The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus

Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan[†],
Rathijit Sen[‡], Kwanghyun Park[‡], Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin–Madison [†]VMware Research [‡]Microsoft

Abstract. We introduce *multi-factor caching* (MFC), a novel approach to caching in modern storage hierarchies. MFC subverts traditional thinking in storage caching – namely, the sole focus on maximizing hit rate – and thus realizes significantly higher performance than classic caching. MFC accounts for both workload and device characteristics to make allocation and access decisions, thus maximizing performance (e.g., high throughput, low 99%-ile latency). We implement MFC in Orthus-CAS (a block-layer caching kernel module) and Orthus-KV (a user-level caching layer for a key-value store). We show the efficacy of MFC via a thorough empirical study: Orthus-KV and Orthus-CAS offer significantly better performance (by up to 2×) than classic caching on various modern hierarchies, under a range of realistic workloads.

1 Introduction

The notion of a *hierarchy* (i.e., a *memory hierarchy* or *storage hierarchy*) has long been central to computer system design. Indeed, assumptions about the hierarchy and its fundamental nature are found throughout widely used textbooks [27, 44, 83]: "Since fast memory is expensive, a memory hierarchy is organized into several levels – each smaller, faster, and more expensive per byte than the next lower level, which is farther from the processor. [44]"

To cope with the nature of the hierarchy, systems usually employ two strategies: *caching* [3, 71] and *tiering* [5, 41, 91]. Consider a system with two storage layers: a (fast, expensive, small) performance layer and a ((slow, cheap, large) capacity layer. With caching, all data resides in the capacity layer, and copies of hot data items are placed, via cache replacement algorithms, in the performance layer. Tiering also places hot items in the performance layer; however, unlike caching, it migrates data (instead of copying) on longer time scales. With a high-enough fraction of requests going to the fast layer, the overall performance approaches the peak performance of the fast layer. Consequently, classic caching and tiering strive to ensure that most accesses hit the performance layer.

While this conventional wisdom of optimizing hit rates may remain true for traditional hierarchies (e.g., CPU caches and DRAM, or DRAM and hard disks), rapid changes in storage devices have complicated this narrative within the modern storage hierarchy. Specifically, the advent of many new non-volatile memories [19, 52, 75] and low-latency SSDs [8, 13, 16] has introduced devices with (sometimes) overlapping performance characteristics. Thus, it is essential to rethink how such devices must be managed in the storage hierarchy.

To understand this issue better, consider a two-level hierarchy with a traditional Flash-based SSD as the capacity layer, and a newer, seemingly faster Optane SSD [8] as the performance layer. As we will show (§3.2), in some cases, Optane outperforms Flash, and thus the traditional caching/tiering arrangement works well. However, in other situations (namely, when the workload has high concurrency), the performance of the devices is similar (i.e., the storage hierarchy is actually not a hierarchy), and thus classic caching and tiering do not utilize the full bandwidth available from the capacity layer. A different approach is needed to maximize performance.

To address this problem, we introduce *multi-factor caching* (MFC), a new approach to caching for modern storage hierarchies. MFC delivers maximal performance from modern devices despite complex device characteristics and changing workloads. We use the term *multi-factor* to emphasize that the cache scheduler monitors the delivered performance from all devices; in contrast, classic caching optimizes only the single factor of hit rate and on only one device. The key insight of MFC is that when classic caching would send more requests to the performance device than is useful, some of that excess load can be dynamically moved to the capacity device. This improves upon classic caching in two ways. First, by monitoring performance and adapting the requests sent to each device, MFC delivers additional useful performance from the capacity device. Second, MFC avoids data movement between the devices when this movement does not improve performance. MFC always performs as well or better than classic caching.

Previous work has addressed some of the limitations of caching [18, 54], offloading excess writes from SSDs to underlying hard drives. However, as we show (§6.4), they have two critical limits: they do not redirect accesses to items present in the cache (hits), and they do not adapt to changing workloads and concurrency levels (which is critical for modern devices).

We implement MFC in two systems: Orthus-CAS, a generic block-layer caching kernel module [31], and Orthus-KV, a user-level caching layer for an LSM-tree key-value store [62]. Under light load, Orthus implementations behave like classic caching; in other situations, they offload excess load at the caching layer to the capacity layer, improving performance. Through rigorous evaluations, we show that Orthus implementations greatly improve performance (up to 2×) on various real devices (such as Optane DCPM, Optane SSD, Flash SSD) and other simulated ones for a range of workloads (YCSB [33] and ZippyDB [30]). We also show MFC is robust to dynamic

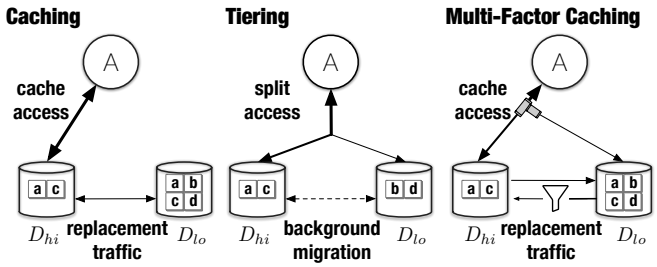


Figure 1: **Caching vs. Tiering vs. Multi-Factor Caching.** *Multi-factor caching is our new approach, described in §4.*

workloads, quickly adapting to load and locality changes. Finally, we compare MFC against prior caching strategies and demonstrate its advantages. Overall, the multi-factor approach extracts high performance from modern storage hierarchies.

2 Motivation

In this section, we discuss classic solutions to storage hierarchy management. We then review current and near-future devices and discuss how they alter the storage hierarchy.

2.1 Managing the Storage Hierarchy

A storage hierarchy is composed of multiple heterogeneous storage devices and policies for transferring data between those devices. For simplicity, we assume a two-device hierarchy, consisting of a *performance* device, D_{hi}, and a *capacity* device, D_{lo}; commonly, D_{hi} is more expensive, smaller, and faster, whereas D_{lo} is cheaper, larger, and slower.

Traditionally, two approaches have been used for managing such a hierarchy: *caching* and *tiering* (Figure 1). With caching, popular (hot) data is copied from D_{lo} into D_{hi} (e.g., on each miss); to make room for these hot data items, the cache evicts less popular (cold) data, as determined by algorithms such as ARC, LRU, or LFU [4, 63, 65, 72, 87, 101]. The granularity of data movement is usually small, e.g., 4-KB blocks.

Tiering [41, 55, 79], similar to caching, usually maintains hot data in the performance device. However, unlike caching, when data on D_{lo} is accessed, it is not necessarily promoted to D_{hi}; data can be directly served from D_{lo}. Data is only periodically migrated between devices on longer time scales (over hours or days) and longer-term optimizations determine data placement. Tiering typically does such migration at a coarser granularity (an entire volume or a large extent [41]). While caching can quickly react to workload changes, tiering cannot do so given its periodic, coarser-granularity migration.

Both classic caching and tiering, to maximize performance, strive to ensure that most accesses are served from the performance device. Most caching and tiering policies are thus designed to maximize hits to the fast device. In traditional hierarchies where the performance of D_{hi} is significantly higher than D_{lo}, such approaches deliver high performance. However, with the storage landscape rapidly changing, modern devices have overlapping performance characteristics and thus it is imperative to rethink how such devices must be managed.

Example	Latency	Read (GB/s)	Write (GB/s)	Cost ($/GB)
DRAM	80ns	15	15	~7
NVDIMM	300ns	6.8	2.3	~5
Low-latency SSD	10us	2.5	2.3	1
NVMe Flash SSD	80us	~3.0	~2.0	0.3
SATA Flash SSD	180us	0.5	0.5	0.15

Table 1: **Diversified Storage Devices.** *Data taken from SK Hynix DRAM(DDR4, 16GB), Intel Optane DCPM [6, 7], low-latency SSDs (Optane SSD 905P [8], Micron X100 SSD [13]), NVMe Flash SSD (Samsung 970 Pro [14, 15]) and SATA Flash SSD (Intel 520 SSD [9]). Low-latency SSD and NVMe Flash SSD assume PCIe 3.0.*

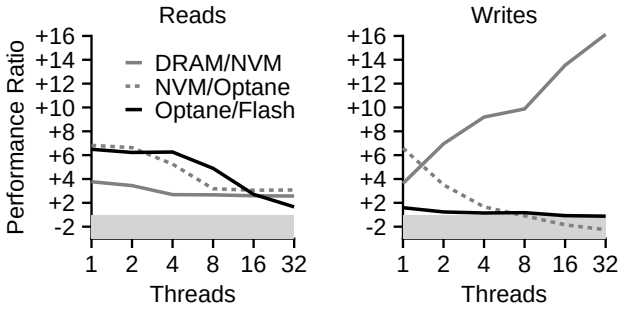


Figure 2: **Performance Ratios Across Modern Devices.** *The ratio of throughput, for varying concurrency, across device pairings. We disable the cache prefetcher and use non-temporal stores for DRAM and NVM. NVM is used as App-Direct mode. Note there is no value between -1 and +1.*

2.2 Hardware Storage Trends

As shown in Table 1, storage systems are undergoing a rapid transformation with a proliferation of high-performance technologies, including persistent memory (e.g., 3D XPoint memory [1, 42]), low-latency SSDs (e.g., Intel Optane SSD [8], Samsung Z-SSD [16], and Micron X100 SSD [13]), NVMe Flash SSDs ([14, 15]), and SATA Flash SSDs ([9]). Although a seeming ordering exists in terms of latency, bandwidth differences are less clear, and a total ordering is hard to establish.

To better understand the performance overlap of these devices, Figure 2 shows the throughput of a variety of real devices for both 4KB read/load and write/store while varying the level of concurrency. The figure plots the performance ratio between pairs of devices: DRAM/NVM plots the bandwidth of memory (SK Hynix 16GB DDR4) vs. a single Intel Optane DCPM (128GB); NVM/Optane uses the DCPM vs. the Intel 905P Optane SSD; finally, Optane/Flash uses the same Optane SSD and the Samsung 970 Pro Flash SSD. For any pair X/Y, a positive ratio ($\frac{X}{Y}$) is plotted if the performance of X is greater than Y; otherwise, a negative ratio ($\frac{-Y}{X}$) is plotted (in the gray region).

For reads with low concurrency, one can see significant differences between device pairs. Thus, one might conclude that a total ordering exists. However, for reads under high concurrency, the ratios change dramatically. In the most extreme case, the Optane SSD and Flash SSD have nearly identical performance. For writes, the results are even more intriguing; because of the low performance of NVM concurrent writes, in one case (NVM/Optane), the ratio changes from much better

under low load to much worse under high load.

To summarize, the following are the key trends in the storage hierarchy. Unlike traditional hierarchy (e.g., DRAM vs. HDD), the new storage hierarchy may not be a hierarchy; performance of two neighboring layers (e.g., NVM vs. Optane SSD) can be similar. Second, performance of new devices vary depending upon many factors including different workloads (reads vs. writes) and level of parallelism. Managing these devices, which exhibit a wide range of performance characteristics, with traditional caching and tiering is no longer effective. Next, we will study the costs of doing so.

3 Caching and Tiering

We now explore caching and tiering in different storage hierarchies. Our goal is simple: to understand how these approaches perform in both traditional and modern hierarchies. In doing so, we hope to build towards a technique that addresses the limitation of these approaches and thus can deliver maximal performance when running on modern, complex devices and underneath a range of dynamic workloads.

For a deeper intuition, we first model caching and tiering performance. We then perform an empirical analysis on real devices, filling in important details not captured by the model.

3.1 Modeling

We develop a simple model of caching and tiering performance. We assume there are two devices, D_{hi} and D_{lo}, where each performs at a fixed rate, R_{hi} and R_{lo} ops/s; of course, real devices are more complex, with internal concurrency and performance that depends on the workload, but this simplification is sufficient for our purposes.

We also assume that the workload has either little concurrency (i.e., one request at a time) or copious concurrency (i.e., many requests at a time). This allows us to bound the performance of caching and tiering between these extremes.

For caching, we assume that the workload is read only; this simplifies our analysis in that we do not account for dirty writebacks upon a cache replacement. For tiering, we assume no background migration takes place; this removes the cost of rearrangement across tiers over time.

3.1.1 Caching

We develop a model of caching performance based on hit rate, H, which varies from 0 to 1. As stated above, we model two extremes: low and high concurrency. For one request at a time, the average time per request is:

$$T_{cache,1} = H \cdot T_{hit} + (1-H) \cdot T_{miss} \tag{1}$$

T_{hit} is simply the inverse of the rate of the fast device, i.e., $T_{hit} = \frac{1}{R_{hi}}$; T_{miss} is the cost of fetching the data from the slow device and also installing it in the faster device, i.e., $T_{miss} = \frac{1}{R_{hi}} + \frac{1}{R_{lo}}$, or $\frac{R_{hi}+R_{lo}}{R_{hi} \cdot R_{lo}}$.

The resulting bandwidth is the inverse of $T_{cache,1}$:

$$B_{cache,1} = \frac{R_{hi} \cdot R_{lo}}{H \cdot R_{lo} + (1-H) \cdot (R_{hi} + R_{lo})} \tag{2}$$

We now model concurrent workloads. Assume N requests. $H \cdot N$ are hits, $(1-H) \cdot N$ are misses. Note that only misses are serviced by the slow device, whereas *all* requests must be serviced by the fast one (data admissions). The time to process N requests on the slow or fast device is:

$$T_{slow}(N) = N \cdot (1-H) \cdot \frac{1}{R_{lo}} \tag{3}$$

$$T_{fast}(N) = N \cdot (1-H) \cdot \frac{1}{R_{hi}} + N \cdot H \cdot \frac{1}{R_{hi}} = N \cdot \frac{1}{R_{hi}} \tag{4}$$

Total time is the maximum of these two, i.e., whichever device finishes last determines the workload time.

$$T_{cache,many}(N) = max(T_{slow}(N), T_{fast}(N)) \tag{5}$$

$$= max(N \cdot \tfrac{1-H}{R_{lo}}, N \cdot \tfrac{1}{R_{hi}}) \tag{6}$$

Dividing by N (not shown) yields the average time per request. Finally, the bandwidth can be computed, as it is the inverse of the average time per request:

$$B_{cache,many} = \frac{1}{max(\frac{1-H}{R_{lo}}, \frac{1}{R_{hi}})} \tag{7}$$

3.1.2 Tiering

We model tiering based on the split rate, $S \in [0,1]$, which determines the fraction S of requests serviced from D_{hi}; the remaining requests $(1-S)$ are served from the tier D_{lo}.

For low concurrency, the total time to serve the workload of N requests is the *sum* of the time to serve the fraction S of requests to D_{hi} and to serve the fraction $1-S$ sent to D_{lo}.

$$T_{tier,1}(N) = N \cdot S \cdot \frac{1}{R_{hi}} + N \cdot (1-S) \cdot \frac{1}{R_{lo}} \tag{8}$$

The average time is found by dividing by N, and the resulting bandwidth by inverting that average time:

$$B_{tier,1} = \frac{1}{\frac{S}{R_{hi}} + \frac{1-S}{R_{lo}}} \tag{9}$$

For high concurrency, both devices are used at the same time, and thus the total time to service N requests is the *maximum* time spent waiting for each tier to process its fraction of requests, as determined by the split ratio.

$$T_{tier,many}(N) = \max(N \cdot S \cdot \frac{1}{R_{hi}}, N \cdot (1-S) \cdot \frac{1}{R_{lo}}) \tag{10}$$

Dividing by N yields the average time per request, and inverting that gives the bandwidth of tiering for a concurrent workload.

$$B_{tier,many} = \frac{1}{\max(\frac{S}{R_{hi}}, \frac{1-S}{R_{lo}})} \tag{11}$$

3.1.3 Model Exploration

We investigate different parameter settings with our model. Figure 3 shows the results for four different settings, starting with a large difference in performance between D_{hi} and D_{lo}, and then slowly increasing the performance of D_{lo}. In each figure, we plot the total throughput (B) achieved for caching (orange or gray, thick lines) and tiering (black, thin lines), while varying the hit/split rate. We show modeled performance for high ("many") and low ("1") concurrency.

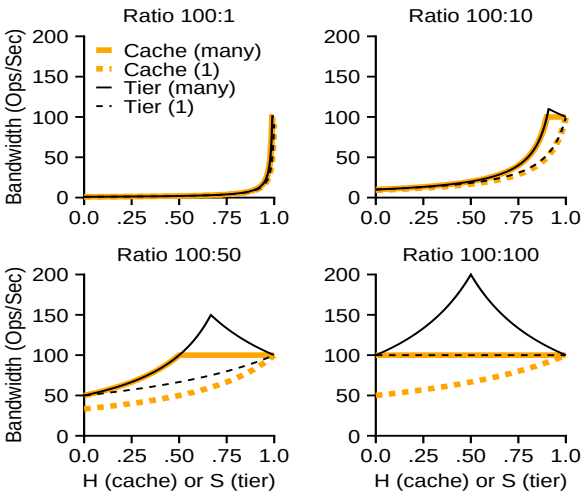


Figure 3: **Modeled Performance.** *Model-predicted performance for caching and tiering across a range of different device performance levels.*

The first graph shows a traditional hierarchy where the performance of D_{hi} is much (100×) higher than the performance of D_{lo}. This graph shows that both caching and tiering can deliver high performance. With caching, the key is a high hit rate. Similarly, tiering performs well if the splitting strategy ensures that nearly all requests are directed to D_{hi}. Even with 80% of requests going to D_{hi}, overall performance of both caching and tiering is quite low, as the slow device dominates.

The next graph (upper right) examines a case where the performance ratio between the devices is still high (10×). Optimizing for high hit/split rate still works well. Note the slight difference between the low and high concurrency cases; with higher concurrency, these approaches can achieve peak performance even with slightly less than a perfect hit rate, as outstanding requests hide the cost of misses.

The next two graphs represent modern hierarchies where the performance of D_{hi} is closer to that of D_{lo} (D_{hi} delivers bandwidth either 2× D_{lo} or equal to it). We make two important observations from these graphs.

First, maximizing the amount of requests served by D_{hi} does not always yield the best performance with tiering. Consider the case where D_{hi} is 2× faster than D_{lo}. With copious concurrency, when about two-thirds of the requests hit D_{hi} (and the remaining to D_{lo}), tiering can realize the aggregate bandwidth of D_{lo} and D_{hi}. Increasing the split rate further only degrades performance. Most tiering systems, however, try to maximize the requests to D_{hi} and thus cannot realize the combined performance of D_{hi} and D_{lo}. Second, classic caching is limited by the performance of D_{hi} and cannot realize the combined performance of both the devices. Even with a 100% hit ratio, caching can only deliver the maximum bandwidth of the faster device D_{hi}, or 100 ops/sec because it does not utilize the bandwidth of the capacity device D_{lo}.

From these results, we observe that when the performance of the two devices is significantly better than the performance

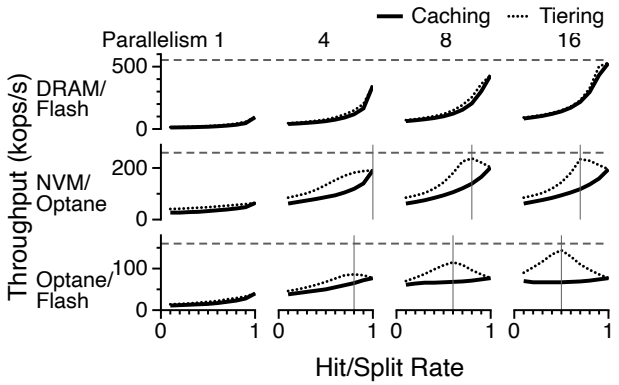


Figure 4: **Performance of Caching and Tiering.** *This figure shows the throughput of read-only workloads. Horizontal dotted lines represent the combined bandwidth of both devices (the maximum possible throughput).*

of the faster device (i.e., $R_{hi} + R_{lo} \gg R_{hi}$), one should utilize the bandwidth of both layers, not just the performant one. What was formerly considered the capacity layer, and its possible performance contributions, should not be ignored.

3.2 Evaluation with DCPM and Optane SSD

Our model indicated that while caching and tiering can obtain excellent performance on traditional hierarchies by optimizing hit/split rates, they cannot do so on modern storage stacks. We now show that these observations hold for real storage stacks. We use one traditional hierarchy consisting of DRAM and a Flash SSD [14], and two modern stacks: first, NVM (Optane DCPM 128GB) + an Optane 905P SSD , second, an Optane SSD + a Flash SSD. We use these hierarchies to cover a wide range of performance differences; meantime, DCPM and Optane SSD are the most popular emerging devices nowadays. While there could be many hierarchies (e.g., with different versions of these devices), we believe our hierarchies are adequate to validate our modeling and draw meaningful implications for our designs.

For these experiments, we have implemented a new benchmarking tool, called the *Hierarchical Flexible I/O Benchmark Suite (HFIO)*. HFIO contains a configurable hierarchy controller that implements caching and tiering. For caching, HFIO uses an LRU-replacement policy. With tiering, the split of a workload across the devices is static, not changing over an experiment. HFIO generates synthetic workloads with a variety of parameters (e.g., mix of reads and writes, locality distribution, and the number of concurrent accesses). HFIO precisely controls the caching layer size and access locality to obtain a desired hit rate. We fix the block size to 32 KB and consider only random accesses. We run our experiments on an Intel Xeon Gold 5218 CPU at 2.3GHz (16 cores), running Ubuntu 18.04. All experiments ran long enough to fill the cache and deliver steady-state performance.

We begin by replicating the results from our model by running read-only workloads and measuring the throughput. Figure 4 shows the results on three hierarchies and workloads with different levels of parallelism. First, in the traditional hier-

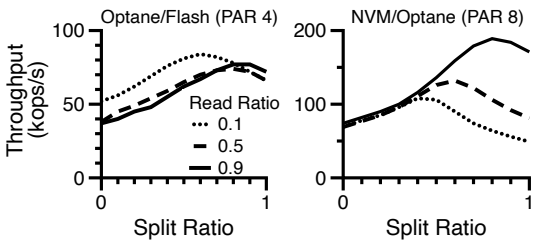
archy with DRAM as the performance layer and a Flash SSD as the capacity layer (first row of Figure 4), as expected, both caching and tiering can achieve high performance. Caching and tiering perform similarly irrespective of parallelism; this is exactly what our model predicted in Figure 3 (100:1 or 100:10 cases). The key, of both caching and tiering on traditional hierarchy, is a high hit/split rate.

The second two rows of Figure 4 show that caching and tiering in new storage hierarchies (e.g., NVM+Optane, Optane+Flash) behave much differently than in the traditional hierarchy. First, with low parallelism (1 or 4), the caching device is not fully utilized, and thus maximizing the hit/split rate still improves performance.

However, for workloads with more parallelism, simply maximizing the hit/split rates does not lead to peak performance. With high parallelism (16), tiering on NVM could achieve almost the aggregate bandwidth of both devices with an optimal split of requests (shown as a vertical line in the figure). However, maximizing the split rate further does not effectively utilize the bandwidth from the capacity device, thereby reducing performance. Similar trends can be observed with the Optane+Flash hierarchy. Higher hit rates on modern hierarchies do not improve caching performance because caching never offloads requests to the capacity layer. Therefore, neither caching nor tiering in their current form (which try to maximize hit/split rate) is suitable for modern hierarchies.

A hierarchy management approach can realize the aggregate performance of both devices *when requests are split optimally*. However, such an optimal split is challenging to achieve. First, the best split rate varies across devices and with parallelism. As shown in Figure 4, Optane and Flash (with high parallelism) have similar performance and thus the best split on Optane+Flash is around 50%; NVM+Optane differ more in performance and thus the optimal rate is higher (about 70%). Optimal split rate also varies significantly with parallelism. For example, with Optane+Flash tiering, the best split for low parallelism (4) is 80%, while it is 50% for parallelism 16. Write ratios also influence the best split rate. As shown in Figure 5, for Optane+Flash, the best split rate for a read-heavy workload is 90%, while it is about 60% when the workload is write-heavy. This change occurs because the difference between the write performances of Optane and Flash is smaller than the difference between their read performances. Similar observations are also made for NVM+Optane hierarchy.

As we noted earlier, tiering (unlike caching) cannot dynamically adapt to changing workloads and parallelism because it migrates data between devices in longer time scales and at coarser granularities. Given that workload and parallelism affect the optimal split rate, it is hard for tiering to adjust the split rate quickly and thus deliver high performance. Further, tiering may have to migrate a lot of data across devices to match the current optimal split rate which could adversely affect performance. Caching, on the other hand, does not need to migrate data to the capacity layer before requests can be



(a) Optane SSD + Flash (b) NVM + Optane SSD

Figure 5: **Mixed Workloads.** *The figure shows the performance of tiering with read-write workloads; PAR: parallelism.*

redirected to the capacity layer. However, classic caching approaches do not utilize the bandwidth of the capacity device.

Given this, we note a new opportunity to improve caching on modern hierarchies. Our main realization is that peak performance can be obtained when the bandwidth of both caching and capacity layers are utilized. The main idea to do so is to offload the excess load at the caching layer to the capacity layer when doing so improves the overall performance.

4 Multi-Factor Caching

We present *multi-factor caching* (MFC), a caching framework that utilizes the performance of devices that would have been treated as only a capacity layer with classic caching. MFC has the following **goals**:

(i) Perform as well or better than classic caching. Classic caching optimizes the performance of a storage hierarchy by optimizing the performance from the higher-level device, D_{hi}; this performance is optimized by finding the working set that maximizes the hit rate. MFC should degenerate in the worst-case to classic caching and should be able to leverage any classic caching policy (e.g., eviction and write-allocation).

(ii) Require no special knowledge or configuration. MFC should not make more assumptions than classic caching. MFC should not require prior knowledge of the workload or detailed performance characteristics of the devices. MFC should be able to manage any storage hierarchy.

(iii) Be robust to dynamic workloads: Workloads change over time, in their amount of load and working set. MFC should adjust to dynamic changes.

The main idea of MFC (Figure 1) is to offload excess load to capacity devices when doing so improves the overall caching performance. MFC can be described in three steps. First, when warming up the system (or after a significant workload change), MFC leverages classic caching to identify the current working set and load that data into the D_{hi}; this ensures that MFC performs at least as well as classic caching. Second, after the hit rate has stabilized, MFC improves upon classic caching by sending excess load to the capacity device, D_{lo}. This excess load has two important components: read hits that are not delivering additional performance on D_{hi} because D_{hi} is already at its maximum performance, and read misses that cause unnecessary data movement between the two de-

vices. Classic caching moves data from D_{lo} to D_{hi} when a miss occurs to improve the hit rate. However, improving hit rate is not beneficial when D_{hi} is already delivering its maximum performance. Therefore, MFC decreases the amount of data admissions into D_{hi} . Finally, if a workload change is observed, multi-factor caching returns to classic caching; if the workload never stabilizes, the algorithm degenerates to classic caching. MFC can leverage the same write-allocation policies as a classic cache (e.g., write-around or write-back).

4.1 Formal Definitions

To describe MFC, we introduce a few terms. We assume that the storage hierarchy is still composed of two devices, D_{hi} and D_{lo}. Caching performance is determined by how the workload is distributed across those two devices. We denote the total workload over a time period δ_t as a constant W, a finite set of accesses to data items. We use w to refer to the subset of W served by D_{hi}, and use its complement set $W-w$ for that served by D_{lo}. We model performance in the time period δ_t as $\mathbf{P}(W,w) = \mathbf{p}_{hi}(w) + \mathbf{p}_{lo}(W-w)$. We make the following assumptions about the devices:

Assumption 1: Performance of a device has an upper bound. The performance of a device cannot increase after it is fully utilized. L_{hi} and L_{lo} represent the maximum possible performance that can be delivered by each device for the current workload, W. We note w_0 as the smallest subset of w such that $\mathbf{p}_{hi}(w_0) = L_{hi}$.

Assumption 2: Increasing the workload on a device does not decrease performance. This implies $\mathbf{p}_{hi}(x)$ and $\mathbf{p}_{lo}(x)$ are monotonically increasing functions. Note that HDD performance can decrease with more random requests due to more seeks, but the assumption generally holds for high-performing devices such as DRAM, NVM, and SSDs.

Assumption 3: Before reaching upper limits, $\mathbf{p}_{hi}(x)$ has a larger gradient than $\mathbf{p}_{lo}(x)$. Based on our observations from real devices, the potential performance gain of using D_{hi} is greater than that of using D_{lo}.

We define classic caching as an approach that optimizes $\mathbf{P}(W,w)$ by maximizing only $\mathbf{p}_{hi}(w)$. Classic caching attempts to maximize $\mathbf{P}(W,w)$ by finding some working set w_{max} that maximizes the hit rate of D_{hi}.

The key insight of MFC is that, when $w_0 < w_{max}$, an opportunity exists to move some portion of the workload $w_{max} - w_0$ away from D_{hi} to D_{lo}. Since $\mathbf{p}_{hi}(w_0) = \mathbf{p}_{hi}(w_{max}) = L_{hi}$, removing $w_{max} - w_0$ from D_{hi} does not decrease the performance of D_{hi} below L_{hi} and now D_{lo} can deliver some amount of performance for $w_{max} - w_0$. Thus, MFC can always perform as well or better than classic caching.

4.2 Architecture

As shown in Figure 6, classic caching can be upgraded to MFC by adding decision points to its cache controller and a *multi-factor cache scheduler*. The classic cache controller serves reads and writes from a user/application to the storage devices (i.e., D_{hi} and D_{lo}) and controls the contents of D_{hi}

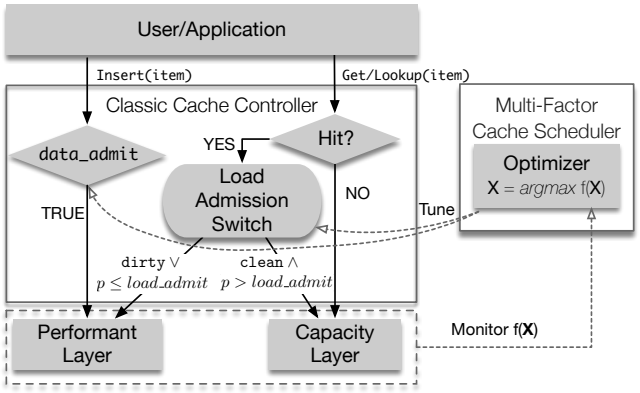


Figure 6: **Multi-Factor Cache Architecture.** *MFC adds decision points and a scheduler to classic caching. As before, MFC is transparent to users. Any classic caching implementation can be upgraded to be a MFC one. Note that decision points only tune cache read hits/misses.*

based on its replacement policy (e.g., LRU). A new *multi-factor cache scheduler* monitors performance and controls whether classic caching is performed and where cache read hits are served. The scheduler optimizes a target function that can be supplied by the end-user (e.g., ops/sec) or use device-level measurements (e.g., request latency).

The MFC scheduler performs this control with a boolean `data_admit (da)` and a variable `load_admit (la)`. The `da` flag controls behavior when a **read miss** occurs on D_{hi}: when `da` is set, missed data items are allocated in D_{hi}, according to the cache replacement policy; when `da` is not set, the miss is handled by D_{lo} and not allocated in D_{hi}. Classic caching corresponds to the case where the `da` flag is true.

The `la` variable controls how **read hits** are handled and designates the percentage of read hits that should be sent to D_{hi}; when `la` is 0, all read hits are sent to D_{lo}. Specifically, for each read hit, a random number $R \in [0, 1.0]$ is generated; if $R <= $ `la`, the request is sent to D_{hi}; else, it is sent to D_{lo}. In classic caching, `la` is always 1.

The MFC framework works with any classic caching write-allocation policy, which handles **write hits/misses**. MFC admits write misses into D_{hi} according to the policy; `da`, `la` do not control write hits/misses. With write-back, cache writes introduce dirty data in D_{hi} and data on D_{lo} can be out-of-date; in this case, MFC does not send dirty reads to D_{lo}.

4.3 Cache Scheduler Algorithm

The MFC scheduler adjusts the behavior of the cache controller to optimize a target function. As shown in Algorithm 1, the scheduler has two states: increasing the amount of data cached on D_{hi} to maximize hit rate, or keeping the data cached constant, while adjusting the load sent to each device.

State 1: Improve hit rate. The MFC scheduler begins by letting the cache controller perform classic caching with its default replacement policy (`da` is true and `la` is 1); during this process, the cache is warmed up and the hit rate improves as the working set is cached in D_{hi}. The MFC scheduler monitors

Algorithm 1: Multi-factor cache scheduler

```
cache: classic cache controller
step: the adjustment step size for load_admit
f(x): target function value when load_admit = x, the value is
  measured by setting load_admit = x for a time interval
1  while true do
       # State 1: Improve hit rate
2      data_admit = true,  load_admit = 1.0
3      while cache.hit_rate is not stable do
4          sleep_a_while()
5      data_admit = false,  start_hit_rate = cache.hit_rate
       # State 2: Adjust load_admit
6      while true do
7          ratio = load_admit
           # Measure f(ratio-step) and f(ratio+step)
8          max_f = Max(f(ratio-step), f(ratio), f(ratio+step))
           # Modify load_admit based on the slope
9          if f(ratio-step) == max_f then
10             load_admit = ratio - step
11         else if f(ratio+step) == max_f then
12             load_admit = ratio + step
13         else if f(ratio) == max_f then
14             load_admit = ratio
15             if load_admit == 1.0 then
16                 goto line 2 # Quit tuning if w < w0
           # Check whether workload locality changes
17         if cache.hit_rate < (1-α)start_hit_rate then
18             goto line 2
```

the hit rate of D_{hi} and ends this phase when the hit rate is relatively stable; at this point, the performance delivered by D_{hi} for the workload w_{max} is near its peak.

State 2: Adjust load between devices. After D_{hi} contains the working set leading to its optimal hit rate and performance, the MFC scheduler explores if sending some requests to D_{lo} increases the performance of D_{lo}, while not decreasing the performance of D_{hi}, i.e., the algorithm moves from w_{max} toward w_0. In this state, `da` is set to false and feedback is used to tune `la` to maximize the target function. Specifically, the scheduler (Lines 6–18) modifies `la`; in each iteration, performance is measured with `la +/- step` over a time interval (e.g., 5ms see §5). The value of `la` is adjusted in the direction indicated by the three data points. When the current value of `la` leads to the best performance, the scheduler sticks with the current value. The value of `la` is kept in the acceptable domain of [0, 1.0] with a negative penalty function. If the scheduler finds the optimal `la` is 1, it quits tuning and moves back to State 1; intuitively, this means MFC has moved the current workload w below w_0 and hence requires classic caching to improve the hit rate to further improve performance.

Since MFC relies on classic caching to achieve an acceptable hit rate, it restarts the optimization process when workload locality changes. The MFC scheduler monitors the cache hit rate at runtime; if the current hit rate drops, the scheduler re-enters State 1 to reconfigure the cache with the current working set. If the workload never stabilizes, MFC behaves like classic caching.

Performance Target Functions: MFC can leverage different target functions. The target function can optimize metrics such as throughput, latency, tail latency, or any combination. The target function can also capture performance at various levels of the system (e.g., hardware, OS, or application).

Write Operations: MFC handle writes with the write-allocation policy (specified by users) in the classic cache controller. MFC does not adjust the write-allocation policy because it may be chosen for factors other than performance: endurance [35, 84], persistence, or consistency [57].

Summary: Multi-factor caching optimizes classic caching to effectively use the performance of capacity devices. MFC improves on classic caching in two ways. First, MFC does not admit read misses into D_{hi} when the performance of D_{hi} is fully utilized. Second, when the performance of D_{hi} is at its peak, MFC delivers useful performance from the D_{lo} device by sending some of the requests that would have hit in D_{hi} to D_{lo} instead. By determining at run-time the appropriate load, MFC obtains useful performance from D_{lo} instead of using D_{lo} only to serve misses into D_{hi}.

5 Implementation

We implement multi-factor caching in two places: Orthus-CAS, a generic block-layer caching kernel module, and Orthus-KV, a user-level caching layer for a key-value store.

Open CAS [31] is caching software built by Intel that accelerates accesses to a slow backend block device by using a faster device as a cache. It supports different write-allocation policies such as write-around, write-through, and write-back, and uses an approximate LRU policy for eviction. Open CAS is a kernel module that we modify to leverage MFC. Orthus-CAS works with all policies supported in Open CAS.

We also implement MFC within a persistent block cache for Wiskey [62], an LSM-tree key-value store. LSM trees are a good match for Optane SSD, and have garnered significant industry interest [2, 12, 36]. Wiskey is derived from LevelDB; the primary difference is that Wiskey separates keys from values to reduce amplification. While keys remain in the LSM-tree, values are stored in a log and each key points to its corresponding value in the log. Separating keys and values also improves caching because it avoids invalidating values when compacting levels; this is similar to the approach in memcached [11] for spilling data to SSD. We integrate MFC with Wiskey's persistent block cache layer. The cache keeps hot blocks (both LSM-tree key and value blocks, 4KB in size) on the cache device using sharded-LRU. Eviction occurs in units of 64 blocks. We call this implementation Orthus-KV.

Target Functions: Our implementations support three different target functions: throughput, average latency, and tail (P99) latency, with throughput being the default. When optimizing

throughput, we use the Linux block-layer statistics [10] to track device throughputs. When optimizing for latency, we track the end-to-end request latency of the caching system.

Tuning Parameters: MFC implementations must measure the target metrics and tune parameters periodically. The speed at which MFC adapts to workload changes depends on both the interval between target function measurements and the step size. With a smaller interval, tuning converges faster. Though frequent tuning means more CPU overheads, our CPU overheads are negligible. We found the Linux block layer counters [10] are not accurate when the interval is smaller than 5 ms, so we use the smallest yet accurate interval of 5 ms. A large step size leads to faster convergence but may get sub-optimal results. MFC adjusts the load ratio by 2% in each step; we have found this gives a reasonable converging time with end results similar to smaller step sizes. We leave an adaptive step size for future work.

Implementation Overhead: We find that implementing MFC into existing caching layers is fairly straightforward and requires nominal developer effort. We added only 460 (not including cache mode registration code) and 228 LOC into Open CAS and Wiskey, respectively.

6 Evaluation

Our evaluation aims to answer the following questions:

- How does MFC in Orthus-CAS perform across hierarchies, write-allocation policies, and target functions? (§6.1)
- How does MFC as implemented in Orthus-KV perform on static workloads? (§6.2)
- How does Orthus-KV handle dynamic workloads? How does it adapt to changes in load and data locality? (§6.3)
- How does MFC compare to previous work? (§6.4)

Setup. We use the following real devices: a SK Hynix DDR4 module (denoted as DRAM), an Intel Optane 128GB DCPM (NVM), an Intel Optane SSD 905P (Optane), and a Samsung 970 Flash SSD (Flash). We also use FlashSim [56] to simulate flash devices with different performance characteristics.

6.1 Orthus-CAS

We begin by evaluating MFC as implemented in Orthus-CAS running on microbenchmarks where the workloads do not change over time. The accesses are uniformly random and 64KB (the suggested page size for Open CAS). We use 1GB of the cache device and generate workloads with different hit ratios. We report the stable performance of classic caching; for MFC, we report when its tuning stabilizes. Unless otherwise noted, we use throughput as the target function.

6.1.1 Storage Hierarchies

We show the normalized throughput of Open CAS and Orthus-CAS for read-only workloads with different hierarchies, amounts of load, and hit ratios in Figure 7. We define Load-1.0 as the minimum read load to achieve the maximum read bandwidth of the cache device; we generate Load-0.5, 1.5, and 2.0 by scaling load-1.0. We investigate hierarchies that include DRAM, NVM, Optane SSD, and Flash. We also mimic hierarchies with two performance differences (50:10 and 50:25) using FlashSim; we configure FlashSim to simulate devices with maximum speeds of 50MB/s, 25MB/s, and 10MB/s. We make the following observations from the figure:

First, when load is light (e.g., Load-0.5), cache devices always outperform capacity devices. In this case, MFC does not bypass any load and behaves the same as classic caching.

Second, when the workload can fully utilize the cache device, Orthus-CAS improves performance. Intuitively, a higher hit ratio and load gives MFC more opportunities bypass requests and improve performance. Figure 7 confirms the intuition: with 95% hit ratio and Load-2.0, MFC obtains improvements of 21%, 32%, 54% for DRAM+NVM, NVM+Optane, and Optane+Flash, respectively. Such improvements are marginally reduced with an 80% hit ratio.

Third, among these hierarchies, Optane+Flash improves the most with Orthus-CAS since the performance difference between Optane and Flash is the smallest, followed by NVM+Optane and DRAM+NVM. Our results with FlashSim show how practitioners can predict the improvement of using MFC on their target hierarchies.

Finally, our measurements indicate that Orthus-CAS adapts to complex device characteristics. With an 80% hit ratio, classic caching does not achieve 1.0 normalized throughput on any real hierarchy because cache misses introduce additional writes to the cache device. MFC handles such complexities.

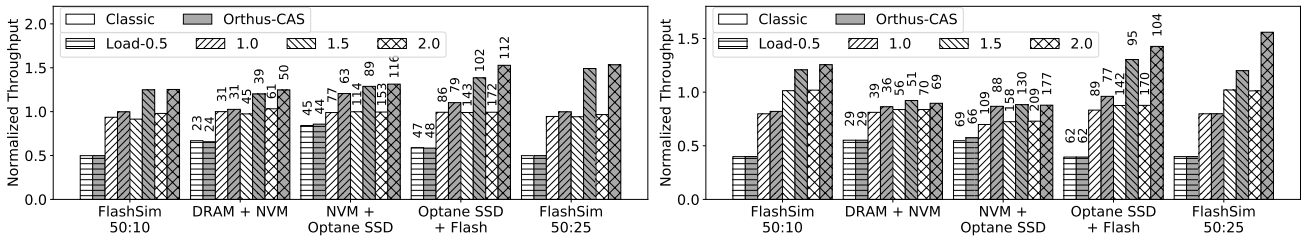
Latency Improvement. As shown in Figure 7, Orthus-CAS also improves average latency on all hierarchies. For instance, with Load-2.0, MFC reduces average latency by 19%, 25%, 39%, for DRAM+NVM, NVM+Optane, and Optane+Flash hierarchies, respectively.

6.1.2 Write-Allocation Policies

Open CAS supports many write-allocation policies (write-around, write-back, and write-through). Our experiments show that MFC improves performance relative to classic caching with each policy (we do not show the results due to space constraints). The improvements are dependent upon a combination of the workload write and dirty-read ratios (how many read hits will be dirty) and the write-allocation policy. MFC offers more benefits when there are fewer writes. With write-back, MFC cannot offload reads of dirty items to the capacity device and thus performs much better with fewer dirty reads. Write-around and write-through maintain consistent copies and thus Orthus-CAS offers benefits independent of the dirty-read ratio.

6.1.3 Target Functions

MFC can improve different performance metrics by optimizing different target functions. Table 2 shows the performance of Open CAS and Orthus-CAS when using throughput, average latency, and tail (P99) latency as target functions. Optimizing throughput or average latency yields similar improvements to both metrics on both hierarchies, but increases tail

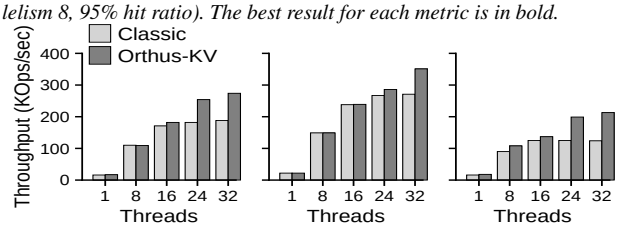


(a) 95% Hit Rate (b) 80% Hit Rate

Figure 7: **Orthus-CAS on Various Hierarchies.** *Read-only workloads; (a) and (b) show different cache hit rates. All throughputs are normalized to the maximum read bandwidth of the caching device. We show latency (µs) on top of each bar (not comparable across hierarchies).*

Target Func	NVM + Optane			Optane + Flash		
	Throughput GB/s	Avg. lat. µs	P99 lat. µs	Throughput GB/s	Avg. lat. µs	P99 lat. µs
Open CAS	6.7	77	115	2.3	227	269
Throughput	**8.0**	**64**	147	**3.9**	**132**	289
Avg. lat.	**8.0**	**64**	143	**3.9**	**132**	285
P99 lat.	6.9	75	**106**	3.3	155	**245**

Table 2: **Different Target Functions.** *Read-only workloads (parallelism 8, 95% hit ratio). The best result for each metric is in bold.*



(a) V:1KB theta:0.6 (b) V:1KB theta:0.8 (c) V:16KB theta:0.8

Figure 8: **Orthus-KV, YCSB-C Performance.** *We use 16B keys and 1KB or 16KB values. Accesses follow a Zipfian distribution (theta).*

latency. This increase occurs because in each of these storage hierarchies, the performance device has much better tail latency than the capacity device; thus classic caching defaults to appropriate behavior. When MFC is configured with P99 latency as the target function, Orthus-CAS has better tail latency than Open CAS and than it does with other target functions.

6.2 Orthus-KV: Static Workloads

We use Orthus-KV, the MFC implementation in Wisckey, to show the benefits for real applications. Caching in Wisckey uses write-around, due to the LSM-tree's log-structured writes. In these experiments we focus on Optane+Flash since it is often used for key-value stores [12, 36]. We set the caching layer to 33 GB, 1/3 of the 100 GB dataset. We use `cgroup` to limit the OS page cache to 1 GB to focus on caching in the storage system instead of caching in main memory.

Our initial evaluation uses the YCSB workloads [33]. Most YCSB workloads are constant: their load does not change and they have a stable key popularity distribution (i.e., Zipfian); we evaluate YCSB-D as a dynamic workload (§6.3).

Gets: Figure 8 compares the throughput of Wisckey and Orthus-KV for three YCSB-C workloads and different amounts of parallelism. Orthus-KV achieves equivalent or higher throughput than Wisckey for all workloads. Orthus-KV significantly improves throughput at high load levels: with 32 threads, 46%, 30%, and 71% higher throughput for the

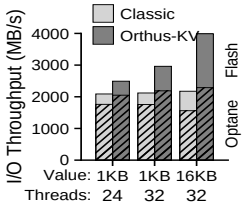


Figure 9: **Bandwidth Breakdown.** *Optane/Flash read bandwidth breakdown during YCSB-C.*

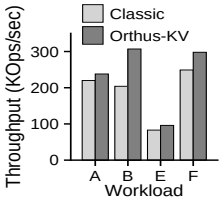


Figure 10: **Other YCSB Workloads.** *16B keys, 1KB values, 32 threads and theta = 0.6.*

three workloads. When load is high enough to saturate Optane, the relative benefits of Orthus-KV depend on how much it can avoid unnecessary data movement and perform better load distribution. Figure 9 illustrates these two benefits with the read bandwidth delivered by each device. First, classic caching suffers from unnecessary data admissions into Optane: its effective Optane read bandwidth never reaches the peak (2.3GB/s). MFC avoids this wasteful data movement. Second, classic caching never delivers more than the maximum Optane bandwidth to the application. In contrast, MFC improves the performance out of the hierarchy by distributing some cache hits to the Flash SSD.

Updates, Inserts, and Range Queries: Figure 10 shows Orthus-KV handles a range of operations (gets, updates, inserts, and range queries) and always performs at least as well as Wisckey. MFC improves all YCSB workloads, with greater benefits with more get operations.

Latency Improvement: With throughput as its target, Orthus-KV reduces YCSB average latency by up to 42%. For YCSB-C (32 threads, 0.8 theta), Orthus-KV provides 30% and 38% lower latency for 1KB and 16KB values, respectively.

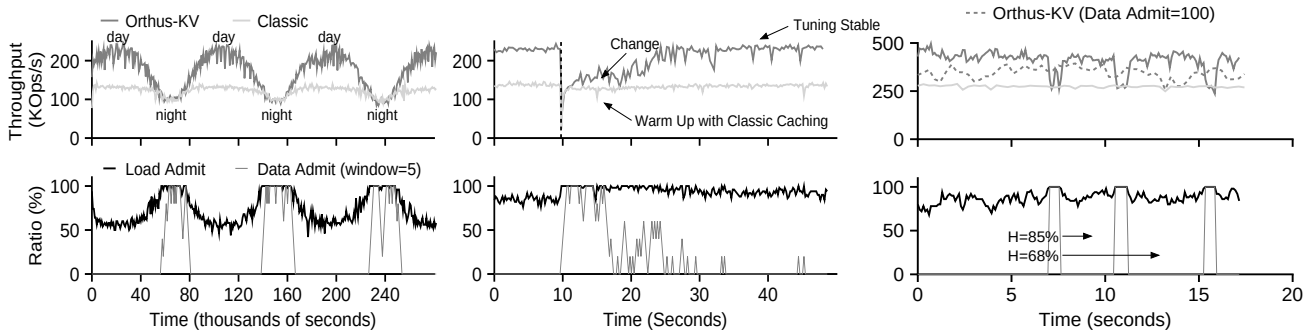
CPU Overhead: Orthus-KV increases CPU utilization slightly (0-2% for 24 threads) due to a few additional counters that track caching behavior and device performance over time.

6.3 Orthus-KV: Dynamic Workloads

We evaluate MFC for dynamic workloads using the same experimental setting as §6.2. We explore how Orthus-KV handles time-varying workloads and dynamic working sets.

6.3.1 Dynamic Load

We evaluate how well MFC handles load changes with the Facebook ZippyDB benchmark [30]. ZippyDB is a distributed key-value store built on RocksDB and used by Facebook. The



(a) ZippyDB Workloads (b) Sudden Change in Data Locality (c) Gradual Changes in Data Locality

Figure 11: **Orthus-KV with Dynamic Workloads.** *This figure shows the throughput of Orthus-KV and classic caching (top graphs), as well as load/data admit ratio over time in Orthus-KV (bottom graphs). Because data admit is 0 or 1, we show a fractional windowed sum of its value over 5 time steps for readability. In (a), we replay the ZippyDB benchmark on a single machine. The average value size is 16KB; the number of key ranges is 6. We use 32 threads for the maximum load and adjust the number of threads dynamically according to the diurnal load model. We speed up the two-day workload by 1000×. In (b), the workload is similar to YCSB-C 16KB value, 32 threads, but with two different working sets before and after 10s. In (c), we use YCSB-D with 16B keys, 1KB values, 32 threads. We also show throughput of a modified Orthus-KV that always performs data admit in (c).*

ZippyDB benchmark generates key-value operations according to realistic trace statistics: 85% gets, 14% puts, 1% scans following a hot range-based model; the request rate models the diurnal load sent to ZippyDB servers. We speed up the replay of Zippydb requests by 1000 to stress the storage system and to better evaluate MFC's reactions to changes in load.

As shown in Figure 11a (top), Orthus-KV outperforms Wisckey during the day by up to 100%, but performs similarly when the load is lower at night. Figure 11a (bottom) shows how Orthus-KV adjusts data and load admit ratios. During the night, both are around 100%; Orthus-KV occasionally adjusts the load admit ratio when the hit rate is stable, but quickly returns to classic caching after finding no improvements. During the day, Orthus-KV keeps the data admit ratio at 0 and adjusts the load admit ratio to adapt to the dynamic load.

6.3.2 Dynamic Data Locality

We demonstrate that MFC reacts well to abrupt changes in the working set in Figure 11b. The experiments base on YCSB-C, beginning with one working set (Zipfian theta=0.8, hot spot at beginning of the key space), and then changing at time 10s (a hot spot at the end of the key space). The graph shows that when the working set changes (time=10s), Orthus-KV quickly detects the change in hit rate and switches to classic caching: the load and data admit ratios increase to 1.0. After the hit rate begins to stabilize (time=11s), Orthus-KV tunes the load admit ratio. Initially (11s-28s), because the hit rate is still not high enough,Orthus-KV often identifies 1.0 as the best load admit and returns to classic caching with data movement. Approximately 20s after the workload change, the hit rate stabilizes and Orthus-KV reaches steady-state performance that is 60% higher than classic caching.

Finally, we show that MFC can outperform classic caching even when the working set changes gradually. Figure 11c shows Orthus-KV's performance on YCSB-D (95% reads, 5% inserts), where locality shifts over time as reads are performed on recently-inserted values. Due to the locality changes and not admitting new data to the cache, the hit rate in Orthus-KV decreases over time, until MFC identifies that 1.0 is the best load admit rate. Then Orthus-KV returns to classic caching and raises the hit rate. Once the hit rate restabilizes, the cycle resumes with Orthus-KV adjusting the load admit rate.

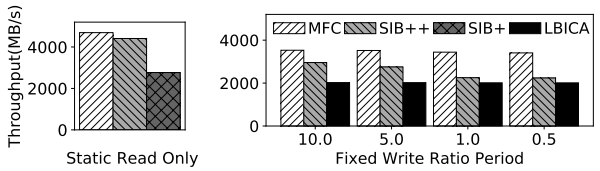
We also explore the alternative approach of always performing data admission while tuning the load admit rate in Figure 11c. As shown, this alternative maintains a stable hit rate, avoiding abrupt phases of admitting new data; this always performs better than classic caching but its peak performance does not reach that of the default Orthus-KV. Our results illustrate the interesting tradeoff between avoiding unnecessary data movement and maintaining a stable hit rate for dynamically changing workloads.

6.4 Comparisons with Prior Approaches

We now show that MFC significantly outperforms two other approaches that have been suggested for utilizing the performance of a capacity device: SIB [54] and LBICA [18]. SIB targets HDFS clusters with many SSDs and HDDs, in which case the aggregate HDD throughput is non-trivial: SIB uses SSDs as a write buffer (does not admit any read miss), and proposes using the HDDs for handling extra read traffic. LBICA determines when a performance layer is under "burst load" at which point it will not allocate new data to the performance layer; unlike MFC, LBICA does not redirect any read hits.

To compare MFC against SIB and LBICA, we have implemented these approaches in Open CAS. To make SIB suitable for general-purpose caching environments, we have improved it in two ways. First, SIB operates on a per-process granularity instead of per-request: the traffic from some processes is not allowed to use the SSD cache; we altered SIB so that it adjusts load per-request (SIB+). Second, we modified SIB so that it admits read misses into the cache (SIB++).

Using the experimental setup from §6.1 on Optane+Flash, we start with a read-only workload in Figure 12.a. SIB+ does not perform well because it does not admit read misses into



(a) Static Workload (b) Dynamic Workloads

Figure 12: **MFC vs. SIB and LBICA.** *(a) uses a static read-only workload. (b) uses dynamic workloads; the write-ratio is fixed in each period (e.g., 10s), but changes (randomly between 0% to 50%) across periods. We use workloads with parallelism 16, 95% hits, runtime 100s on Optane+Flash.*

Optane. SIB++ performs better, but suffers when the workload changes as in Figure 12.b; in these workloads, the amount of write traffic is changed every period, for periods between 10 and 0.5 seconds. SIB cannot handle dynamic workloads because SIB has two phases; in its training phase, SIB learns the maximum performance of the caching device for the current workload; in the inference phase, SIB judges whether the caching device is saturated and, if it is, bypasses some processes (requests). As we have shown, the maximum performance of modern devices depends on many workload parameters: read-write ratios, load, and access patterns. Thus, if the workload changes in any way, SIB must either relearn the target maximum performance or use an inaccurate target. In our experiments, as the duration of each phase decreases, the performance of SIB decreases dramatically. Finally, LBICA performs poorly because it does not redirect any read hits to use the capacity device; it simply does not allocate more data to the performance device when it is overloaded.

7 Related Work

Algorithms and Policies in Hybrid Storage Systems: Algorithms and policies for managing traditional hierarchy have been studied extensively [63, 65, 66, 68, 76, 87, 89, 94, 101]. Techniques have been introduced to optimize data allocation [3, 47, 78, 82, 87, 88], address translation [24, 80], identify hot data [4, 49, 51, 69, 71, 73, 77, 86, 89, 93] and perform data migration [25, 32, 38, 41, 68, 85, 89, 97]. Most previous work improves performance by focusing on workload access locality (i.e., single factor caching). In contrast, MFC improves by taking all devices and workloads into account.

Storage Optimization: A long line of pioneering work in storage management [20–22, 87, 89] shows how to trace workloads and optimize storage decisions for improved performance; MFC could fit into such a system, making short-term decisions to handle more dynamic workload changes, leaving longer-term optimization to a higher-level system.

Storage Aware Caching/Tiering: Our paper shares aspects with storage-aware caching/tiering [18, 28, 40, 41, 45, 50, 54, 57, 58, 70, 91, 96], which considers more factors than hit rate. For instance, Oh et al. [70] propose over-provisioning in Flash to avoid the influence of SSD garbage collection. Modern devices like NVDIMM and Optane SSD have distinctive characteristics compared to Flash. We study the implications of these important emerging devices to caching/tiering. Wu et al. [91] study tiering on SSDs and HDDs and recognize a similar problem: SSDs (or faster devices) can be the throughput bottleneck. To mitigate the problem, they proposed to periodically migrate data from SSDs to HDDs when the SSD response time is higher than that of HDDs. This approach is limited in three aspects. First, due to its tiering nature, it cannot react to workload changes quickly, its migration traffic can be significant, and it requires extra metadata to track objects across devices. Second, similar to SIB approach, it estimates workload intensity in a period and then migrates data based on the estimation; it hence struggles with dynamic workloads. Third, it is tuned for a specific hierarchy (SSDs and hard drives). Unlike this approach, MFC focuses on improving caching, adapts its behavior during runtime, can react to complex and dynamic workloads, and works well on a range of modern devices.

Managing NVM-based Devices: Other related work integrates NVM-based devices into the memory-storage hierarchy [17, 26, 42, 43, 46, 61, 67, 81]. This work includes extensive measurements for both Optane SSD [90] and Optane DCPM [48, 95]. New file systems and databases were proposed to manage NVDIMM [74, 92] and low latency SSDs [59, 64, 98]. Many works have evaluated the potential benefits of caching and tiering on NVM. Kim et al. [53] provide a simulation-based measurement of NVM caching with performance numbers from a Micron all-PCM SSD prototype. [39] provides a I/O cache simulator that assists the analysis of caching workloads on new storage hierarchies. Strata [60] and Ziggurat [100] are file systems that tier data across a DRAM, NVM, and SSD hierarchy. Dulloor et al. [34], Arulraj et al. [23] and Zhang et al. [99] proposed NVM-aware data placement strategies for the new storage hierarchy. These strategies optimize data placements in a longer period (e.g., offline or periodically). MFC can work with them, providing further improvement by handling more dynamic workload changes. Finally, there have been many companies utilizing NVM/ Optane SSD as a caching layer [29, 36, 37]. Our paper is the first to analyze general caching and tiering on modern hierarchies through modeling and empirical evaluation. We are also the first to propose a generic solution (MFC) to realize the full performance benefits of such a hierarchy.

8 Conclusion

In this paper, we show how emerging storage devices have strong implications for caching on modern hierarchies. We introduced multi-factor caching, a new approach optimized to extract peak performance from modern devices. MFC is based upon a novel multi-factor cache scheduling algorithm, which accounts for workload and device characteristics to make allocation and access decisions. Through experiments, we showed the benefits of MFC on a wide range of devices, cache configurations, and workloads. We believe MFC can serve as a better foundation to manage storage hierarchies.

References

[1] 3D XPoint. https://en.wikipedia.org/wiki/3D_XPoint.

[2] Accelerate Ceph Clusters with Intel Optane DC SSDs. https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/accelerate-ceph-clusters-with-optane-dc\-ssds-brief.pdf.

[3] Cache (computing). https://en.wikipedia.org/wiki/Cache_(computing).

[4] Cache replacement policies. https://en.wikipedia.org/wiki/Cache_replacement_policies.

[5] Caching vs. Tiering. https://storageswiss.com/2014/01/15/whats-the-difference-between-tiering-and-caching/.

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

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

[8] Intel Optane SSD 905P. https://www.tomshardware.com/reviews/intel-optane-ssd-905p,5600-2.html.

[9] Intel SSD 520 Series. https://ark.intel.com/content/www/us/en/ark/products/series/66202/intel-ssd-520-series.html.

[10] Linux block layer statistics. https://www.kernel.org/doc/Documentation/block/stat.txt.

[11] Memcached Exstore. https://memcached.org/blog/nvm-caching/.

[12] Micron Heterogeneous-Memory Storage Engine. https://www.micron.com/products/advanced-solutions/heterogeneous-memory-storage-engine.

[13] Micron X100 NVMe SSD. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology/x100.

[14] Samsung 970 Pro. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970pro/.

[15] Samsung 980 Pro Flash SSD. https://www.anandtech.com/show/15352/ces-2020\-samsung-980-pro-pcie-40-ssd-makes\-an-appearance.

[16] Samsung Z-NAND SSD. https://www.samsung.com/semiconductor/ssd/z-ssd/.

[17] 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*, pages 971–985. ACM, 2019.

[18] 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.

[19] Ameen Akel, Adrian M Caulfield, Todor I Mollov, Rajesh K Gupta, and Steven Swanson. Onyx: A prototype phase change memory storage array. *HotStorage*, 1:1, 2011.

[20] Guillermo A Alvarez, Elizabeth Borowsky, Susie Go, Theodore H Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems (TOCS)*, 19(4):483–518, 2001.

[21] Eric Anderson, Michael Hobbs, Kim Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[22] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems (TOCS)*, 23(4):337–374, 2005.

[23] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. Multi-tier buffer management and storage system design for non-volatile memory. *arXiv preprint arXiv:1901.10938*, 2019.

[24] Shi Bai, Jie Yin, Gang Tan, Yu-Ping Wang, and Shi-Min Hu. Fdtl: a unified flash memory and hard disk translation layer. *IEEE Transactions on Consumer Electronics*, 57(4):1719–1727, 2011.

[25] Swapnil Bhatia, Elizabeth Varki, and Arif Merchant. Sequential prefetch cache sizing for maximal hit rate. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 89–98. IEEE, 2010.

[26] Timothy Bisson and Scott A Brandt. Flushing policies for nvcache enabled hard disks. In *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pages 299–304. IEEE, 2007.

[27] Randal E Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 281. Prentice Hall Upper Saddle River, 2003.

[28] 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.

[29] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. Polarfs: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[30] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 209–223, 2020.

[31] Open CAS. Open Cache Acceleration Software. `https://open-cas.github.io/`.

[32] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing belady's limitations: In search of flash cache offline optimality. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 379–392, 2016.

[33] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, pages 143–154, Indianapolis, IN, June 2010.

[34] 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*, page 15. ACM, 2016.

[35] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 65–78, 2019.

[36] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with *NVM* in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018.

[37] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using Non-volatile Memory for Storing Deep Learning Models. *arXiv preprint arXiv:1811.05922*, 2018.

[38] Ahmed Elnably, Hui Wang, Ajay Gulati, and Peter J Varman. Efficient qos for multi-tiered storage systems. In *HotStorage*, 2012.

[39] Tyler Estro, Pranav Bhandari, Avani Wildani, and Erez Zadok. Desperately seeking... optimal multi-tier cache configurations. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[40] Brian Forney, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2002.

[41] Jorge Guerra, Himabindu Pucha, Joseph S Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *FAST*, volume 11, pages 20–20, 2011.

[42] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform Storage Performance With 3D XPoint Technology. *Proceedings of the IEEE*, 105(9), 2017.

[43] Theodore R Haining and Darrell DE Long. Management policies for non-volatile write caches. In *1999 IEEE International Performance, Computing and Communications Conference (Cat. No. 99CH36305)*, pages 321–328. IEEE, 1999.

[44] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

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

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

[47] Ilias Iliadis, Jens Jelitto, Yusik Kim, Slavisa Sarafijanovic, and Vinodh Venkatesan. Exaplan: queueing-based data placement and provisioning for large tiered storage systems. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 218–227. IEEE, 2015.

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

[49] Jaeheon Jeong and Michel Dubois. Cost-sensitive cache replacement algorithms. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pages 327–337. IEEE, 2003.

[50] 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.

[51] Shudong Jin and Azer Bestavros. Popularity-aware greedy dual-size web proxy caching algorithms. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 254–261. IEEE, 2000.

[52] Takayuki Kawahara. Scalable Spin-transfer Torque RAM Technology for Normally-off Computing. *IEEE Design & Test of Computers*, 28(1):52–63, 2010.

[53] Hyojun Kim, Sangeetha Seshadri, Clement L Dickey, and Lawrence Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, pages 33–45, 2014.

[54] 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.

[55] Youngjae Kim, Aayush Gupta, Bhuvan Urgaonkar, Piotr Berman, and Anand Sivasubramaniam. HybridStore: A Cost-Efficient, High-Performance Storage System Combining SSDs and HDDs. MASCOTS '11, 2011.

[56] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A Simulator for Nand Flash-based Solid-State Drives. In *Proceedings of the First International Conference on Advances in System Simulation (SIMUL '09)*, Porto, Portugal, September 2009.

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

[58] 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.

[59] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast {NVM} storage with udepot. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 1–15, 2019.

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

[61] Chun-Hao Lai, Jishen Zhao, and Chia-Lin Yang. Leave the cache hierarchy operation as it is: A new persistent memory accelerating approach. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[62] Lanyue Lu and Thanumalayan Sankaranarayana Pillai and Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 133–148, Santa Clara, California, February 2016.

[63] Donghee Lee, Jongmoo Choi, Jun-Hum Kim, Sam H. Noh, Sang Lyul Min, Yookum Cho, and Chong Sang Kim. On The Existence Of A Spectrum Of Policies That Subsumes The Least Recently Used (LRU) And Least Frequently Used (LFU) Policies. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, Atlanta, Georgia, May 1999.

[64] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[65] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[66] Michael Mesnier, Feng Chen, Tian Luo, and Jason B Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 57–70. ACM, 2011.

[67] Sparsh Mittal and Jeffrey S Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2015.

[68] David Montgomery. Extent migration scheduling for multi-tier storage architectures, November 5 2013. US Patent 8,578,107.

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

[70] 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.

[71] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[72] 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)*, pages 297–306, Washington, DC, May 1993.

[73] Dongchul Park and David HC Du. Hot Data Identification for Flash-Based Storage Systems Using Multiple Bloom Filters. In *Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies (MSST '11)*, Denver, Colorado, May 2011.

[74] Sheng Qiu and AL Narasimha Reddy. Nvmfs: A hybrid file system for improving random write in nand-flash ssd. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, 2013.

[75] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[76] Benjamin Reed and Darrell DE Long. Analysis of caching algorithms for distributed file systems. *ACM SIGOPS Operating Systems Review*, 30(3):12–21, 1996.

[77] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '90)*, Boulder, Colorado, April 1990.

[78] Reza Salkhordeh, Hossein Asadi, and Shahriar Ebrahimi. Operating system level data tiering using online workload characterization. *The Journal of Supercomputing*, 71(4):1534–1562, 2015.

[79] Mohit Saxena, Michael M Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 267–280. ACM, 2012.

[80] Andre Schaefer and Matthias Gries. Adaptive address mapping with dynamic runtime memory mapping selection, 2012. US Patent 8,135,936.

[81] Priya Sehgal, Sourav Basu, Kiran Srinivasan, and Kaladhar Voruganti. An empirical study of file systems on nvm. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

[82] Haixiang Shi, Rajesh Vellore Arumugam, Chuan Heng Foh, and Kyawt Kyawt Khaing. Optimal disk storage allocation for multitier storage system. *IEEE Transactions on magnetics*, 49(6):2603–2609, 2013.

[83] Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating system concepts*. Wiley, 2018.

[84] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *FAST*, volume 10, pages 101–114, 2010.

[85] Elizabeth Varki, Allen Hubbe, and Arif Merchant. Improve prefetch performance by splitting the cache replacement queue. In *IEEE International Conference on Advanced Infocomm Technology*, pages 98–108. Springer, 2012.

[86] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ml-based lecar. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[87] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 487–498, 2017.

[88] Hui Wang and Peter Varman. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, pages 229–242, 2014.

[89] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

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

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

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

[93] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-all replacement for a multilevel cache. In *Fast*, volume 7, pages 25–25, 2007.

[94] Gala Yadgar, Michael Factor, and Assaf Schuster. Cooperative caching with return on investment. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13. IEEE, 2013.

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

[96] 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.

[97] 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. IEEE, 2010.

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

[99] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, pages 1–27, 2020.

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

[101] Yuanyuan Zhou, James F. Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, pages 91–104, Boston, Massachusetts, June 2001.