattern with more data on faster devices, the appropriate weight depends on the workload; furthermore, the ideal striping ratio for performance is unlikely to match the ratio of the devices' capacity (i.e., faster devices are usually smaller). Thus, striping is not a good match for heterogeneous storage.

Tiering [16, 25, 38, 44, 54, 60, 61, 67, 73, 79, 80] also allocates a single copy of each block; however, instead of allocating blocks in a static pattern, tiering dynamically places data blocks based on their hotness [16, 38, 43, 54, 61, 73, 79, 80] or some other optimization goal [23, 30, 75]. Table 2 summarizes the characteristics of three tiering approaches.

First, classic hotness-based tiering is exemplified by

		Single Copy				Multiple Copy		
		Striping	HeMem	BATMAN	Colloid	Mirroring	Orthus	MOST
Bandwidth Utilization	Random Read-only	Low ●	Low ●	Medium ●●	Medium ●●	High ●●●	High ●●●	High ●●●
	Random Write-only	Low ●	Low ●	Medium ●●	Low ●	Low ●	Low ●	High ●●●
	Random RW-mixed	Low ●	Low ●	Medium ●●	Medium ●●	Medium ●●	Medium ●●	High ●●●
	Sequential Write	Low ●	Low ●	Low ●	Low ●	Low ●	Low ●	High ●●●
Capacity Utilization		High ●●●	High ●●●	High ●●●	High ●●●	Low ●	Low ●	High ●●●
Dynamic Workload		Low ●	Low ●	Low ●	Low ●	Medium ●●	Medium ●●	High ●●●

Table 2: **Qualitative Comparison of Different Techniques in a Modern Storage Hierarchy.** *Bandwidth utilization is categorized as follows: Low means no load-balancing mechanism; Medium represents limited load-balancing with restrictions under specific workloads; High indicates effective load-balancing across a wide range of workloads.*

HeMem [56], which manages a memory system containing DRAM and NVM. HeMem promotes hot data into the performance tier and serves hot data exclusively from the performance tier, leading to inefficient utilization of the capacity tier's bandwidth; thus, HeMem delivers low bandwidth for intensive workloads due to not effectively utilizing NVM (as shown later §4). Second, BATMAN [23] balances tier bandwidth by migrating data, but its fixed bandwidth ratio prevents it from adapting to hierarchies whose performance ratios vary with workload or evolve over time. Third, Colloid [64] also dynamically migrates data between memory tiers to utilize the bandwidth of the capacity device; however, Colloid does not handle dynamic and time-varying workloads because costly data migrations are required to adjust the load distribution when the workload changes (§4.2).

Other tiering systems, such as AutoTiering [75] and EDT-DTM [30], also balance data placement to utilize the bandwidth of all tiers, but since they rely solely on data migration to adjust load distribution, their adaptability and responsiveness to workload changes is limited. *Exclusive caching* [29] maintains only a single copy of data within the hierarchy: when hot data is promoted or demoted between tiers, the original copy is discarded. At a high level, exclusive caching is similar to hotness-based tiering but moves data at smaller time intervals; consequently, it behaves similarly. In summary, as shown in Table 2, single-copy approaches may partially utilize the bandwidth of the capacity device by migrating data across tiers. However, this migration-based strategy struggles to adapt to dynamic workloads, as migration is both costly and slow. In addition to limited responsiveness, migration-based approaches also suffer from excessive device write and performance degradation due to migration-induced traffic (see §2.3 for further discussion).

Multiple Copies. To utilize both devices, other approaches maintain two copies of at least some data blocks across both the performance and capacity devices. *Mirroring* [19,22] simply replicates all data on both devices. Mirroring delivers high read bandwidth since reads can be dynamically load balanced across both devices to account for performance differences; however, mirroring delivers low write bandwidth since both copies must be updated and bandwidth is limited by the slower device. Mirroring also has low capacity utilization since each block is stored on both devices.

Inclusive caching allocates all items on the capacity device, but replicates only frequently-accessed items on the performance device [6,18,27,46,48,49,53,69,74]; inclusive caching inherently wastes the capacity of the performance device and fails to exploit the performance of the capacity device [17,34]. To utilize the performance of the capacity device, Orthus [69] introduces Non-Hierarchical Caching (NHC). Orthus dynamically redirects read traffic from the performance device to the capacity device when the performance device is overloaded. However, Orthus is built on a traditional caching model, which introduces two fundamental limitations. First, it is space-inefficient—NHC uses the entire capacity of the performance device to store duplicate copies of data from the capacity tier, wasting the performance tier's storage capacity. Second, Orthus struggles with write-intensive workloads, requiring clean copies for routing read requests: writes that hit in the cache can be handled with write-through or write-back/write-around. With write-through, both copies remain clean, but additional writes are incurred, and performance is constrained by the write bandwidth of the capacity device. With write-back/write-around to only one of the block's copies, Orthus can only route subsequent reads to the dirty block. Thus, Orthus performs poorly with writes (§4.1).

Nomad [72] proposes a variant to hotness-based tiering that maintains temporary copies of data during migration; specifically, the original copy on the capacity device can still be accessed while the data is being migrated to the performance device. While Nomad improves the performance penalty of migration, it does not maximize the bandwidth of the underlying devices in the common case.

Summary. Existing approaches for handling heterogeneous memory and storage fail along important metrics (Table 2). Approaches that maintain a single copy of data, such as striping, tiering (e.g., HeMem, BATMAN, Colloid, AutoTiering, and EDT-DTM), and exclusive caching, fall short in handling dynamic workloads. Current approaches, such as mirroring, inclusive caching, and non-hierarchical caching, that maintain multiple copies of data items suffer from low capacity utilization and struggle with write-intensive workloads. We will show that MOST delivers high bandwidth and reduces device writes across a wide range of static and dynamic workloads with only a small amount of mirrored data.

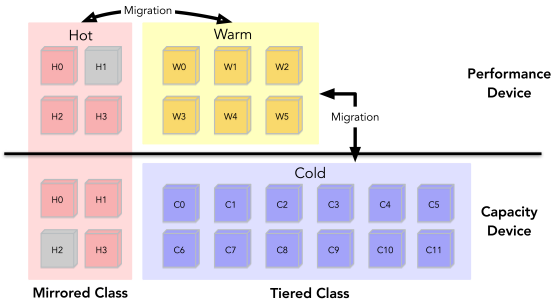


Figure 1: **MOST Data Layout.** *Data are logically grouped but not physically placed together on the device.*

2.3 Storage vs. Memory Hierarchies

The approaches that handle data placement in memory hierarchies [23, 39, 56, 64] versus storage hierarchies [66, 75, 79] have many similarities. In both the memory and the storage domain, strategies for data placement optimize for many of the same metrics, such as high performance (bandwidth and latency) and high capacity. In both domains, solutions share many of the same high-level approaches (e.g., striping, tiering, caching), as well as similar low-level techniques (e.g., attempting to balance load across heterogeneous devices by reacting to observed performance whether for memory [23, 64] or for storage [69]).

However, storage hierarchies have characteristics that require special attention and introduce additional opportunities.

Larger Capacity and Datasets: Storage devices manage significantly larger datasets than memory. Thus, if migration is used as the primary technique for balancing load across tiers, more data will need to be migrated leading to larger convergence times when adjusting to workload changes.

Limited Write Bandwidth: Storage devices have lower write bandwidth than memory; combined with the previous point, this further increases convergence time if migration is used to adjust to dynamic workloads.

Read/Write Interference: Some storage devices [68, 70] have complex performance characteristics such that write operations significantly impact the performance of read operations. Migrating data in the background introduces write traffic that can significantly degrade foreground performance.

Device Endurance: Frequent migrations accelerate device wear, reducing the lifespan of storage devices such as SSDs [45, 46].

Software-based Indirection: Since storage devices have slower access time (10 - 500 μs) than memory (50 - 100 ns), software is used to determine the location of blocks on different devices instead of hardware. As a result, techniques such as dynamic and selective mirroring are more straightforward to implement for the storage stack.

Thus, while dynamic tiering with data migration has been shown to work well for heterogeneous memory hierarchies [56, 64], heterogeneous storage hierarchies need additional techniques that do not rely as heavily on migration.

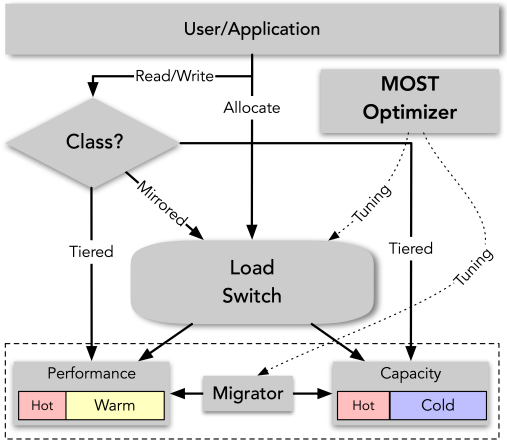


Figure 2: **MOST Architecture.**

3 Mirror-Optimized Storage Tiering

We present *Mirror-Optimized Storage Tiering* (MOST), a combination of mirroring and tiering that enables flexible load balancing across tiers by mirroring a small amount of data. MOST is based on three design goals:

Maximized Bandwidth Utilization. MOST should dynamically balance load across all storage tiers based on workload.

Quick Response to Dynamic Workloads. MOST should dynamically and quickly adapt to varying workloads, including fluctuations in load and working set size.

Independence from Device and Workload Characteristics. MOST should operate effectively across any storage hierarchy without requiring prior knowledge of device characteristics or workload patterns.

3.1 Basic Design

The key idea of MOST is to combine the load-balancing advantages of mirroring with the capacity benefits of classic tiering. MOST uses a hybrid data layout with two classes: the mirrored class and the tiered class (Figure 1). Data in the mirrored class is replicated across two devices to enable fast load balancing; data in the tiered class is stored as a single copy to maximize the space efficiency. In MOST, *hot* data, which is accessed most frequently, is moved to the mirrored class and the remaining data is stored in the tiered class. Within the tiered class, *warm* data is migrated to the performance device; the remaining *cold* data is migrated to the capacity device.

As desired, the performance device stores the most-accessed data: *hot* data in the mirrored class and *warm* data in the tiered class. Under light workloads, this layout maximizes the likelihood that a request can be served from the performance device. Under intense workloads, MOST dynamically increases the size of the mirrored class and routes some requests in the mirrored class to the capacity copy to balance the load across both devices.

3.2 Design Details

Figure 2 shows the basic architecture of MOST. Blocks are placed in the mirrored or tiered class depending on their fre-

Algorithm 1 MOST Optimizer

```
1:  while true do
2:      sleep(tuningInterval); Measure end-to-end latency.
3:      if L_P > (1 + θ) * L_C
4:          if offloadRatio = offloadRatioMax
5:              if Mirrored class is not maximized
6:                  Enlarge the mirrored class
7:              else
8:                  Improve hotness of the mirrored class
9:              only migrate to capacity device
10:         else offloadRatio += ratioStep
11:     else if L_P < (1 - θ) * L_C
12:         if offloadRatio = 0
13:             only migrate to performance device
14:         else offloadRatio -= ratioStep
15:     else stop all migration
```

quency of access: the goal is to place the hottest data in the mirrored class, warm data in the tiered class on the performance device, and cold data in the tiered class on the capacity device. When handling a request, MOST first determines if the request is to the tiered or mirrored class. Requests to tiered data are straightforward since the data resides strictly in either the performance or capacity device; those reads and writes are simply forwarded to the appropriate location. Requests to mirrored data involve more complexity: a load switch (or balancer) sends some percentage of these requests to each device, where the percentage depends upon the current workload performance and whether the request is a read or write. Other components of the architecture calculate this percentage (i.e., Optimizer) and migrate data between the classes and devices (i.e., Migrator).

We address key questions of MOST's design. For reads to the hottest data, which data copy in the mirrored class should be read to balance traffic across devices? On which device should newly written data be allocated? What data should be migrated between mirrored and tiered classes, and between devices? For writes to mirrored data, which copy should be updated? At what granularity should clean copies be tracked? When should copies be cleaned? How to protect the tail latency of mirrored data?

3.2.1 Balancing Read Traffic in the Mirrored Class

For simplicity, we begin by focusing on reads that are to the hottest data and thus in the mirrored class. We later show how MOST allocates data, moves data to and from the mirrored class, and handles writes. The full algorithm is shown above.

At a high level, reads to mirrored data should be routed to the device that delivers the best performance. To achieve this, MOST employs probability-based routing to determine which copy to read [69]. Specifically, MOST routes an incoming read with probability *offloadRatio* to the capacity device, and otherwise, to the performance device. *OffloadRatio* is the key variable controlling load distribution, as well as write traffic and allocation. *OffloadRatio* is tuned by a simple yet efficient feedback-driven dynamic optimization algorithm within the Optimizer; *offloadRatio* is adjusted such that the measured end-to-end latency of the devices is equalized.

Specifically, assuming L_P is the latency of the performance device and L_C is the latency of the capacity device, then

- When $L_P < L_C$, accessing the performance device is faster than accessing the capacity device, and so more traffic should be routed to the performance device (see Lines 11–14); thus, *offloadRatio* is decreased.
- When $L_P > L_C$, accessing the capacity device is faster than accessing the performance device, and so more traffic should be routed to the capacity device (see Lines 3–10); thus, *offloadRatio* is increased.
- When $L_P = L_C$, the two access latencies are approximately equal, and therefore, no further action is required (Line 15).

MOST efficiently adapts to both high-load and low-load scenarios. Under low load, the latency of accessing the performance device is lower than that of the capacity device, and so traffic is routed to the performance device; in these conditions, MOST operates like classic tiering, directing as many requests as possible to the performance device. In contrast, under high load, all hot and warm data reside on the performance device, causing its access latency to increase—due to queuing and internal resource contention—and eventually exceed that of the capacity device; consequently, MOST offloads requests to the capacity device which improves throughput. Under high load, MOST offloads traffic from the performance device to the capacity device until the end-to-end latency between the two devices is equalized.

3.2.2 Dynamic Write Allocation

When a data block is first written, MOST allocates its location. MOST divides both mirrored and tiered storage into fixed-sized segments (e.g., 2MB) and maintains in-memory data structures to track the metadata for the segment. All data are initially allocated into the tiering class. In classic tiering, the allocation policy is load-unaware, meaning that it always allocates newly-created blocks on the performance device, even when that device is saturated. MOST introduces a probability-based write allocation policy: similar to load balancing within the mirrored class, the newly allocated data is placed on the capacity device with *offloadRatio* probability. As a result, when the performance device is under high load (higher latency), more data is allocated on the capacity device; when the performance device is under light load (lower latency), all data is allocated on the performance device, as desired and as in classic tiering [56, 64]. After a block has been allocated to the tiered class, it may later be promoted to the mirrored class, as described next.

3.2.3 Mirror-Class Migration

To meet the basic goal of placing the hottest data in the mirrored class, MOST migrates data between classes such that it minimizes the amount of movement between devices. To identify hot/warm/cold data, MOST tracks read and write counters for each segment, similar to HeMem [56]; thus, hotness is based on access frequency.

When the amount of mirrored data is insufficient to effectively balance traffic (Line 4), MOST increases the size of the mirrored class up to a configured maximum. Our experiments (§4) show that devoting 20% of the total capacity to the mirrored class is sufficient for our workloads. To migrate data into the mirrored class, MOST selects the hottest segment from the tiered class on the performance device; this hottest data is simply duplicated onto the mirrored class on the capacity device. If the mirrored class reaches its maximum size and the hottest segment in the tiered class has a higher access frequency than the coldest segment in the mirrored class, MOST swaps those segments. Space within the mirrored class is reclaimed when the available system capacity drops below a predefined watermark (2.5% of the total capacity). During reclamation, MOST reclaims the coldest segment in the mirrored class: if a valid copy of this segment exists on the performance device, the capacity copy is discarded; otherwise, the copy on the performance device is discarded.

Migration Regulation. In classic tiering, hot data is always migrated to the performance device and cold data to the capacity device; however, with MOST and its mirrored class, hot segments may be migrated to the capacity device. MOST dynamically manages this bidirectional migration, adhering to a principle of migrating exclusively *away* from the device experiencing higher end-to-end latency. Precisely, when the performance device has higher latency, migration to the performance device is stopped and migration to the capacity device is enabled; when the capacity device exhibits higher latency, the opposite is performed; if the latencies of both devices are approximately equal, all migration is stopped.

3.2.4 Balancing Writes and Tracking Clean Subpages

Our previous discussion focused on reads; writes need special consideration. Writes to the tiered class remain straightforward since there is only one copy to update. However, writes to the mirrored class have a new property: if the write is performed to both copies, then the system is not performing any write load-balancing; however, if the write updates only one copy, then future reads must be directed only to that clean, valid copy. In order to perform load-balancing of writes, MOST updates only one copy and carefully tracks which portions of each segment are valid.

Writes to the mirrored class are balanced as follows. If both copies of the mirrored data are valid, the write request is probabilistically sent to either the capacity or performance device based on *offloadRatio*. However, if only one copy in the mirror is valid, a naive implementation based only on segments must direct later write traffic only to the valid segment (or overwrite the entire invalid segment, making it valid).

Mirrored Data Subpages. To enable better load balancing of writes, MOST manages segments in the mirrored class at a finer granularity: an invalid bit and a location bit are tracked for each subpage corresponding to the device's unit of access (e.g., 4KB). Each subpage exists in one of three states: clean (both copies are valid), invalid on the performance device, or invalid on the capacity device. Thus, given an aligned subpage write to the mirrored class, MOST can route the write to either device without having to update the entire segment; that is, a 4KB-aligned write can be load balanced through simple routing, similar to reads. Subpages in the mirrored class slightly increase metadata overhead (2 bits of metadata), but since the mirrored class is relatively small, the overall overhead remains minimal; for instance, in a 2TB hierarchy in the extreme case where all performance device data is mirrored (50% mirroring), the metadata overhead is only 128MB, which is negligible.

Selective Cleaning in the Mirror Class. Data blocks with only one valid copy are selectively cleaned by a background thread. This cleaning thread selects blocks with a large *rewrite distance*, the average number of reads between two writes for a given block. We have observed that when a block has a small *rewrite distance*, it is likely to be rewritten soon, making cleaning ineffectual.

3.2.5 Tail Latency Protection

The description thus far has focused on maximizing total bandwidth from the performance and capacity devices; however, MOST allows users to protect the tail latency of mirrored (hot) data by setting a maximum *offloadRatio* which limits the traffic offloaded to the capacity device in the mirrored class when the capacity device shows significantly worse tail latency than the performance device.

3.3 Implementation: Cerberus

We introduce Cerberus, a user-level storage management system that integrates MOST into CacheLib. CacheLib is a generic flash cache library used extensively in data center infrastructures [11, 18, 26, 46, 62, 63]. As shown in Figure 3, CacheLib consists of three layers: a DRAM cache layer, a flash cache layer, and a storage management layer. CacheLib provides two default flash cache layers: a Large Object Cache (LOC) employs a log with a DRAM index for items 2KB or larger; a Small Object Cache (SOC) stores key-value pairs in a 4KB bucket hash table. Users can also implement custom flash caches in this flash cache layer (e.g., Kangaroo [46] and Fairy-WREN [45]).

By default, the storage management layer in CacheLib only provides striping across devices. Cerberus is our storage management layer between the flash cache engine and the storage hierarchy employing MOST; Cerberus transparently manages the underlying performance and capacity devices, providing a block interface and a large address space. In

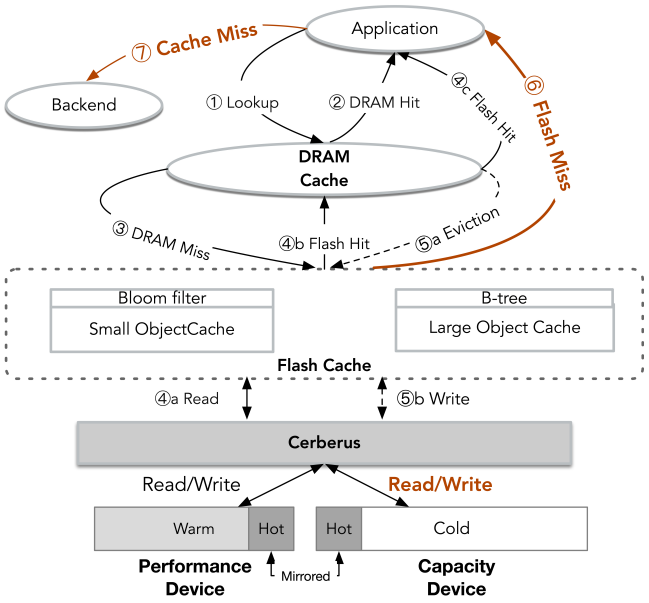


Figure 3: **CacheLib Architecture.** This figure shows CacheLib's architecture and the lookup workflow. A lookup first checks the DRAM cache (①) and immediately returns the object on a hit (②). On a miss, it checks the flash cache (③), issuing a read to the underlying storage devices (④a); if the object is found, it returns the result (④b). A flash cache hit promotes the item to DRAM and may evict an existing DRAM entry (⑤a, ⑤b). A miss in the flash cache (⑥) leads to a backend access (⑦).

addition to MOST, we have implemented other tiering and caching approaches within the CacheLib storage management layer.

Striping. The default implementation in CacheLib.

Orthus. The non-hierarchical caching implementation described and provided by Wu et al. [69] in .6K lines of code (LOC).

HeMem. Our implementation of the classic tiering and migration algorithm [56] in about .7k LOC. The original HeMem uses a quantum of 10ms which is appropriate for memory latency, but not storage; we use 200ms, the smallest interval that accurately reacts to our storage device latency.

BATMAN. Our implementation of BATMAN heterogeneous tiering [23], requiring about .4K LOC.

Colloid. Our implementation of Colloid [64] is built on HeMem (with a 200ms quantum) in about .4K lines of code. Since Colloid only balances reads, we implement Colloid+ to incorporate write latency into decisions. Additionally, since Colloid is sensitive to parameters, we implement Colloid++ with $\theta = 0.2$ and $\alpha = 0.01$, which improves robustness given storage device performance fluctuations.

Metadata Management. MOST divides storage into 2MB segments, where each segment requires 76 bytes of metadata as shown in Table 3. Smaller segments would result in larger metadata overhead (e.g., 4KB segments require 512 times more metadata) and degrade performance since they obtain lower device bandwidth (Table 1). Conversely, larger segments would lead to inefficient utilization of the performance tier, as only a small fraction of subpages might be frequently accessed. We have found that 2MB segments balance the metadata footprint while minimizing space waste on the performance tier, matching the choice of many other systems [39, 56, 64].

Member Variable	Size (bytes)
id (uint64_t)	8
addr[2] (uint64_t[ ])	16
invalid (bitset<512>*)	8
location (bitset<512>*)	8
clock (uint64_t)	8
readCounter (uint8_t)	1
writeCounter (uint8_t)	1
rewriteReadCounter (uint64_t)	8
rewriteCounter (uint64_t)	8
flags (uint8_t)	1
storageClass (enum class)	1
mutex (SharedMutex)	8
Total	**76**

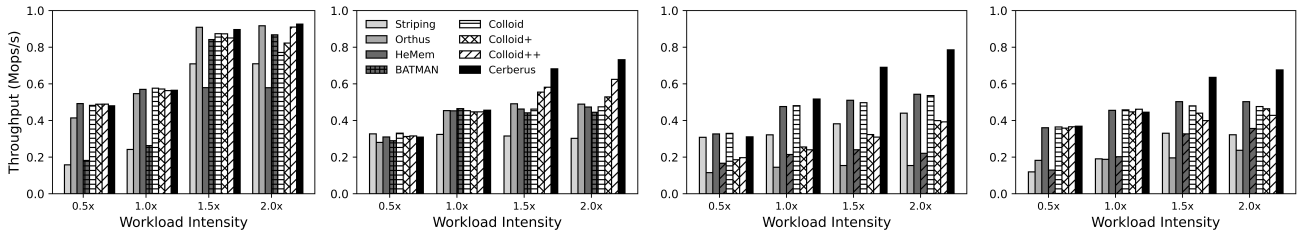
Table 3: **In-Memory Metadata per Segment.**

Implementation Details. The MOST optimizer runs on a single pinned thread during each 200ms interval. At each interval, the optimizer estimates the access latency of each device by comparing counters from the Linux block-layer [10] to measurements from the previous interval. Similar to prior systems [64], we apply Exponentially Weighted Moving Average (EWMA) to measured latency to smooth out short-term fluctuations and maintain long-term stability. The optimizer leverages this smoothed latency signal to guide migration decisions and adjust routing accordingly. We set $\theta = 0.05$, a commonly used tolerance in tuning-based systems [64], to treat values as approximately equal, striking a balance between stability and responsiveness. MOST demonstrates robust performance across diverse workloads without requiring fine-tuning, indicating low sensitivity to the specific choice of θ. We also adopt *ratioStep* = 0.02, following guidance from similar systems (e.g., Orthus [69]), which works well across different workloads. Cerberus introduces approximately 1.5k LOC to CacheLib, leveraging and extending the core HeMem tiering logic.

4 Evaluation

We evaluate Cerberus to answer the following questions.

- How does Cerberus compare to striping, caching (Orthus), and tiering (HeMem, BATMAN, and Colloid) for static workloads with different intensity levels and patterns? (§4.1)
- How quickly does Cerberus adapt to dynamic workloads? How does the number of writes performed by MOST compare to migration-based tiering approaches with load-balancing capabilities? (§4.2)
- How effective are Cerberus's techniques for mirror class



(a) **Random Read-only** (b) **Random Write-only** (c) **Sequential Writes** (d) **Read Latest**

Figure 4: **Static Workload.** *The workload is running on the Optane/NVMe hierarchy with 750GB working set. In (a), under intensity 2.0×, Colloid, Colloid+, Colloid++, Cerberus migrate 134GB, 122GB, 62GB, 50GB of data in total. In (b), under intensity 2.0×, Colloid, Colloid+, Colloid++, Cerberus migrate 0GB, 93GB, 80GB, 65GB of data in total.*

sizing, subpage management, and selective cleaning? (§4.3)

- How does CacheLib with Cerberus perform on static workloads? How does it perform on dynamic workloads? (§4.4)

Storage Configurations. We evaluate two storage hierarchies: Optane (performance) / NVMe (capacity), and NVMe (performance) / SATA (capacity). The latency and bandwidth of these three storage devices (750GB Intel Optane SSD P4800X, 1TB Samsung 960 NVMe SSD, 1TB Samsung 870 SATA SSD) are shown in Table 1. The server has a 40-core Intel Xeon Gold 5218R CPU @ 2.1GHz (Ubuntu 20.04) and 64GB DRAM.

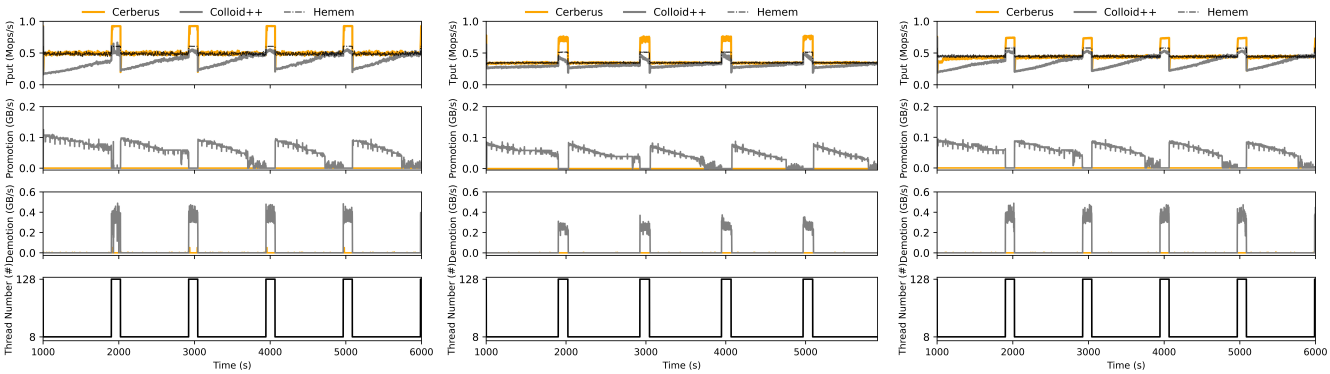
4.1 Comparison to Previous Approaches

We begin with a high-level comparison of Cerberus to other approaches for handling heterogeneous devices: striping, Orthus (caching), HeMem (classic tiering), BATMAN (with a static ratio matching the read bandwidth of the devices), and three versions of Colloid (a state-of-the-art tiering approach). To focus on the basic performance of these algorithms, we isolate the storage management layer from CacheLib and exercise that layer with controlled workloads; we begin with a static micro-benchmark in which multiple threads perform synchronous random reads and writes, given a skewed access pattern where a 20% hotset is accessed with 90% probability. Figure 4 compares steady-state throughput for read-only, write-only, sequential write, and read-latest workloads with varying intensities; 1.0× represents the minimum load at which the bandwidth of the performance device is saturated. We show that on static workloads Cerberus performs as well as or better than all other approaches.

Random Read-only. Even on this simple static workload, the other approaches do not match Cerberus. Striping delivers suboptimal performance, since it is bottlenecked by the slower device. Orthus delivers similar throughput to Cerberus, but Orthus achieves this by mirroring 690GB of data compared to only 50GB in Cerberus. HeMem (classic tiering) reaches its performance limit when the load intensity is 1.0× because HeMem does not offload traffic to the capacity device once the performance device's bandwidth is saturated; consequently, further load increases do not improve throughput. For BATMAN, no static allocation ratio works well for all load levels: BATMAN performs relatively well under high load because its allocation ratio is configured to match this bandwidth ratio for the hierarchy; however, at low load, this ratio causes traffic to be sent to the capacity device, reducing throughput. Colloid experiences significant performance degradation at intensity 2.0× due to migrations triggered by latency spikes arising from background activity; *Colloid++* improves this throughput, showing that Colloid is sensitive to parameter choice. As a result, Colloid and Colloid++ incur 2.68× and 1.24× more migration traffic than Cerberus (as shown in the caption). In summary, Cerberus delivers high throughput at both low and high load and also reduces device writes compared to migration-based approaches that attempt to balance the load.

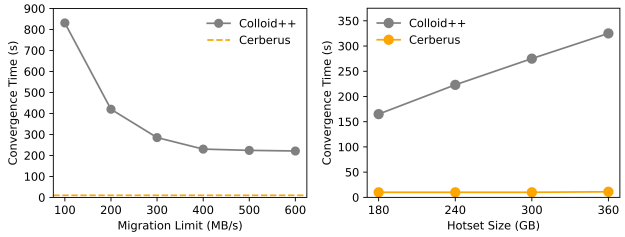
Random Write-only. We focus on those results that are noticeably different for writes versus reads. Orthus has a static write-back policy, and so does not balance write traffic or scale under high load. BATMAN no longer performs well at high load because a different allocation ratio is required to match the performance of write traffic versus read traffic across the two devices. Colloid does not perform write balancing and therefore exhibits performance similar to HeMem; *Colloid+* balances write traffic but suffers from migration overhead triggered by latency spikes, resulting in suboptimal throughput; *Colloid++* demonstrates improved performance, but still incurs migration overhead. For write-intensive workloads, Cerberus performs significantly better than other approaches, since it balances writes in the mirrored class and is robust to latency spikes.

Sequential Write. Sequential writes emulate popular applications with log-structured data (e.g., flash caches, file systems, and databases). As with random writes, Orthus cannot efficiently balance write traffic. Colloid behaves similarly to HeMem because it doesn't take write latency into account when balancing. Interestingly, *Colloid+* and *Colloid++* perform worse due to additional interfering migrations when balancing write latency; in a sequential workload, demoted blocks are not re-accessed, which makes these migrations ineffective. As desired, Cerberus dynamically allocates new writes proportionally across devices: when the performance device becomes saturated, writes are allocated directly on the capacity device.



(a) **Read-only** (b) **Write-only** (c) **RW-mixed**

Figure 5: **Dynamic Bursty Workload.** *Running on Optane/NVMe hierarchy with 1.2TB working set and same skewness as §4.1. Colloid++ migrates 282GB, 214GB, 260GB of data to the performance device and 262GB, 186GB, 239GB of data to the capacity device under workload (a), (b) and (c), respectively. In comparison, Cerberus migrates 87GB, 107GB and 64GB of data to the capacity device.*



(a) **Migration Limit** (b) **Hotset Size**

Figure 6: **Limitation of Migration-based Approach.** *The same workload as Figure 5a.*

Read Latest. This workload uses a 50% write ratio, where 20% of the newly-written blocks have a 90% probability of being read. Colloid performs worse than HeMem at intensities higher than 1.0× due to migration traffic; these migrations are ineffectual since the migrated blocks soon become cold as new blocks are written into the system. Colloid+ and Colloid++ perform more migration, resulting in worse performance. Cerberus efficiently balances the workload by dynamically allocating a portion of new writes to the capacity device when the performance device is saturated; as a result, Cerberus effectively uses the bandwidth of each device as the intensity increases.

Summary. For a variety of static workloads, Cerberus achieves better throughput than striping, Orthus, BATMAN, HeMem, and Colloid. We omit BATMAN in subsequent experiments since it performs worse than Colloid or its variants.

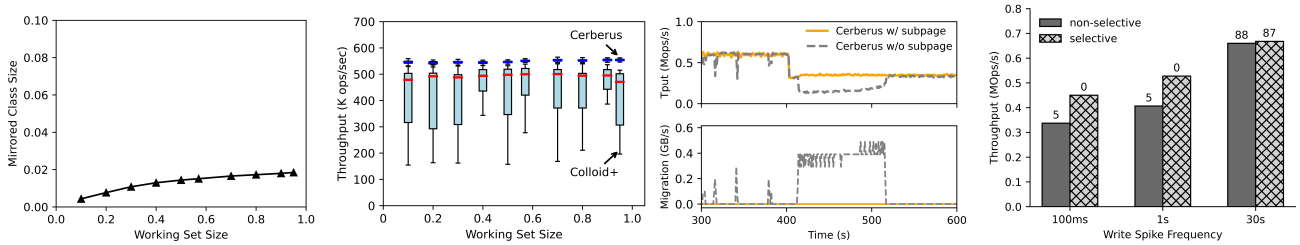
4.2 Dynamic Workloads

Dynamic time-varying workloads are challenging for previous approaches. To compare Cerberus to HeMem and Colloid++, we create a micro-benchmark modeling a bursty workload similar to those in practice [18]. After pre-warming the system with intensive load for 1000 seconds, the benchmark transitions to generating a 2-minute burst every 15 minutes. Figure 5 shows the delivered throughput and amount of data promoted and demoted as a function of load (i.e., thread count) for read-only, write-only, and read-write workloads.

Read-only workload. Before 1000 seconds, the systems operate under high load; in this warm-up period, Colloid demotes and Cerberus mirrors approximately 7% of the hot data to the capacity device. At 1000 seconds, when workload intensity drops, Colloid promotes the hotset back to the performance device, causing intensive promotion traffic (as shown in the second graph), which negatively impacts throughput (first graph). In contrast, within 10 seconds Cerberus efficiently rebalances traffic by routing requests to mirrored data back to the performance device. When the 2-minute load burst occurs after about 1000 seconds, the performance device is saturated, causing its latency to surpass that of the capacity device. Colloid again demotes the hotset back to the capacity device (as depicted in the third graph). Consequently, Colloid performs worse than HeMem, which simply keeps hot data on the performance device and does not conduct any load balancing. Again, during the load burst, Cerberus efficiently offloads excess traffic by redirecting requests for mirrored data back to the capacity device. Cerberus matches HeMem's performance under low load and achieves 1.53× higher throughput during workload bursts since Cerberus also utilizes the capacity device.

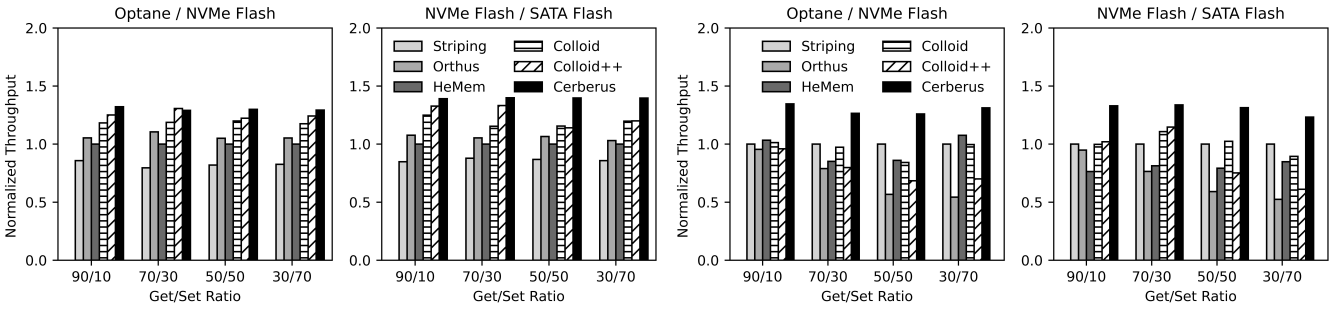
Write-only and Read-Write Workloads. The other workloads in Figure 5b and Figure 5c demonstrate similar behavior. Cerberus is able to load balance aligned 4KB writes by using subpages that track the valid versus invalid portions of each 2MB segment. For the write-only workload, Cerberus achieves 1.48× higher throughput than HeMem under high load conditions.

Disadvantages of Migrating Data. Colloid's reliance on migrating data has a number of disadvantages: not only does migration interfere with foreground traffic, but it also increases device writes and the time required for convergence. As shown in the caption of Figure 5, Colloid migrates on aver-



(a) **Working set vs. Mirrored** (b) **Working set vs. Throughput.** (c) **Subpage Management.** (d) **Selective Cleaning.**

Figure 7: **In-depth Analysis.** *In (a), workload is random read/write mixed (50% writes) at a high load (128 threads), varying the working set size, which is shown as a fraction of the total storage system capacity. The hotset is 20% of the working set with 90% access probability. In (d), the workloads are 256 threads workload with occasional write spikes every 0.1, 1, 30 seconds.*



(a) **Small Object Cache** (b) **Large Object Cache**

Figure 8: **Lookaside Cache Workload.** *The DRAM cache is restricted to 200MB to stress each flash cache component in CacheLib. In (a), the workloads are 256 threads get/set-mixed Zipfian with 25M keys and 1KB value. SOC is set to 100GB, one third of the total capacity (100GB/200GB hierarchy). In (b), the workloads are 256 threads get/set-mixed Zipfian with 5M keys and 16KB value.*

age 252 GB to the performance and 229 GB to capacity tiers across three representative workloads. In contrast, Cerberus only mirrors on average 86 GB of data to the capacity tier. For instance, under a read-only workload, Colloid issues 282 GB to the performance tier and 262 GB to the capacity tier, while Cerberus writes just 87 GB. Running this workload for one day results in 6.6 DWPD and 3.1 DWPD on the performance and capacity tiers, respectively. With a warranted endurance of 30 DWPD over 5 years [8], the performance-tier device lasts 5.0 years with Cerberus. Colloid reduces the lifespan to 4.1 years, an 18% drop. For the 1 TB capacity-tier device rated for 0.37 DWPD over 3 years [14], Colloid introduces 3.1 DWPD of migration writes, shortening the lifespan from 3.0 years to 129 days, an 88% reduction.

While Colloid's migration overhead can be controlled, limiting the rate of migration increases Colloid's convergence time, which is already significantly longer than Cerberus. Figure 6a shows Colloid's convergence time on the read-only workload when transitioning from low to high load given migration limits between 100MB/s (5 DWPD) to 600MB/s (30 DWPD). With a limit of 100MB/s, Colloid requires more than 800 seconds to adapt to a load increase, making it ineffective for workloads containing shorter burst intervals. In contrast, Cerberus requires less than 10 seconds to adapt to load changes.

Relying exclusively on migration to adapt to workload changes, also causes convergence time to increase as the size of the hotset increases. Figure 6b shows that Colloid's convergence time increases with larger (read-only) hotset since more data must be demoted to offload traffic from the performance device to the capacity device. In contrast, Cerberus's convergence time is independent of the hotset size, since no migration is performed once the data is mirrored.

Summary. Cerberus integrates mirroring into classic tiering, allowing load adjustments without frequent migrations; this significantly enhances adaptability to bursty workloads and reduces overall write traffic. In contrast, Colloid, and other tiering approaches, lack intra-tier redundancy and rely exclusively on migration for load balancing, since data can only be accessed from one location; under bursty workloads, Colloid incurs extensive writes for migration, greatly reducing device lifespan and resulting in performance worse than HeMem.

4.3 Cerberus In-depth Analysis

We now perform an in-depth analysis of Cerberus's techniques for determining the size of the mirrored class, tracking the state of subpages in mirrored data, and selective cleaning.

Mirrored Class Size. Cerberus performs efficient load balancing with a relatively small mirrored class. Figure 7a shows the amount of space needed for Cerberus' mirrored class as a function of the workload's working set size. Even when the

Name	Get	Set	LoneGet	LoneSet	Key Size (B)	Avg Value Size (B)
A (flat-kvcache)	0.98	0	0.02	0	16-255	335
B (graph-leader)	0.82	0	0.18	0	8-16	860
C (kvcache-reg)	0.87	0.12	1.04e-05	0.003	8-16	33112
D (kvcache-wc)	0.6	0	8.2e-06	0.21	8-16	92422

Table 4: **Production Trace Distributions from CacheBench.** *Flat-kvcache and graph-leader are from an application cache with small value sizes, resulting in mostly random traffic; kvcache-reg and kvcache-wc are from a storage cache with large value sizes, leading to log-structured traffic. LoneGet and LoneSet are requests for a key not present in the cache.*

working set size reaches 95% of the total system capacity, Cerberus only mirrors 1.8% of the total data. Figure 7b shows the corresponding throughput of Colloid+ and Cerberus. The throughput of Colloid+ is highly unstable due to interference from frequent migrations; Cerberus consistently demonstrates higher and more stable throughput due to its effective use of a small amount of mirrored data.

Mirrored Data Subpages. The use of subpages within the mirrored class enables Cerberus to efficiently handle write requests. Figure 7c compares the behavior of Cerberus with subpages to Cerberus without subpages on a 4KB write-only workload undergoing a sudden load drop (from 128 to 8 threads) at 400 seconds. After the load drop, Cerberus with subpages immediately redirects writes back to the performance device, quickly adapting to light load without requiring any data migrations. In contrast, Cerberus without subpages must migrate entire segments back to the performance device, as each write request is smaller than the segment size (2MB). Cerberus without subpages incurs additional migrations and significantly longer convergence times.

Selective Cleaning. Cleaning of mirrored data is rarely needed in Cerberus: cleaning is needed only for workloads containing a relatively small percentage of writes that invalidate a mirrored copy of data that is then frequently read (e.g., caching for ML models where write spikes occur when model parameters are refreshed [74]). Figure 7d shows the efficiency of Cerberus's selective cleaning under a read-intensive workload with occasional write spikes. The number atop each bar represents the clean data percentage. We make two key observations. First, non-selective cleaning results in a 25% decrease in cache throughput, but only a 5% increase in the clean block percentage by cleaning those frequently written data. Second, Cerberus's selective cleaning policy effectively filters out data subject to high-frequency writing while cleaning long-term written data (writes every 30s).

4.4 CacheLib

We now compare the end-to-end performance of CacheLib using Cerberus for its storage management layer, as compared to striping (CacheLib's default), caching (Orthus), and tiering (HeMem, Colloid and Colloid++). For workloads, we use CacheBench [3], a highly configurable benchmarking tool bundled with CacheLib that can model real-world production cache workloads [4]. We also extend CacheBench to support

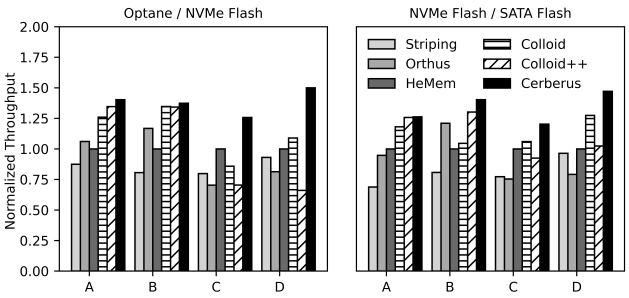


Figure 9: **Production Workloads.** *DRAM cache is set to 1GB. For flat-kvcache and graph-leader, the SOC is set to one third of the total hierarchy capacity. We use 256 threads for flat-kvcache, graph-leader, and kvcache-wc and 80 threads for kvcache-reg. The cache throughput is normalized to HeMem's performance.*

Zipfian distributions and dynamic workloads.

4.4.1 Static Workloads

We examine workloads that stress CacheLib's Small Object Cache (SOC) as well as its Large Object Cache (LOC).

Small Object Cache. We begin with 1KB lookaside cache workloads varying the Get/Set ratio across the two storage hierarchies. Figure 8a shows that striping, Orthus, and HeMem continue to deliver suboptimal performance. Colloid and Colloid++ perform worse than Cerberus due to migration overhead caused by latency spikes, especially on the NVMe/SATA Flash capacity layer where NVMe and SATA exhibit more severe read/write interference than the Optane device.

Large Object Cache. The 16KB lookaside cache workload stresses the Large Object Cache, which stores large key-value pairs in a sequential log and an in-memory index; this results in sequential writes to the storage management layer and reads primarily to the most-recently-written blocks (i.e., the log head). Figure 8b shows that HeMem and Colloid cannot utilize the capacity device bandwidth once the performance device becomes saturated. Cerberus performs optimally across all workloads, achieving up to 1.36× higher throughput on Optane/NVMe and up to 1.54× higher throughput on NVMe/SATA.

CPU Overhead. Under the 4KB random workload that stresses the mirroring mechanism most (Figure 8a), Cerberus slightly increases CPU utilization (0–1.5%) at 256 threads compared to the best-performing baseline, Colloid++, due to mirrored-page tracking and routing logic.

4.4.2 Production Cache Workloads

We evaluate four long-running production cache workloads provided by Meta [4], as detailed in Table 4. Figure 9 summarizes the results. For Workload A (lookaside cache workload) and Workload B (graph leader workload), Cerberus performs slightly better than Colloid and Colloid++, as these workloads involve small key-value pairs and stress the Small Object Cache. On Workloads C and D, Cerberus significantly

Device	Workload	Metric	Striping	Orthus	HeMem	Colloid	Colloid++	Cerberus
Optane NVMe	A	Avg (ms)	0.81	0.67	0.71	0.56	0.52	**0.50**
		P99 (ms)	15.76	18.18	19.24	9.59	9.00	**7.33**
	B	Avg (ms)	0.73	0.61	0.59	0.45	0.44	**0.43**
		P99 (ms)	7.00	14.28	12.83	4.12	4.12	**3.65**
	C	Avg (ms)	0.76	0.86	0.59	0.67	0.88	**0.47**
		P99 (ms)	9.53	10.69	6.50	7.78	8.85	**5.43**
	D	Avg (ms)	9.67	12.38	8.26	7.15	8.32	**3.16**
		P99 (ms)	67.68	272.00	73.63	72.93	86.32	**27.76**
NVMe SATA	A	Avg (ms)	2.30	1.76	1.58	1.35	1.21	**1.15**
		P99 (ms)	46.43	27.96	22.98	25.98	21.31	**20.40**
	B	Avg (ms)	1.90	1.41	1.54	1.48	1.17	**1.10**
		P99 (ms)	23.40	16.84	11.59	14.15	9.34	**8.54**
	C	Avg (ms)	1.95	2.00	1.49	1.40	1.62	**1.23**
		P99 (ms)	18.18	18.42	12.35	16.50	18.50	**12.31**
	D	Avg (ms)	27.55	32.15	26.69	19.98	16.68	**16.09**
		P99 (ms)	239.86	364.20	160.10	158.07	170.29	**101.85**

Table 5: **Average and P99 GET Latency (ms) of Production Workloads.**

outperforms others, as these stress the Large Object Cache (LOC); Cerberus effectively distributes writes through its dynamic write allocation. Cerberus demonstrates better performance across all four production workloads due to its efficient mirroring-based load balancing and dynamic allocation. Compared to Colloid, Cerberus achieves an average throughput improvement of 1.24× on the Optane/NVMe hierarchy and 1.17× on the NVMe/SATA hierarchy.

Latency. Table 5 shows that Cerberus reduces average latency by 14% and P99 latency by 19% on average compared to the best-performing baseline. Striping performs worst on workloads A and B due to bottlenecks from the slower device, while Orthus performs worst on log-heavy workloads C and D, consistent with Figures 4c and 4d. Performance gains are more substantial on the Optane/NVMe hierarchy, with 20% lower average latency and 26% lower P99 latency compared to the best-performing baseline. In comparison, the NVMe/SATA hierarchy shows smaller improvements, with 6.6% and 12% reductions respectively.

4.4.3 Dynamic Workload

We compare how well Colloid and Cerberus handle load changes using the CacheBench benchmark [4]. The bursts happen every 180 seconds and last for 60 seconds, matching those studied in [18] for data center workloads. Figure 10 shows that Colloid struggles to adapt to these bursty workloads, generating significant migration traffic. In contrast, Cerberus efficiently adapts to bursty workloads without incurring any migration overhead.

4.4.4 YCSB

We compare Cerberus to striping, tiering (HeMem), and caching (Orthus) under the YCSB benchmark across two local hierarchies shown in Figure 11. Since YCSB does not natively handle cache misses, we extended it to implement a lookaside caching pattern [18], where cache misses trigger a fetch from the backing store (simulated with a 1.5ms delay) and re-insertion into the cache. Cerberus shows up to 1.43× higher throughput and 30% less P99 latency compared to the best-performing system.

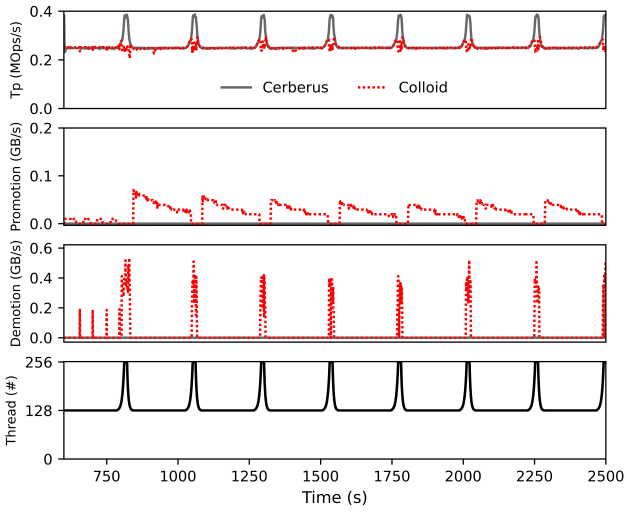


Figure 10: **Dynamic Cache Workload.** *Read-heavy workload with 95% GET and 5% SET operations on Optane/NVMe hierarchy, where 20% of keys belong to the hotset; the hotset is accessed uniformly at random with a 90% probability. 25 million key-value pairs, with value sizes ranging from 2KB to 4KB; 1 billion operations. Size of SOC is 450GB.*

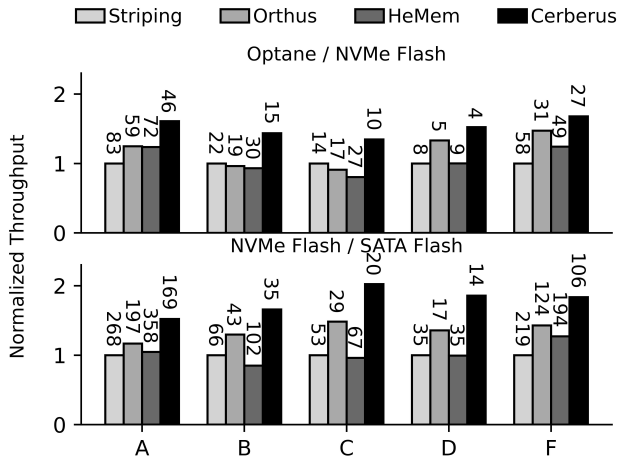


Figure 11: **YCSB.** *CacheLib uses the default parameters from CacheBench with 4GB DRAM cache. All the workloads are Zipfian (θ = 0.8) with 1KB values and 16-byte keys running under 256 threads with 20M records. The cache throughput is normalized to the throughput of the default (striping) system and the number above each bar is P99 latency in microseconds. Workload E is excluded due to CacheLib's lack of support for range queries.*

5 Discussion

Multi-tier Extensions. MOST's mechanism naturally extends beyond the two-tier scenario. For example, data can be mirrored across multiple tiers, allowing requests to be

dynamically routed to the tier with the lowest latency. This generalization calls for a more sophisticated optimization policy, which we leave as future work.

Consistency. MOST can be extended to provide stronger consistency guarantees. One possible approach is to maintain a write-ahead log for mapping updates, such as those triggered by data migration. We leave a full study as future work.

Performance Isolation. Currently, MOST transparently manages the underlying storage devices at the block level and is unaware of which tenant a given request belongs to. One potential solution is to use hints to associate each request with its corresponding tenant. With this additional metadata, MOST can be extended to support and enforce performance isolation policies, such as fairness and quality of service (QoS), across multiple tenants.

6 Related Work

Tiered Memory/Storage Systems. Our paper shares scope with research on tiered memory and storage [15, 25, 31, 37, 39, 44, 54, 56, 61, 65, 67, 77] for managing heterogeneous hierarchies of different devices, including DRAM, NVM, CXL-enabled memory, NVMe SSD, SATA SSD, and hard drives. vTMM [60], a tiered memory management system for virtual machines, optimizes memory-access tracking through page-modification logging. Nimble [73] is an OS-level system designed to improve page migration throughput. HeterOS [33] functions as an OS/VMM-level heterogeneous memory manager, coordinating memory placement and migration with guest OSes. Thermostat [16] identifies hot memory pages using page table sampling. All of these systems are orthogonal to MOST, as they focus on memory-access tracking, hot/cold data identification, and data migration rather than load balancing across tiers.

Strata [38] is a cross-media file system designed to mitigate the limitations of NVM, SSD, and HDD in a traditional storage hierarchy. Ziggurat [79] migrates cold data to lower tiers in an NVM-disk tiered file system. Spitfire [80] is a three-tier caching-based buffer manager that leverages machine learning for optimal data placement. AutoRAID [66] integrates mirroring for improved write performance with RAID-5 for cost-effective capacity. hStorage-DB [42] enforces QoS policies for requests in hybrid storage systems. However, these systems are load-unaware and not able to fully utilize lower-tier bandwidth. PolyStore [58] is a meta layer built on top of storage medium-optimized file systems that maximizes cumulative storage bandwidth. However, PolyStore is optimized exclusively for file systems and does not maintain cross-tier duplication, therefore not able to efficiently adapt to dynamic workload.

Distributed multi-tier caching systems [41, 55, 57] broadly share a similar problem space with our work. However, their approaches differ significantly from MOST in both design and objectives. For instance, EC-CACHE [55] uses erasure coding instead of mirroring and is limited to load balancing within a single tier. ELECT [57] replicates data within a single tier and lacks support for cross-tier load balancing. eMRC [41] focuses on multi-tier miss ratio approximation but does not address the load balancing problem, making its objective orthogonal to ours. In contrast, MOST introduces a novel data management model that supports selective mirroring and adaptive load balancing across a heterogeneous storage hierarchy.

Machine learning-based storage and memory tiering approaches (e.g., IDT [50], RLTiering [40], A3C [76], ArtMem [21]) target a similar problem space as our work. These systems typically utilize reinforcement learning to guide data placement decisions but are built on traditional single-copy tiering models. As shown in our evaluation of Colloid, such models struggle with dynamic and time-varying workloads. Like Colloid, these ML-based approaches are fundamentally constrained by the inherent limitations of single-copy designs and fail to address the critical challenge of load balancing across tiers in heterogeneous hierarchies.

Storage Aware Caching/Tiering. Our work aligns with research on storage-aware caching and tiering [17, 34, 53, 69–71]. Wu et al. [71] identified a similar issue, where SSDs become throughput bottlenecks, and proposed migrating data to HDDs when SSD response times exceed those of HDDs. However, this migration-only approach reacts slowly to workload changes and incurs high migration overhead as Colloid. SIB [34] targets HDFS clusters with SSDs and HDDs, using SSDs as write buffers and offloading reads to HDDs. LBICA [17] implements cache load balancing by halting new allocations to performance devices under burst loads but does not balance read traffic. Like Orthus, these caching-based systems fail to fully utilize the capability of modern storage hierarchies.

7 Conclusion

We introduced Mirror-Optimized Storage Tiering, a new tier-based approach optimized for modern heterogeneous storage hierarchies. We demonstrate that with a small amount of mirrored data, MOST improves throughput, reduces latency, enhances adaptability, and decreases migration-induced writes under dynamic workloads. In the future, hybrid approaches that integrate elements of caching, tiering, and RAID may offer further benefits, potentially leading toward a "unified theory" of storage hierarchy management.

8 Acknowledgement

We thank Huaicheng Li (our shepherd), the anonymous reviewers and the members of ADSL for their valuable input. This material was supported by funding from the NSF under the award number CNS-2402859 and the taxpayers of Wisconsin and the USA. We thank NetApp, Microsoft, InfluxData, and GE HealthCare for their generous support. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or any other institutions.

References

[1] Amazon EBS. https://aws.amazon.com/ebs/pricing/.

[2] CacheLib. https://cachelib.org/.

[3] CacheLib Cachebench. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_Overview.

[4] CacheLib Cachebench Real Workload. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval.

[5] Dell 1.6TB, Enterprise, NVMe, Mixed Use Drive, U.2, Gen4 with Carrier. https://www.dell.com/en-us/shop/dell-16tb-enterprise-nvme-mixed-use-drive-u2-gen4-with-carrier/apd/400-bkfk/storage-drives-media#support_section.

[6] Intel Open Cache Acceleration Software. https://open-cas.github.io/.

[7] Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[8] Intel Optane SSD DC P4800X. https://www.intel.com/content/www/us/en/products/sku/97154/intel-optane-ssd-dc-p4800x-series-750gb-2-5in-pcie-x4-3d-xpoint/specifications.html.

[9] KIOXIA FL6 XL-FLASH SSD. https://americas.kioxia.com/en-us/business/ssd/enterprise-ssd/fl6.html.

[10] Linux Block Layer Statistics. https://www.kernel.org/doc/Documentation/block/stat.txt.

[11] Netflix Technology Blog. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef.

[12] NVMe over Fabrics Specification. https://nvmexpress.org/specification/nvme-of-specification/.

[13] Samsung 870 EVO Flash SSD. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/870-evo-sata-2-5-ssd-1tb-mz-77e1t0b-am/.

[14] Samsung 960 EVO Flash SSD. https://semiconductor.samsung.com/consumer-storage/internal-ssd/960evo/.

[15] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 971–985, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. SIGPLAN Not., 52(4):631–644, apr 2017.

[17] 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.

[18] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 753–768. USENIX Association, November 2020.

[19] Dina Bitton and Jim Gray. Disk shadowing. In Proceedings of the 14th International Conference on Very Large Data Bases, VLDB '88, page 331–338, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[20] Arthur W. Burks, Herman H. Goldstine, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument (1946), page 39–48. Ablex Publishing Corp., USA, 1989.

[21] Juneseo Chang, Wanju Doh, Yaebin Moon, Eojin Lee, and Jung Ho Ahn. Idt: Intelligent data placement for multi-tiered main memory with reinforcement learning. In Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing, pages 69–82, 2024.

[22] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: high-performance, reliable secondary storage. ACM Comput. Surv., 26(2):145–185, jun 1994.

[23] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. BATMAN: techniques for maximizing system bandwidth of memory systems with stacked-DRAM. In Proceedings of the International Symposium on Memory Systems, MEMSYS '17, page 268–280, New

York, NY, USA, 2017. Association for Computing Machinery.

[24] Toni Cortes and Jesús Labarta. A case for heterogeneous disk arrays. In Proceedings IEEE International Conference on Cluster Computing. CLUSTER 2000, pages 319–325. IEEE, 2000.

[25] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[26] 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.

[27] Brian C. Forney, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Storage-Aware caching: Revisiting caching for heterogeneous storage systems. In Conference on File and Storage Technologies (FAST 02), Monterey, CA, January 2002. USENIX Association.

[28] Jamel Gafsi and Ernst W. Biersack. Data striping and reliability aspects in distributed video servers. Cluster Computing, 2(1):75–91, jan 1999.

[29] Binny S Gill. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In FAST, volume 8, pages 1–17, 2008.

[30] 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.

[31] Morteza Hoseinzadeh. A survey on tiering and caching in high-performance storage systems. arXiv preprint arXiv:1904.11560, 2019.

[32] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. Ssds striking back: The storage jungle and its implications on persistent indexes. In 11th Conference on Innovative Data Systems Research, CIDR, pages 9–12, 2022.

[33] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos — os design for heterogeneous memory management in datacenter. In 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), pages 521–534, 2017.

[34] 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, 2018.

[35] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[36] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17, page 345–359, New York, NY, USA, 2017. Association for Computing Machinery.

[37] K.R. Krish, Ali Anwar, and Ali R. Butt. hats: A heterogeneity-aware tiered storage for hadoop. In 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pages 502–511, 2014.

[38] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In Proceedings of the 29th Symposium on Operating Systems Principles, pages 17–34, 2023.

[40] Mingyu Liu, Li Pan, and Shijun Liu. Rltiering: a cost-driven auto-tiering system for two-tier cloud storage using deep reinforcement learning. IEEE Transactions on Parallel and Distributed Systems, 34(2):501–518, 2022.

[41] Zhang Liu, Hee Won Lee, Yu Xiang, Dirk Grunwald, and Sangtae Ha. {eMRC}: Efficient miss ratio approximation for {Multi-Tier} caching. In 19th USENIX Conference on File and Storage Technologies (FAST 21), pages 293–306, 2021.

[42] Tian Luo, Rubao Lee, Michael Mesnier, Feng Chen, and Xiaodong Zhang. hstorage-db: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. Proc. VLDB Endow., 5(10):1076–1087, jun 2012.

[43] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. Multi-clock: Dynamic tiering for hybrid memory systems. In 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA'22), 2022.

[44] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[45] Sara McAllister, Benjamin Berg, Daniel S Berger, George Amvrosiadis, Nathan Beckmann, Gregory R Ganger, et al. {FairyWREN}: A sustainable cache for emerging {Write-Read-Erase} flash interfaces. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 745–764, 2024.

[46] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

[47] NetApp. How SSD Cache works. docs.netapp.com/us-en/e-series-santricity/sm-storage/how-ssd-cache-works.html, 2023.

[48] 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 10th USENIX Conference on File and Storage Technologies (FAST 12), San Jose, CA, February 2012. USENIX Association.

[49] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery.

[50] Nedjah Oussama, Mokhtari Omar, Boumahdi Fatima, and Mancer Yasmine. Autonomous data tiering using reinforcement learning for 3-tier hierarchical storage. In International Symposium on Modelling and Implementation of Complex Systems, pages 171–181. Springer, 2024.

[51] B. Ozden, R. Rastogi, and A. Silberschatz. Disk striping in video server environments. In Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems, pages 580–589, 1996.

[52] 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.

[53] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern hardware. In Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys '23, page 557–573, New York, NY, USA, 2023. Association for Computing Machinery.

[54] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J. Freedman. Efficient compactions between storage tiers with prismdb. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, page 179–193, New York, NY, USA, 2023. Association for Computing Machinery.

[55] KV 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, 2016.

[56] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[57] Yanjing Ren, Yuanming Ren, Xiaolu Li, Yuchong Hu, Jingwei Li, and Patrick PC Lee. {ELECT}: Enabling erasure coding tiering for {LSM-tree-based} storage. In 22nd USENIX Conference on File and Storage Technologies (FAST 24), pages 293–310, 2024.

[58] Yujie Ren, David Domingo, Jian Zhang, Paul John, Rekha Pitchumani, Sanidhya Kashyap, and Sudarsun Kannan. {PolyStore}: Exploiting combined capabilities of heterogeneous storage. In 23rd USENIX Conference

on File and Storage Technologies (FAST 25), pages 539–555, 2025.

[59] Kenneth Salem and Hector Garcia-Molina. Disk striping. In 1986 IEEE Second International Conference on Data Engineering, pages 336–342, 1986.

[60] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. Vtmm: Tiered memory management for virtual machines. In Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys '23, page 283–297, New York, NY, USA, 2023. Association for Computing Machinery.

[61] 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, page 588–602, New York, NY, USA, 2023. Association for Computing Machinery.

[62] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[63] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. {RIPQ}: Advanced photo caching on flash for facebook. In 13th USENIX Conference on File and Storage Technologies (FAST 15), pages 373–386, 2015.

[64] Midhul Vuppalapati and Rachit Agarwal. Tiered memory management: Access latency is the key! In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, pages 79–94, 2024.

[65] Hui Wang and Peter Varman. Balancing fairness and Efficiency in tiered storage systems with Bottleneck-Aware allocation. In 12th USENIX Conference on File and Storage Technologies (FAST 14), pages 229–242, Santa Clara, CA, February 2014. USENIX Association.

[66] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The hp autoraid hierarchical storage system. ACM Trans. Comput. Syst., 14(1):108–136, feb 1996.

[67] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. Proc. VLDB Endow., 12(6):624–638, feb 2019.

[68] Kan Wu, Andrea C. Arpaci-Dusseau, and Remzi H. 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. USENIX Association.

[69] 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.

[70] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. NyxCache: Flexible and efficient multi-tenant persistent memory caching. In 20th USENIX Conference on File and Storage Technologies (FAST 22), pages 1–16, Santa Clara, CA, February 2022. USENIX Association.

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

[72] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad:{Non-Exclusive} memory tiering via transactional page migration. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 19–35, 2024.

[73] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.

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

[75] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. Autotiering: automatic data placement manager in multi-tier all-flash datacenter. In 2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC), pages 1–8. IEEE, 2017.

[76] Xinyue Yi, Hongchao Du, Yu Wang, Jie Zhang, Qiao Li, and Chun Jason Xue. Artmem: Adaptive migration in reinforcement learning-enabled tiered memory. In Proceedings of the 52nd Annual International Symposium on Computer Architecture, pages 405–418, 2025.

[77] Gong Zhang, Lawrence Chiu, and Ling Liu. Adaptive data migration in multi-tiered storage based cloud environment. In 2010 IEEE 3rd International Conference on Cloud Computing, pages 148–155, 2010.

[78] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, and Jiesheng Wu. What's the story in EBS glory: Evolutions and lessons in building cloud block store. In 22nd USENIX Conference on File and Storage Technologies (FAST 24), pages 277–291, Santa Clara, CA, February 2024. USENIX Association.

[79] 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.

[80] 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, page 2195–2207, New York, NY, USA, 2021. Association for Computing Machinery.