ASPLOS'23 Accepted Paper,
28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems [ASPLOS],
March 25–29, 2023, Vancouver, BC, Canada

Re-architecting I/O Caches for Emerging Fast Storage Devices

Mohammadamin Ajdari[1,2], Pouria Peykani Sani[2], Amirhossein Moradi[2], Masoud Khanalizadeh Imani[2],
Amir Hossein Bazkhanei[1,2], and Hossein Asadi[2]
[1]HPDS Research, Tehran, Iran.— [2]Sharif University of Technology, Tehran, Iran
{m.ajdari, p.peyk, am.moradi, masoud.khanalizadeh, amir.bazkhanei, asadi}@sharif.edu

Abstract

I/O caching has widely been used in enterprise storage systems to enhance the system performance with minimal cost. Using Solid-State Drives *(SSDs) as an I/O caching layer on the top of arrays of* Hard Disk Drives *(HDDs) has been well studied in numerous studies. With the emergence of ultra-fast storage devices, recent studies suggest using them as an I/O cache layer on top of mainstream SSDs in I/O intensive applications. Our detailed analysis shows despite significant potential of ultra-fast storage devices, existing I/O cache architectures may act as a major performance bottleneck in enterprise storage systems, which prevents taking advantage of the device full performance potential.*

In this paper, using an enterprise-grade all-flash storage system, we first present a thorough analysis on the performance of I/O cache modules when ultra-fast memories are used as a caching layer on top of mainstream SSDs. Unlike traditional SSD-based caching on HDD arrays, we show the use of ultra-fast memory as an I/O cache device on SSD arrays exhibits completely unexpected performance behavior. As an example, we show two popular cache architectures exhibit similar throughput due to performance bottleneck on the traditional SSD/HDD devices, but with ultra-fast memory on SSD arrays, their true potential is released and show 5× performance difference. We then propose an experimental evaluation framework to systematically examine the behavior of I/O cache modules on emerging ultra-fast devices. Our framework enables system architects to examine performance-critical design choices including multi-threading, locking granularity, promotion logic, cache line size, and flushing policy. We further offer several optimizations using the proposed framework, integrate the proposed optimizations, and evaluate them on real use cases. The experiments on an industry-grade storage system show our I/O cache architecture optimally configured by the proposed framework provides up to 11× higher throughput and up to 30× improved tail latency over a non-optimal architecture.

1. Introduction

In recent years, I/O intensive applications in enterprise environments, e.g., databases and *Virtual Desktop Infrastructures* (VDI), have become popular and shown an increasing demand for high I/O rates and low access latency [40, 22]. Replacing arrays of *Hard Disk Drives* (HDDs) with *Solid State Drives* (SSDs) has been an early trend to comply with this performance need. With recent technology downscaling to accommodate the cost of SSD arrays, however, the performance and lifetime limitations of flash-based SSDs have placed a major barrier to further improve the overall array performance.

To meet the rising performance demands of recent data-intensive workloads, fast and ultra-fast storage devices are emerging products in the industry. For example, P5800X Optane SSDs based on 3DxPoint technology [24],[20] provides 1.5M *I/O per Second* (IOPS) for random reads and less than 10μs I/O latency. For shorter latency, persistent memory DIMMs, e.g., Intel Persistent memory [23], provide similar I/O throughput, but with much lower I/O latency.

Despite being extremely high performance, ultra-fast devices impose up to 85× higher cost (e.g., 12,150 $/TB PMEM [14, 21]) compared to the conventional SATA SSDs (143 $/TB) [37]). Such high price makes it impossible or unaffordable for business owners to build large-capacity enterprise systems (i.e., over 100 TBs [40]). Alternatively, using the conventional SSD arrays as the backend storage and adding small-sized ultra-fast devices as a temporary storage for frequently accessed blocks (i.e., I/O cache device) can be adopted as a cost-effective practice to provide both high performance and high capacity.

A limited number of existing works have studied ultra-fast devices as I/O caches, but they fail to provide sufficient insight for an I/O cache design in general *Storage Area Networks* (SANs). Some consider special use cases such as accelerating HPC applications through compute-node caching [7] or accelerating filesystem synchronization operations through buffer caching based on either PMEM or DRAM [6]. Orthus-CAS [45] considers a limited scenario where the cache device has similar IOPS (but lower latency) compared to the backend device and proposes a cache-tier load balancing mechanism. Several studies address SSD caching for backend HDD arrays [19, 39, 4, 43, 46, 19, 25, 8]. To the best of our knowledge, *none* of the previous work has examined the ultimate performance of I/O caches and *none* of them has offered any

insight into the behavior of I/O caches in presence of ultra-fast devices.

First contribution: an empirical analysis of existing I/O caches for emerging fast storage devices. In this paper, we show that the *existing insights are no longer valid* for modern SAN systems with emerging ultra-fast devices (as the I/O cache) and SSD arrays (as the backend). We observe that the same I/O cache architecture exhibits *significantly different behavior in traditional SANs compared to modern SANs.* For example, our evaluation of three real I/O caches shows that their performance is bounded to 78K IOPS in traditional SANs (SSD cache on top of HDD array) but when switching to modern SANs (e.g., RAMDisk as the I/O cache and SSD array as the backend), the internal parallelism of these caches are unleashed. Depending on the architecture, they can provide 335K to 1.5M IOPS. As another example, in traditional SANs, reducing the cache hit rate significantly degrades the performance due to serving more requests from slow HDD arrays but in modern SANs, *lower hit rate may provide opportunities to load balance* requests between I/O cache and fast SSD backend and even boost the overall performance.

Second contribution: a framework to architect I/O caches for fast storage devices. We present an *experimentation and analysis framework* to systematically understand the behavior of I/O caches on emerging fast devices and propose a scalable I/O cache architecture. Our framework has five key features: (a) proper evaluation of real system I/O caches with intrinsic complexity by relying on *industry-grade I/O cache* modules rather than simplistic cache models, (b) real measurements rather than simulations, (c) deep architectural analysis by combining real *measurements and source code analysis* of industry-grade I/O caches, and (d) generate synthetic I/O loads to cover *various I/O patterns* of enterprise workloads, and (e) high applicability of our framework to analyze the performance behavior of any I/O cache in the production environments.

Third contribution: production-level performance measure for I/O cache developers. Throughout the paper, we show the real performance of three open-source I/O caches (OpenCAS, EnhanceIO, and DM-Cache) in fast SAN environments. Developers of these caches and other I/O caches (which are usually based on such open-source caches) can benefit from our reported results.

Fourth contribution: a scalable I/O cache architecture. By using our framework, we provide a couple of major findings on how to design a scalable I/O cache for emerging SAN storage systems. **(1)** If an I/O cache lookup logic is implemented by a small number of threads (e.g., EnhanceIO cache [9]), or uses many threads but simple coarse-grained locks (e.g., locking full metadata table for tag search in DM-Cache [2]), the system IOPS is bounded to few hundred thousand IOPS. But *matching the number of cache threads to CPU cores and using fine-grained locks (e.g., one lock per small number of cache entries of interest as in OpenCAS [12]) provides multi-million IOPS scalability*. **(2)** Immediate promotion of a missed block may cause I/O cache contention while delaying the promotion by even one more access leading to a miss in the same address (e.g., *nhit-2* policy) provides opportunities to take advantage of the fast backend SSD array, which boosts IOPS by over 2× in workloads with medium locality and improves read latency predictability or *Quality of Service* (QoS), i.e., lower tail latency, by 6×-76×. **(3)** Emerging ultra-fast cache devices are small-sized and become full of dirty blocks by write bursts. We show bypassing the write misses to the backend, i.e., lazy flushing, avoids cache contention and may improve the overall IOPS (load balancing with the backend), while aggressive cache flushing to free cache blocks causes performance degradation, e.g., due to locking during victim block selection. In addition, lazy flushing improves QoS by 4.3× in high hit rate workloads. **Overall**, on real workloads, our proposed *optimal* architecture provides 1.7×-11× higher IOPS and 2.6×-30× lower 99.99% tail latency over a *non-optimal* architecture.

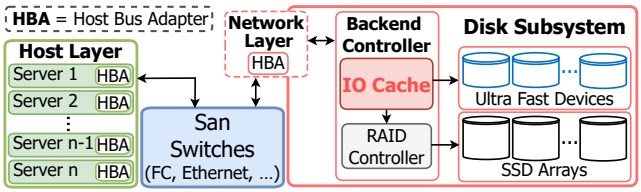


Figure 1: Common data-center architecture using SAN storage

2. Background

2.1. Storage Area Network

A *Storage Area Network* (SAN) is a high-speed storage networking architecture, which interconnects multiple servers to shared pools of storage. SAN is predominantly deployed in enterprise environments, which require low latency and high throughput. A SAN consists of *host*, *fabric*, and *storage* layers (Fig. 1). The host layer includes servers (compute nodes) attached to the SAN. Enterprise workloads such as databases run on host servers and access the storage through a dedicated networking infrastructure, typically using the *Fiber Channel* protocol, called the fabric layer. The storage layer mainly includes storage devices. Modern SANs usually leverage all-flash storage devices including SATA SSDs and NVMe devices. Storage devices are organized as arrays of disks (RAID) with data replication or parity support to provide high performance and fault tolerance. The storage space is divided into logical storage entities, each is assigned a unique *Logical Unit Number* (LUN) and is presented to host servers as a logical block disk.

2.2. Ultra-fast Storage Devices and I/O Caching

Emerging storage devices (e.g., modern persistent memories) provide orders of magnitude higher performance than conventional SSDs (Table 1); thus, they are finding their way to

Table 1: Performance-price of fast and ultra-fast storage devices. Device prices follow major online shops [5, 36]. Characteristics obtained from datasheets or published measured data of Micron 5100 Eco SATA SSD [34], Intel Optane P5800X SSD [24, 20], Intel Persistent Memory 200 series (NVDIMM) [23], and a 16GB DDR-4 DIMM (2133 MT/s) with SW block RAM Disk (brd) layer (RAMDisk).

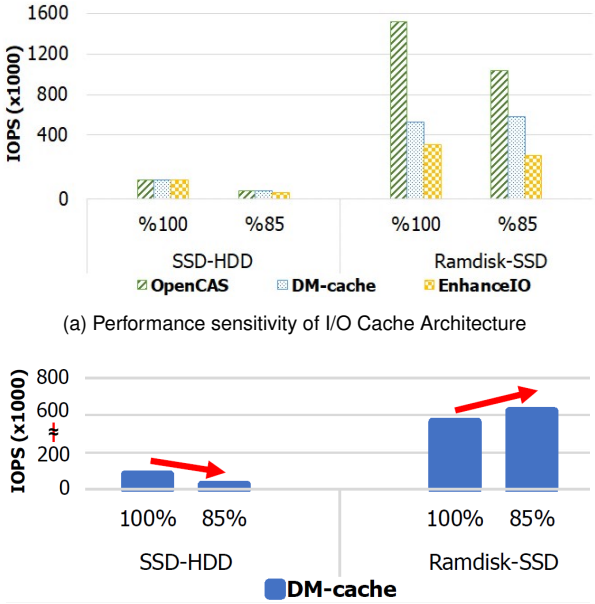
Device	SATA SSD	Optane SSD	NVDIMM	RAMDisk
Technology	NAND Flash	3DxPoint	3DxPoint	SW block on RAM
Interface	SATA	NVMe	DDR	DDR for HW
Example Capacity	3.8TB	400GB	128GB	16GB
Max IOPS (R/W)	93K/30K	1.5M/1.6M	1.9M/560K	1.8M/1.4M
4KB Latency (R)	150us	6us	<1us	<3us
Price ($/TB)	143	2500	12,150	6400

mission-critical and high-performance SAN storage systems. An example conventional SATA SSD provides up to 93K IOPS for random reads and 24K IOPS for random writes. Recent NVMe SSDs (and Optane SSDs) boost the I/O rates by 6× to 14× and provide very short latency compared to SATA SSDs. To handle increasing performance demands, moving toward ultra-fast devices with near-DRAM performance is vital. Emerging persistent memory DIMMs push the performance closer to DRAM, with sub-microsecond latency and over a million IOPS. RAMDisk is also an example of an ultra-fast device with its DRAM hardware and software layer to make a block device. RAMDisk with one DIMM also provides about 2 million IOPS and less than 3 μs latency.

Although ultra-fast storage devices provide high performance, they can be up to 85× more expensive than conventional devices (Table 1). This makes using ultra-fast devices at large capacities very difficult, which motivates using them as small-sized persistent I/O caches for existing conventional SSD arrays. I/O caching in the SAN storage stack is responsible to track frequently accessed data and following its caching policy, place them on the ultra-fast device. If the running application on the host has enough access locality, SAN storage serves most of the I/O requests from the I/O cache device. In this case, the performance of the SAN storage increases toward that of an ultra-fast device, while it provides the large capacity of the conventional SSD arrays.

3. Motivation

Here, we provide three insights to motivate the need for architectural redesign of I/O caches and relying on industry-grade I/O caches for valid analysis on emerging fast storage devices. First, we reveal that existing analyses on I/O caches, which mainly focus on SSD caching for HDD arrays are not directly applicable to emerging fast storage devices. We also show I/O caching in fast environments exhibits completely different behavior. Second, we show that I/O caches for real environments are very complex, e.g., 60,000 *Lines of Code* (LOC) in OpenCAS, which makes their evaluation either intractable or very inaccurate using simulation-based environments or simplistic cache models. Hence, any evaluation of architectural



(a) Performance sensitivity of I/O Cache Architecture

(b) Performance trend change

Figure 2: Motivation to redesign I/O caches: (a) high sensitivity of performance to architectural choices with emerging devices and (b) opposite performance trend compared to widespread SSD cache on HDD arrays

ideas necessitates to use industry-grade I/O caches rather than using simulation-based toolsets. Third, we show that enterprise workloads have a wide spectrum of I/O patterns, which requires a valid analysis to cover various workload patterns.

3.1. Need for Redesign of I/O Caches

We observe that various I/O cache architectures with different levels of complexity show almost similar performance on traditional SAN storage environment with *SSD cache on HDD-array backend* due to device-level bottleneck; however, in fast SAN storage systems with *emerging fast devices as cache devices and SSD-array as backend*, such I/O caches show significantly different performance for the same workload, and even reverse performance trend across two workloads (Fig. 2). To experiment, we use three real, enterprise-grade I/O cache modules (OpenCAS, DM-Cache, and EnhanceIO) and run two I/O workloads with random 4KB read accesses with 100% and 85% cache hit rate potential. We use an SSD as the cache layer with RAID5(8+1) HDDs (as the backend) in a traditional setup and RAMDisk (as the ultra-fast cache) with RAID5(8+1) SSDs as the backend of the modern setup.

Our observation reveals that emerging fast SAN storage systems are highly dependent on the I/O cache architecture, while this was not true for traditional SANs and hence, redesigning for scalability is crucial. For example, OpenCAS, DM-Cache, and EnhanceIO with default configuration show up to 1.5M IOPS, 540K IOPS and 335K IOPS for 100% cache hit workload, respectively; however, all of these three cache modules show roughly 78K IOPS in a traditional SAN setup

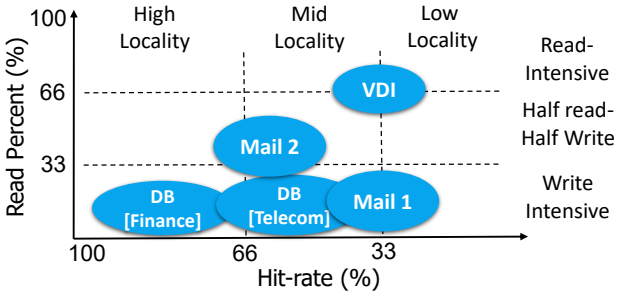


Figure 3: I/O patterns in enterprise workloads. Cache space is set as 1%-5% of workload access range. Mail1: two weeks of FIU mail server traces [29], Mail2: Microsoft Exchange mail traces [35]), VDI: VDI traces [30], DB [Finance]: a transactional database for financial sector (Oltpbench TPCC on MySQL DB [15, 11]), DB [Telecom]: a database used in telecom companies (OLTPBench TATP on MySQL DB [15, 11])).

(Fig. 2a). As another example, DM-Cache on a traditional environment shows lower performance when a workload hit rate changes from 100% to 85%; however, with a fast SSD array as the backend, a lower hit rate even provides 15% higher performance (Fig. 2b). Such reverse performance trend is an attribute of a fast backend array compared to an HDD array, which enables the DM-Cache cache promotion policies to indirectly perform load balancing across the cache device and backend, which was not possible in traditional slow-backend arrays.

3.2. Need to Experiment Real Caches in Architectural Design

An I/O cache in a SAN storage is a complex piece of code, which may behave completely differently compared to simplified implementations of cache models. For example, Open-CAS [13], an open source I/O cache initiated by Intel, has over 60,000 lines of code and is designed for deployment in the block I/O layer of Linux kernel. DM-Cache is another I/O cache module, which has smaller code base (8,300 LOC), but tightly depends on using external functions in Linux kernel, which increases the complexity of its component interactions. Our observations show that a) it is very hard to build an I/O cache within an operating system from scratch and b) evaluations of new policies or popular existing policies on simplified cache implementations may not be valid for SAN storage systems. Thus, it is mandatory to rely on the implementations of existing real I/O caches to do a practical performance analysis on emerging fast storage devices.

3.3. Variety of I/O Patterns

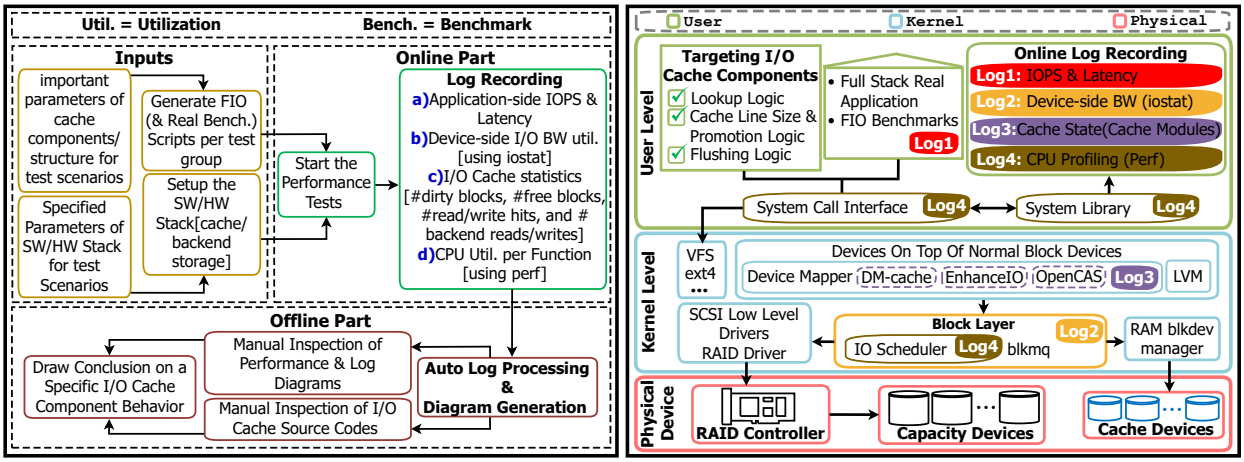
Enterprise workloads have a variety of performance requirements and I/O patterns (Fig. 3). Our characterization of five major workloads shows that the I/O behavior of such workloads varies from low access locality (e.g., VDI [30]) to high access locality (e.g., the database of financial transactions on MySQL DB [15, 11]), read-intensive (e.g., VDI) to write-intensive (e.g., some mail servers [29]). Among such workloads, some require high average performance (i.e., IOPS and average latency as in large-scale mail servers) and some require short tail latency (i.e., QoS guarantee as in financial transactional databases). The wide spectrum of I/O patterns makes the I/O cache design challenging, especially in the presence of ultra-fast devices.

4. Our Proposed Framework

To understand the behavior of I/O caches on emerging fast devices and design a scalable architecture, we propose an *experimentation and analysis framework with four considerations*. **First**, to properly evaluate the I/O caches with their intrinsic complexity, we use industry-grade I/O cache modules rather than simplistic cache models. **Second**, to consider full-stack effects, we use real system measurements and statistics logging at different layers instead of simple simulations. **Third**, to provide in-depth architectural analysis, we combine conclusions of real measurements with expectations from source code analysis of industry-grade I/O caches. **Fourth**, we design specialized synthetic tests to cover various I/O workload patterns, in addition to final measurements with real traces.

I/O cache components/architecture/workload choices. We classify the major components of I/O caches for emerging storage devices into three groups: (1) lookup logic (2) cache line size and promotion logic, and (3) flushing logic. These components have a few important architectural choices following industry-grade I/O caches (Table 2). The first component, *lookup logic*, is mainly about *the level of multi-threading and locking granularity* of cache architecture. We evaluate three major industry-grade implementations of this layer through synthetic workloads with 100% hit rate and different I/O parallelism (i.e., number of I/O threads). For the second component, *cache line size, and promotion logic*, we analyze the effect of different cache line sizes and three important promotion policies (*always*, *nhit*, and *SMQ*). The workload patterns that we use for this component cover different I/O request sizes, I/O access locality (hit rates), and I/O access skewness. Finally, the third component, *flushing logic*, has two popular architectural choices that we analyze; *lazy flushing and aggressive flushing*. we also evaluate these architectural choices by mixed read-write workloads and different access localities.

Proposed Framework. To systematically analyze the architectural choices, we propose a partly automated framework with three stages: inputs, online part, and offline part (Fig. 4). At the **first stage** (*Inputs*), our master script generates the desired synthetic tests, in addition to setting up the SW/HW stack following our desired scenarios. In our master script, we define which I/O cache components and at what conditions should be analyzed in each test, and accordingly, we define the range of the values for the parameters of the test (e.g., I/O access locality and I/O request type). Setting up the SW/HW stack (e.g., the backend HW array and system parameters) is executed in this stage.



(a) Proposed framework (b) System setup

Figure 4: Our proposed framework and system setup for architectural analysis of major components in Linux I/O caches

Table 2: Major I/O cache components and selected important design choices analyzed for a scalable I/O cache design on emerging fast devices. CG: Coarse-Grained, FG: Fine-Grained, HT: Highly Multi-threaded, LT: Low Multi-threaded

I/O cache components	Architecture/implementation choices analyzed	Workload feature dependency	Answer (Sec. #)
1) Lookup logic internal parallelism?	(a) LT + F.G. Locking (b) HT + C.G. Locking (c) HT + F.G. Locking	Number of I/O submitting threads	5.1
2) Cache line size and promotion logic?	(a) Cache line size (b) Promotion: always, nhit, SMQ	(a) Block size of I/O requests (b) I/O access locality [cache hit rate] (c) I/O access skewness	5.2
3) Flushing logic?	(a) Delayed lazy flushing (b) Aggressive periodic flushing	(a) Amount of workload writes (b) I/O access locality	5.3

Table 3: Design choices in block-level caches. AG:aggressive

Caches	OpenCAS	EnhanceIO	DM-cache	bcache
Cache Line Size	4KB-64KB	2KB-8KB	32KB-1GB	512B-4KB
Promotion	always nhit	always	SMQ	always
Flushing	lazy, AG	AG	AG	AG
Lookup	HT+FG	LT+FG	HT+CG	HT+CG

At the **second stage** (*online part*), our framework runs the script on the desired system configurations and uses system profiling tools to log statistics at different layers. We record application-side performance (IOPS or latency), I/O bandwidth utilization on the backend array (in the block layer using *iostat* tool [17]), I/O cache statistics (in the device mapper layer), and CPU utilization per system function (using *perf* tool [44]).

At the **third stage** (*offline part*), our framework collects the statistics logs in different layers and automatically converts them into summarized useful visual diagrams. Such diagrams would show how each architectural choice affects the performance of the I/O cache along with other statistics (in the cache, and other layers) for easier reasoning. By combining visual experiment data with manual inspection of the I/O cache source at designated components, we provide in-depth valuable insights on the performance effect of architectural choices on I/O caches for emerging fast devices.

I/O cache component prototypes. Following the motivation to use real I/O caches, our analysis reveals architectural design choices of popular industry-grade block-level I/O caches (Table 3). We select three caches with widely different architectures to serve as our real prototypes in performance analysis. Selected caches are OpenCAS (initiated by Intel), EnhanceIO (initiated by STEC Inc.), and DM-Cache (initiated by IBM). The software versions we use are OpenCAS version 21.3 [12], Lanconnected branch of EnhanceIO [1], and DM-Cache integrated with CentOS 7.

Synthetic workloads. We use FIO to run synthetic workloads with desired I/O patterns for specific I/O cache components. We set the number of I/O jobs and I/O queue depth to 16, and the I/O request size to 4KB. For selected tests, we modify the number of I/O jobs or I/O request size, which is explicitly mentioned in each section.

We use FIO zoned random distribution to create workloads with desired cache hit rates. For example, for a workload with 55% cache hit rate and 60 GB cache size, we define 55% of read accesses to be on the first 60GB address range and the remaining 45% to be on the rest of address range. Exact values for zoned accesses to achieve specific hit rates were set through multiple trial and error. For selected tests that require highly skewed locality, we use FIO *Zipf* distribution. Note that

we do not run sequential I/O workloads in this work, because enterprise workloads with sequential accesses especially with large blocks (e.g., video recording) may quickly fill up the expensive small-sized ultrafast caches and cannot properly benefit from such environments. In fact, most storage vendors do not recommend using I/O cache for sequential workloads.

Devices for I/O caching. We deploy our I/O caches on top of RAMDisk as an example emerging fast DRAM-like device, but the results are applicable to other emerging devices with similar performance characteristics.

Experimental setup. We use a Chenbro storage server with dual-socket Intel Xeon E5-2620 v4 8-core CPUs (total 32 logical cores), 9× Samsung SM863a 1.92TB SSDs setup using MegaRAID 9361. We set up a 6TB partition from a RAID5(8+1) as back-end storage, unless explicitly other configurations are stated. Our I/O cache device is 60GB, which is as small as 1% of back-end storage to be practical in enterprise environments with petabyte-scale storage servers.

Real workload usecases. In addition to synthetic workload patterns, we also evaluate the proposed cache architecture with real I/O traces. We select three large enough real traces with widely different behaviors: (1) VDI (read-intensive), (2) enterprise mail server (with half-reads, half-writes), and (3) transactional database workload (write-intensive).

VDI Trace: We select a single volume (LUN0) of a VDI trace [30] for two consecutive days (2016.2.24-25), one day as a warm-up and the second day as the main test. We re-parse the trace and distribute the I/O across 16 threads for more realistic I/O parallelism. The total amount of I/O in both days is equal to 1TB, from which 78% are I/O reads. We set the I/O cache to 1% of the workload address range (i.e., 50GB).

Microsoft Exchange traces: We replay three volumes (i.e., volumes 2-4) of Microsoft Exchange mail traces [35]. In this trace, 52%-57% of I/O requests are reads. To replay these three volumes simultaneously and evaluate the effect of caching more realistically, we set up a logical volume group (using LVM) on top of the I/O caching layer. We make three LUNs, each with around 480GB of size, which is the address range of the trace volumes. We then replay trace volume 2 to 4 on LUN1 to LUN3, respectively. We set up the I/O cache with the size of 5% of the address range of three LUNs (i.e., 74GB). We consider the first half of the trace I/Os as a warm-up and the second half as the main test.

Oltpbench TPCC: This workload on MySQL database models the transactions in financial environments [11, 15]. In this workload, over 95% of requests are writes. We first set the benchmark scale factor to 3750 warehouses to create a 400GB table. We run the workload with 25 terminals for 20 minutes (5 minutes as a warm-up and 15 minutes as the main phase). Since this workload is CPU intensiveness, we first capture its block I/O trace and then replay it with 32 threads at maximum speed to evaluate the cache performance without worrying of CPU bottleneck of workload generator. The used I/O cache has 5% size of database table (i.e., 20GB).

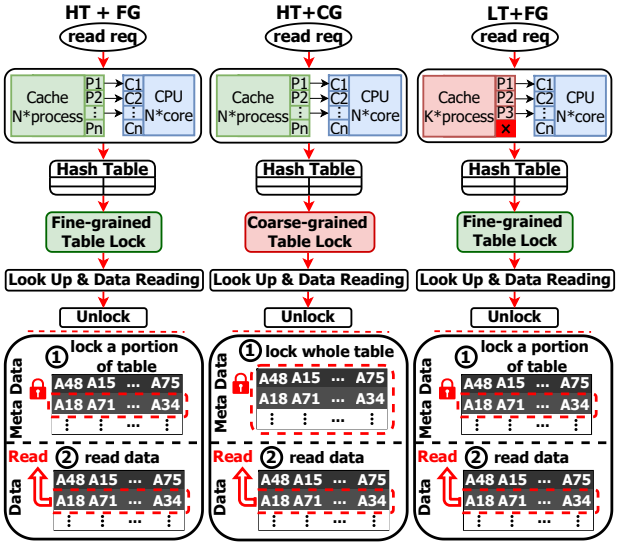


Figure 5: I/O cache lookup logic in three popular architectures

5. Analysis and Findings

5.1. Internal Scalability in I/O Caching

Question 1: *What architecture for request lookup management provides the most scalable hit-request management on emerging fast I/O cache devices?*

The core of lookup operations in any I/O cache is usually the same: it hashes an I/O request address, looks up the tag in the metadata table, and reads the data. Based on the number of parallel cache processes and the granularity of accessing shared cache resources, we consider three popular architectures (Fig. 5). The first architecture is *Low multi-Threaded (LT)* and has minimal parallelism with a fixed number of processes (or threads) to handle specific operations (e.g., one process for metadata update and one for flushing). This is easy to implement and is used in popular I/O caches (e.g., EnhanceIO). The second architecture (similar to DM-Cache) is *Highly multi-Threaded with Coarse-Grained locks (HT+CG)*. It can take advantage of many CPU cores for parallel I/O handling but uses coarse-grained locks (i.e., locking the full cache metadata table for tag search/update). The third architecture is *Highly multi-Threaded with Fine-Grained locks (HT+FG)* as in OpenCAS. It takes advantage of a high core count, but the locking is more optimized (i.e., locking is applied at set or bucket granularity instead of full cache locking).

Finding 1: *As the number of I/O submitting jobs increases, the architecture with the number of threads matched to the CPU core count, accompanied by a fine-grained locking mechanism (e.g., in OpenCAS) provides linear performance scalability; however, the architecture with low, fixed number of threads (e.g., in EnhanceIO), or multi-threaded design with coarse-grained locking (e.g., in DM-Cache) causes major performance bottleneck on emerging fast SAN systems.*

Modern storage applications demand supporting many I/O submissions at the same time (e.g., in databases or multi-VM

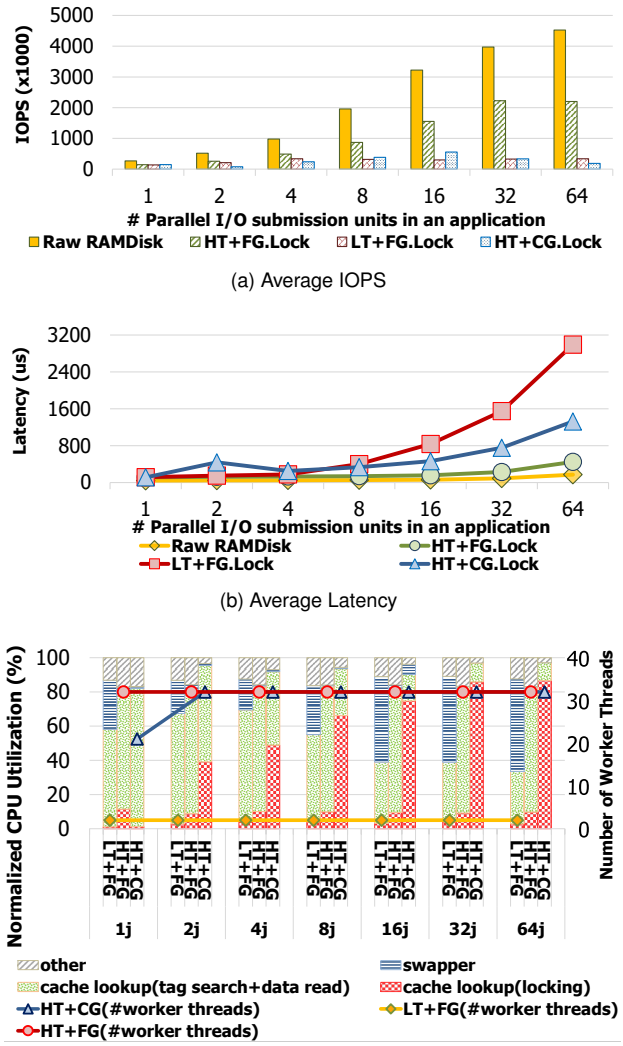


Figure 6: Evaluating effect of three lookup logic architectures in I/O caches when used with emerging storage devices. (Raw RAMDisk is the software-based block device built on RAM Disk without any cache module involved to show the upper performance potential of any I/O cache on RAMDisk)

environments) and a simplified but valuable way to check the architecture scalability is through evaluating its IOPS-latency and CPU utilization breakdown under a varying number of I/O submitting jobs (Fig. 6). We observe that I/O caches with LT architecture saturate at low IOPS. For example, EnhanceIO saturates with four jobs at 350K IOPS. Increasing the I/O load is not supported by the limited cache parallelism, thus the load generator is put to sleep (ie., CPU cycles of the swapper process increase). For HT+CG design, as the number of I/O jobs increases from 1 to 64, this architecture suffers from increased spin-lock overheads to over 80% of CPU cycles, which causes a sharp performance degradation with high parallel loads. HT + FG architecture exhibits the most scalable design. Increasing the number of jobs does not change the portion of CPU cycles

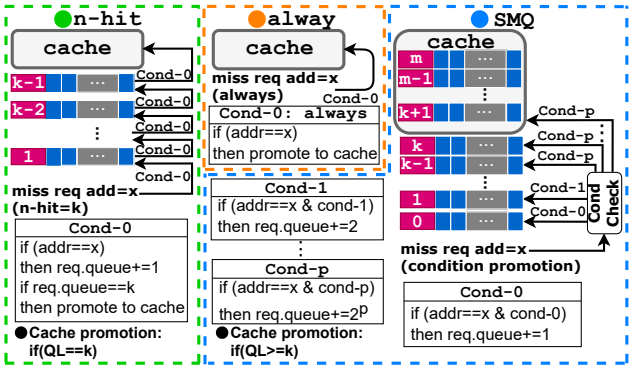


Figure 7: Conceptual view of three I/O cache promotion logic

for locking and swapper. This enables the performance of this architecture to linearly scale to over 2.2M IOPS.

Note that fine-grained locking and multi-threading are known techniques to improve application performance (e.g., as stated in Shinjuku [26]). In this paper, for the first time, we reveal the architecture of three industry-grade caches (regarding threading and locking decisions) and also show a complete insight on how fine-grained locking compared to coarse-grained locking and low-multi threading compared to high multi-threading affects the performance of I/O caching in emerging fast SAN storage environments.

5.2. Read Miss Management

Question 2. *Which cache line size and promotion logic architectures best suit the read-intensive workloads running on systems equipped with emerging fast I/O cache devices?*

Fig. 7 shows the conceptual view of three popular promotion logic architectures; namely *always*, *nhit*, and *SMQ*. *Always* is the most simple and popular promotion logic that promotes a block to the cache as soon as the block address is detected as a miss (i.e., not found in the cache). This policy may provide a cache hit for the next time that the same address is accessed. However, due to the limited cache space, this policy may sometimes result in *cache pollution* and degrade performance. To account for this issue, some I/O caches use *nhit* policy. In this policy, instead of immediate promotion, after N times that the same block address is accessed, the block is promoted to the cache. This reduces cache contention and serves more requests from the backend. *Stochastic Multi-Queue (SMQ)* is also an extended, more complex version of *nhit* that assumes (a) multiple queues or levels that requests move up till they reach a level that promotes the block to the cache and (b) the number of steps moving up or down in queues is determined following predefined conditions. For example, DM-Cache implements a variation of SMQ that uses 64 queues, in which requests start from the first queue and are gradually elevated till they reach the level for promotion to the cache. The number of moving up or down in queues is usually determined by cache hit rates.

Finding 2a. *Delaying a missed request by one access*

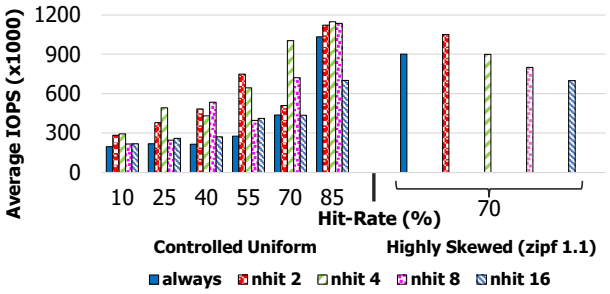


Figure 8: Exploring effect of different I/O cache promotion logic on system IOPS for read-intensive 4KB workloads with different cache hit rates and locality

before promoting to the cache (i.e., nhit-2) usually provides a load balancing between the backend SSD array and fast cache device, which boosts IOPS, while other promotion policies either cannot take advantage of the fast cache device due to very delayed promotions (e.g., nhit-16) or limit performance due to contention on the I/O cache device (i.e., always policy).

To observe the performance effect of different promotion logic architectures, we first run read-intensive workloads with a very low hit rate (10%) to very high hit rate (85%). These workloads have fairly uniform access in specified regions (we name *controlled uniform workloads*). We also run a workload with *zipf* 1.1 distribution (with most requests accessing a very small region) to observe the effect of access skewness.

On workloads with controlled uniform accesses, the *nhit* policy always performs better than the simple, widely-used *always* promotion policy, with some workloads showing up to 3× higher IOPS on nhit (Fig. 8). The *nhit* promotion logic provides less cache contention and also serves some requests from the SSD array. In modern storage systems with ultra-fast I/O cache and very fast SSD array in the backend, serving requests from the backend is not always undesired and can be interpreted as load balancing between the cache device and the backend. Optimal value of *N* in *nhit* is partly workload dependent. However, our measurements show two solid results: (a) With delaying the promotion by one hit (i.e., *nhit-2* policy), most workloads with small-block accesses work pretty well on modern fast SAN systems, (b) significantly delaying the promotion of a block to the cache (e.g., *nhit-16*) results in low performance especially at high hit rates due to losing an opportunity of serving blocks from the ultra-fast cache device.

Unlike the controlled uniform workloads, those with highly skewed accesses show smaller improvement with *nhit* policy, with *nhit-2* providing about 10% IOPS speedup. We use the random read workload with *zipf* 1.1 distribution, which provides about 70% cache hit rate to a region as small as 1GB. With a such high frequency of per-address accesses, the best promotion logic is the one that maximizes opportunities of serving such requests from the ultra-fast cache. While *always* policy may seem like the best choice, it causes the cache to get filled quickly and some hot accesses become replaced by rarely accessed blocks. *Nhit-2* delays the time that the cache becomes full, thus hot blocks always stay in the cache and provide a decent 10% IOPS speedup. Further delaying the requests (e.g., *nhit-4*) provides similar performance to *always* policy or even degrades performance (as in *nhit-8* or *nhit-16*).

Finding 2b. *Cache line size is highly correlated to the promotion logic especially at larger cache lines, but has little dependence on the I/O granularity of workloads. Increasing cache line size, accompanied by delayed promotion policy (i.e., nhit) boosts performance while larger cache lines make the immediate promotion policies (i.e., always) to provide up to 4× lower performance than with default 4KB lines.*

We conduct numerous experiments with different cache line sizes, promotion policies, workload localities, and block sizes. Due to limited space, we show only *128 representative results* with low hit rate (25%) and high hit rate (85%) workloads (Fig. 9). We observe that increasing the cache line size naturally reduces the metadata overheads of cache management, but also increases unit of data fetching from the backend, whose efficiency depends on the promotion logic.

The *always* policy has suitable performance with 4KB workload and 4KB cache line sizes. However, increasing the cache line size to larger values (e.g., 32KB) makes the *always* policy to fetch large blocks to handle partial misses, which in turn degrades the performance by up to 75% at workloads with high locality. However, delaying the block promotion using *nhit* policy avoids such problem and boosts performance, especially at larger cache blocks, with *nhit-2* or *nhit-4* providing the best performance. Low hit rate workloads also exhibit similar performance trend with less difference between *always* and *nhit* policy due to less overall efficiency of cache usage. For workloads with larger block sizes (e.g., many read requests of an SQL DB server [16]), the default cache line size in many caches (e.g., 4KB in OpenCAS and EnhanceIO [13, 1]) limits the performance at low IOPS (320-482 KIOPS) due to frequent metadata management and locking overheads. Increasing the cache line size to match the workload block size (i.e., 32KB blocks) with *nhit* policy provides up to 2× speedup and saturates at around 765K IOPS.

Finding 2c. *While delaying the promotion of missed requests (i.e., nhit policy) improves average IOPS and average latency, a properly tuned nhit policy has the potential to guarantee the quality of service (QoS) of read requests at up to 76× lower latency than default always policy.*

Guaranteeing QoS on a cache requires minimizing unpredictable behaviors arising from cache contentions. Delaying the promotion of requests properly (i.e., *nhit* policy with proper value) enhances the predictable performance in read-intensive workloads (Fig. 10). In our experiments with workloads of 25% hit rate (i.e., low locality) to 85% hit rate (i.e., high locality), *nhit-4* provides the lowest latency. As switching from 99% percentile to 99.99% percentile, the performance gap of *nhit-4* with other policies becomes significant. Workloads with lower hit rate and more address randomness show more cache contention with *always* policy. If *nhit* policy with

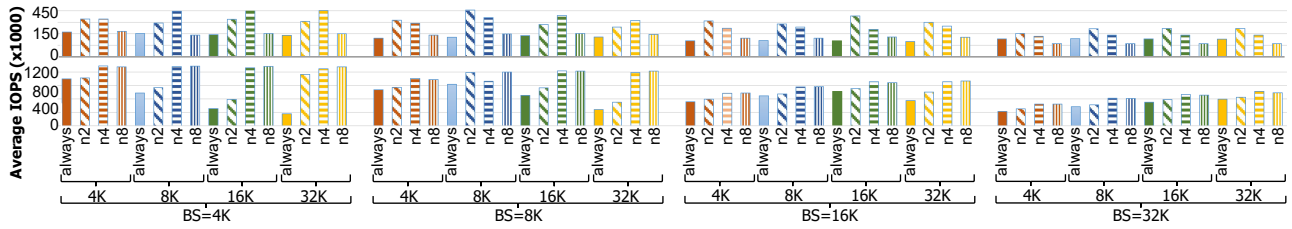


Figure 9: Exploring performance relation of workload block size and locality with cache structure and promotion policy. Upper diagrams: workloads with 25% hit rate, lower diagrams: workloads with 85% hit rate

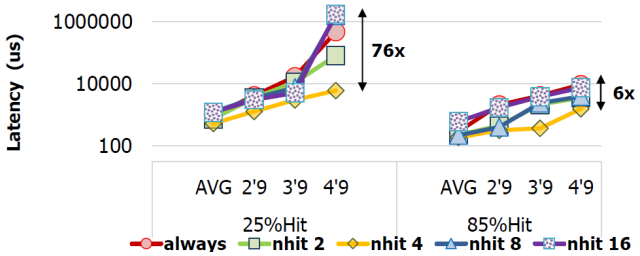


Figure 10: Effect of I/O cache promotion logic on tail latency of read requests. 2'9: 99%, 3'9: 99.9%, 4'9: 99.99% tail latency

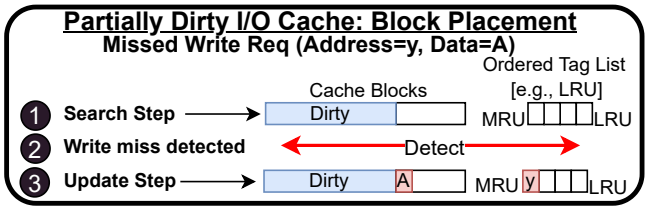
proper value is set, the tail latency is reducible by 76×, but an improperly configured *nhit* policy is either partially optimal (e.g., *nhit-2*) or degrades performance (e.g., *nhit-16*). When the workloads have a higher hit rate, *nhit-4* still provides the lowest tail latency, but the performance gap with the *always* policy is reduced from 76× down to 6×.

5.3. Write Request Management

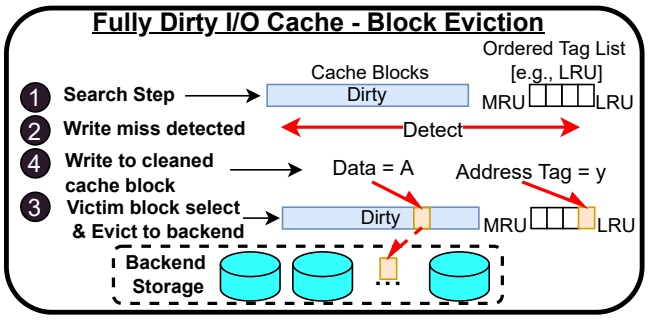
Before a cache is full, all write requests are easily written to the cache. Interesting scenarios arise when the cache becomes full (of dirty blocks). Such a situation easily happens for emerging fast I/O caches due to high prices and low capacity devices used. For example, a cache based on 200GB optane DIMM may become fully dirty in just 100-second write burst. Thus, the design of flushing policies for such cases becomes very important.

Question 3. *For workloads with long write bursts, which policies for eviction, flushing, and promotion are well-suited for the properties of emerging fast I/O cache devices?*

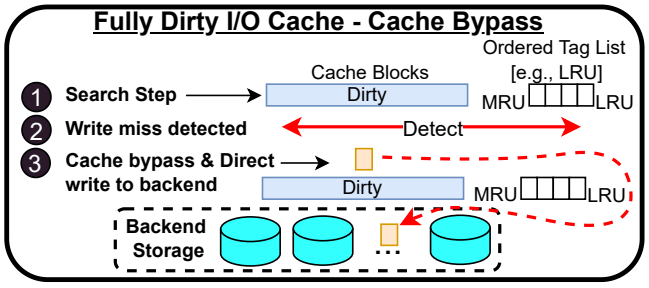
Handling a write request miss when the cache is not full is usually the same across different implementations: searching the cache (step 1), detecting a write miss (step 2), finding an empty cache block and updating that block with new data (step 3) (Fig. 11a). When the cache is full of dirty blocks, however, two popular approaches exist to handle a write request (Fig. 11b and Fig. 11c). The first approach considers newly received writes to have a priority over older blocks. In this approach, after the request lookup in the cache (step 1) and detecting a write miss (step 2), the cache management logic selects a victim cache block (e.g., the least recently used block using LRU list) and evicts its content to the backend storage (step 3). After the completion of the block eviction, the newly received block is written to the freed cache location (step 4). In the second approach (as shown in Fig. 11c), after detecting



(a) Partially dirty cache



(b) Fully dirty cache: eviction policy



(c) Fully dirty cache: cache bypassing

Figure 11: Conceptual view of write policies (a) when the cache is partially dirty and (b and c) when it is fully dirty

the write miss (step 2), the cache module does *not* evict any cache block and thus does not write the request data to the cache. Instead, it bypasses the cache device by directly writing the write-request data to the backend (step 3). While the first approach may be suited for SANs with very slow HDDs, modern backend storage with fast SSD arrays may take advantage of the second approach as well.

Handling a write request is also correlated with the cache flushing mechanisms, namely, aggressive flushing and lazy delayed flushing (Fig. 12). Aggressive flushing follows victim block selection and eviction under any circumstances. Follow-

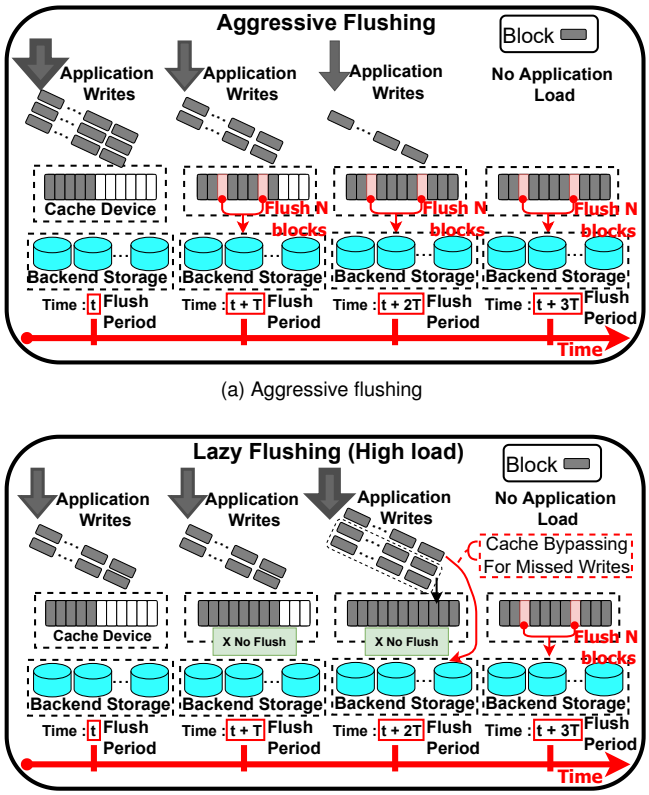


Figure 12: Conceptual view of flushing dirty cache blocks using (a) aggressive flushing and (b) lazy flushing

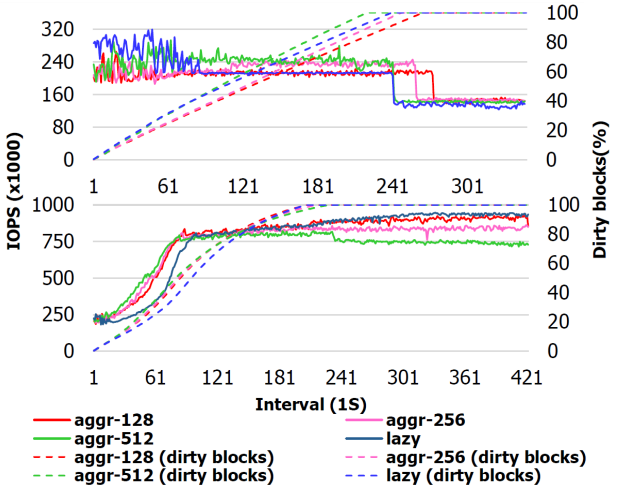


Figure 13: Effect of different flushing mechanisms when cache is full for random 70/30 read-write workloads with 10% hit rate (upper diagram) and 85% hit rate (lower diagram)

ing the conceptual view of its effect on performance, when aggressive flushing periodically starts its operation, the arrival I/O rate decreases. This is due to the additional load on the cache to read victim blocks and clearing the corresponding dirty flag. When the I/O cache is full, the block eviction becomes more forceful to free blocks for new write requests, and this may further degrade performance. On the contrary,

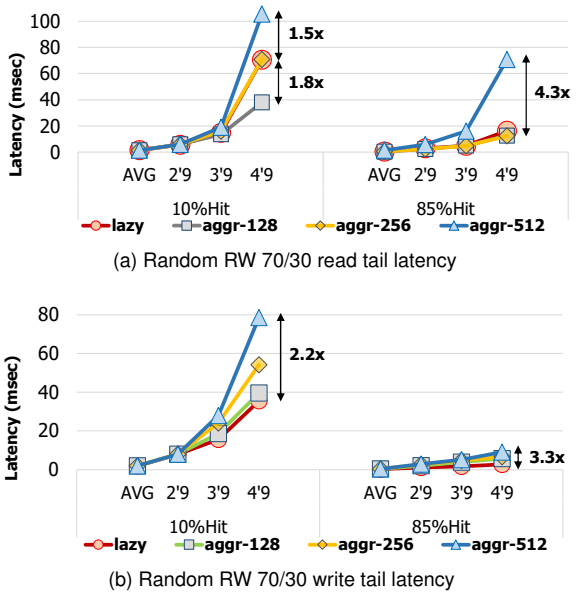


Figure 14: Lazy flushing vs. aggressive flushing for (a) read tail latency, (b) write tail latency, of read-intensive workloads

lazy flushing does *not* flush any blocks unless the I/O cache is idle (to avoid any I/O contention). If the I/O cache becomes full of dirty blocks and it has a high load, write misses bypass the cache, and in effect, the I/O arrival write bandwidth becomes the sum of the I/O cache bandwidth and backend bandwidth, thus, further boosting performance. In our following experiments, lazy flushing is the same as the default flushing policy in OpenCAS while aggressive flushing is the `ACP` policy in OpenCAS (or similar to the default flushing policy in DM-Cache).

Finding 3. *For workloads that have long write bursts, using delayed lazy flushing over aggressive flushing provides the easiest implementation with less cache contention and better load balancing between the fast backend SSD array and the ultra-fast I/O cache, improving the average performance by 10% and boosting QoS at 4.3× lower tail latency.*

Fig. 13 shows the flushing policy effect on IOPS of two workloads with low and high hit rates starting from the empty cache till the cache becomes fully dirty after a few intervals. We observe that at low hit rate workloads, after the cache is fully dirty, whether to flush aggressively (i.e., `ACP`) or avoid flushing (i.e., lazy flushing) has almost *no* effect on IOPS. In such workloads, most requests are served from the backend (due to low hit rate) and thus, trying to free some cache blocks (through flushing) may not provide enough opportunities for speedup. On the other hand, for workloads that have a high hit rate (e.g., 85% hit rate), when the cache is fully dirty, lazy flushing boosts the performance by 10% from 845K to 940K IOPS. In this case, most requests would hit in the cache and only a small portion needs to be served from the backend. As the backend in modern SANs (i.e., SSD array) is fairly fast, bypassing the cache can help load balancing between the cache

device and the backend. Using aggressive flushing with small batches (i.e., the default with 128 blocks in single flush) has a fairly similar result to lazy flushing, but increasing the batch size increases the load on the cache (mainly locking cache entries) and imposes a 20% lower IOPS than lazy flushing.

While average latency is fairly similar across different flushing policies, the tail latency can be significantly lower using lazy flushing (Fig. 14). For 99.99% write tail latency, lazy flushing compared to aggressive flushing can be up to 2.2× better in low hit rate workloads and up to 3.3× better in high hit rate workloads. We expect this gap to be hugely due to the effects of locking cache blocks during flushing (and avoiding writes to those blocks for consistency guarantee). Note that aggressive flushing with the small batch size is fairly similar to lazy flushing at low hit rate, but this is not the case for high rate workloads. Similarly, lazy flushing provides up to 4.3× lower read tail latency than aggressive flushing with large batches. Aggressive flushing with minimal batch size provides almost similar latency to lazy flushing, as we expect write-locks for flushing do not block read requests, thus have no big impact on performance. These results show that lazy flushing is (especially at a high hit rate) almost the best choice for high average performance and QoS guarantee while there is no major concern about the flush batch size.

6. Real Usecases

In this section, we evaluate real workloads (explained in Sec. 4) and run each one for *62 different architectural choices*. We reveal that the optimal design choices with a properly designed/configured cache (i.e., our proposed architecture) provides significantly higher performance over improperly configured/designed caches. Note that improperly configured caches easily arise due to many system administrators using the default configuration of an I/O cache or setting the cache configuration following incomplete insight of the cache behavior in emerging applications and environments.

6.1. Average IOPS

Our proposed optimal architecture for I/O caching on three major real workloads provides 1.7×-11× higher IOPS compared to non-optimally configured architecture (Fig. 15a, 15b, 15c). The optimal architecture in these three workloads is consistent with our findings: HT+FG (as lookup logic) to maximize internal parallelism of handling I/O requests, lazy or aggr-128 (as flushing logic) which minimizes the flushing load on the cache device while handling normal I/O requests, cache line size of 16KB-32KB which is almost matched to the workload average request sizes, and *nhit-2* or *nhit-4* (as promotion logic). We observe that the difference between optimal and non-optimal architecture is as high as 11× (100K IOPS vs. 1.1M IOPS) on TPCC, which has 32 parallel I/O jobs and about 85% cache hit rate. On MS Exchange traces and VDI that have lower parallel jobs and lower hit rate (55% and 35%, respectively), the optimal architecture has less opportunity for speedup but is still 1.7× faster on MS Exchange and 2.3× on VDI (44.9 KIOPS vs. 103 KIOPS).

6.2. Tail Latency

For 99.99% tail latency (QoS), our proposed optimal architecture improves the performance by up to 2.6× for read latency and over 30× for write latency of measured real workloads compared to non-optimal architecture (Fig. 15d, 15e, 15f). The optimal cache architecture design choices for low tail latency are similar to high IOPS choices, but the promotion logic is an exception. Promotion logic (e.g., *nhit* vs. *always*) is about load balancing requests between the cache and the backend (i.e., nhit policy) compared to efforts to serve everything from the cache (i.e., always policy). Load balancing requests with the backend may improve the overall I/O rate but improper use of such technique would suffer from backend SSD array latency (100s of microsecond) instead of the RAM Disk cache latency (less than 10 microseconds). In VDI, which has a low hit rate while 30% of requests are writes (thus making cache dirty and hard to use for read requests), significantly delaying request promotion (e.g., nhit-8 or nhit-16) minimizes the tail latency (while nhit2 or nhit-4 works well for high IOPS). On the other hand, when the cache hit rate is high as in MS Exchange or TPCC and high locality exists (e.g., zipf access distribution in TPCC), then *always* promotion policy or *nhit-2* may also work well and improve the tail latency.

7. Related Work

SSD Caching for HDD Arrays. Many existing works use HDDs as backend storage and exploit a faster device such as an SSD as the cache [3, 19, 39, 4, 43, 46, 33, 18, 19, 27, 25, 8]. ECI-Cache [3] presents a hypervisor-based caching architecture that prioritizes data based on reuse distance and request type. LARC [19] uses a virtual LRU queue to improve the LRU algorithm for a higher hit rate, and thus better performance. Few studies [4, 43] propose bypassing the SSD cache for selected write requests and submit the writes to HDDs. All these studies assume HDDs as the backend storage, and following our motivational data, their performance results may *no* longer be valid in fast SAN systems, which we target in this paper.

Some studies propose models to improve performance. These models include machine learning for cache write management [46] and techniques to estimate cache miss rate for specific cache configurations [33, 18]. The domain of these studies differs from modern SANs, but their approach is orthogonal to ours and can be used as a complementary mechanism.

I/O Caching on Ultra-fast Devices. A limited number of studies use faster SSDs (i.e., NVMe SSDs) as the caching layer [6, 7, 28, 45, 31]. Chen et. al [6] use non-volatile main memories and DRAM as buffer caching to improve performance and lifetime of SSDs. This work focuses on frequent synchronization operations of file systems to improve the per-

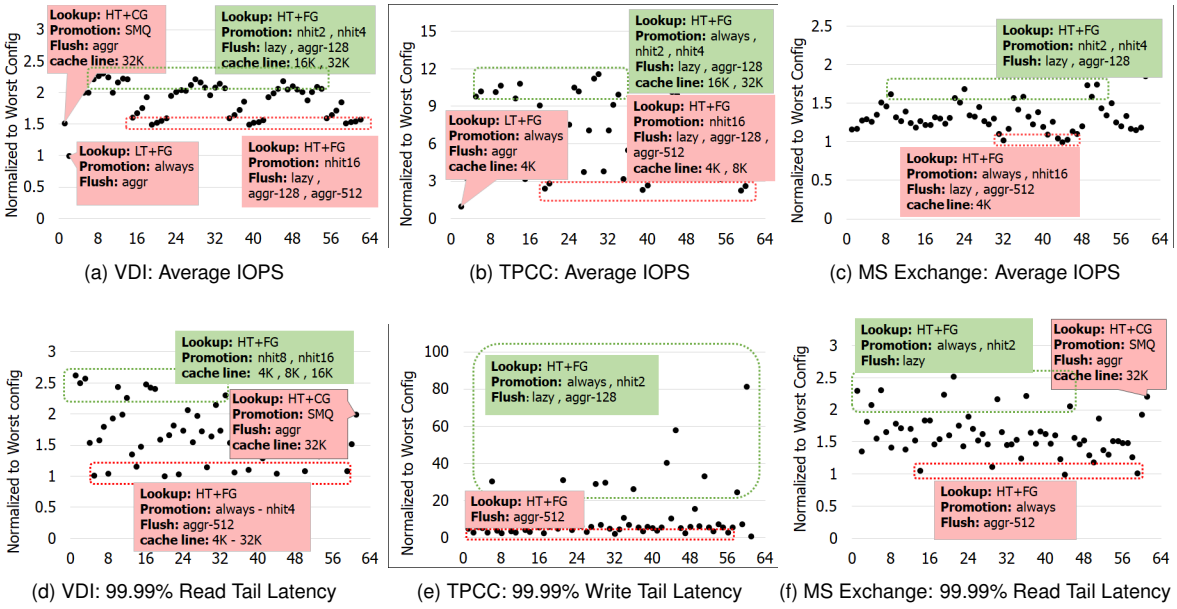


Figure 15: Performance of three major real workloads under various design choices for an I/O cache on an ultra-fast device

formance and endurance of SSDs, instead of our target on general block storage. Cheng et. al [7] propose an NVMM-based caching in the Lustre file system on the compute-node side (instead of SAN storage side) of HPC clusters. Kim et. al [28] suggest utilizing application hints to cache write requests and passing such hints to low-level storage layers, which is orthogonal to our work. OrthusCAS [45] proposes a cache-tier for modern storage systems and assumes the cache device and backend storage have similar IOPS, but the cache has lower latency. The authors propose a load balancing mechanism, which is similar to *nhit* promotion policy with dynamic *N* selection. We may adopt their policy to improve the promotion logic and combine it with our other proposed per-component architecture. A recent work characterized the performance of DM-cache and b-cache on NVDIMM [31]. However, they do not consider recent high-performance I/O cache architectures (e.g., OpenCAS), important design choices (e.g., flushing logic), and workload behaviors (e.g., I/O access locality).

Small Cache Inside SSD Device. Few works have aimed for improving the management of the small cache inside SSDs [41, 42, 32]. For example, Tripathy et. al [42] improve the performance of concurrently running workloads on NVMe SSDs by prioritizing workloads and reducing the request interference at the NVMe SSD internal device (DRAM) cache. Co-Active [41] modifies the management policy of NVMe SSD internal device (DRAM) cache to reduce the request response times and also improves SSD lifetime. Our work with I/O caching in the OS layer is orthogonal to these studies.

8. Discussion

RAMDisk as an ultrafast device. RAMDisk is very fast and easily deployable in SAN storage systems and does not require special support from CPU and motherboard (unlike persistent memory DIMMs). Thus, we used RAMDisk throughout the paper, but our results are applicable to any fast memory with similar performance characteristics.

Analyzed architectures. With the goal of providing an I/O cache architecture for real SAN storage environment, we focused on popular choices in industry-grade I/O caches. While a few I/O cache policies exist, most of them are either minor variations of our analyzed choices or are less popular in industry (e.g., due to implementation difficulties).

Our framework logging overhead and accuracy. We avoid the overheads of log collection by using three techniques: (a) using separate SSDs for OS (and logs) and data SSDs, thus no I/O contention happens between logging and normal workload I/O requests, (b) enabling only required logging features per test scenario, (c) controlling the period of each log sample. In our tests, *Perf* tool has the most overhead (less than 10% effect) on IOPS, but we only enable it for specific CPU task breakdown tests in Section 5.1.

Risk of data loss in different flushing policies. If the cache device is volatile (e.g., DRAM), all flushing policies (e.g., aggressive or lazy) will have a risk of data loss, due to keeping some dirty blocks in the cache. In our paper, we use RAMDisk as an example of emerging fast devices. In practice, enterprise systems are either battery-backed (e.g., by using Universal Power Supply) or use non-volatile fast persistent memory; thus, flushing more frequently (i.e., aggressive) or less frequently (i.e., lazy) has practically no effect on data loss.

How the community can benefit from our paper. The community can use our findings in a couple of ways. First, we revealed that system administrators should NOT use the best practices of SSD caching on HDDs to tune I/O caches for emerging fast devices. Second, our proposed framework can be used to analyze the performance behavior of any I/O

caches. Third, our findings are helpful for architects to convert a conventional I/O cache to a scalable one on emerging fast devices. Fourth, many existing I/O caches are designed based on open-source caches, thus developers and companies relying on the examined caches (OpenCAS, DM-Cache, and EnhanceIO) can directly benefit from our reported performance results to tune their caches.

9. Conclusion

In this paper, we showed that I/O cache architectures in HDD-array SANs are not applicable to modern SAN systems. Through evaluating major architectural choices on real I/O caches, we provided guidelines on how to make a scalable I/O cache for emerging storage systems. We revealed that compared to non-optimal configuration, our proposed architecture provides up to 11× higher IOPS and 30× lower tail latency.

10. Acknowledgements

We would like to thank Eno Thereska, our shepherd, and the anonymous reviewers for their insightful comments. This research was funded by *High Performance Data Storage and Processing* (HPDS) Corporation, Tehran, Iran.

A. Artifact Appendix

A.1. Abstract

Throughout this artifact, we briefly explain about the implementation of our proposed framework and its software/hardware requirements. We plan to release the source codes and more details on how to run the framework through GitHub.

A.2. Artifact Check-list

- **Compilation:** gcc 4.8.5, Python 2.7.5, and Python 3.6.8 to run the prerequisites (e.g., I/O caches). The main code of our framework consists of bash scripts, which do not require any separate compilation.
- **Data set**: We use two groups of data sets: synthetic and real traces. We generate the synthetic data sets using our scripts and FIO tool. The real data sets are based on public I/O traces of Microsoft Exchange server, VDI, and the benchmark of Oltpbench TPCC (refer to Section 4 for more details).
- **Hardware:** Almost all of our scripts run on any hardware that can provide two block devices (one as the cache device and the other as the backend). To provide expected results, the hardware configurations must match the hardware specifications stated Section 4.
- **Output:** Our scripts generate a couple of files as the output of each experiment: (1) overall I/O performance results in a single FIO output file (average IOPS, average latency, 99%, 99.9%, 99.99% tail latency), (2) various periodic system logs including cache statistics (number of allocated blocks, number of dirty blocks, read/write hit rates, etc.), disk bandwidth usage and periodic performance logs (especially, IOPS), (3) *Perf* statistics (i.e., CPU utilization breakdown) for specified experiments.
- **How much disk space required (approximately)?:** Each experiment requires about 5MB of disk space to store the monitoring logs. For all experiments throughout the paper, about 5GB space is required to store the experiment logs. Real trace runs require an additional 6GB for the input traces.
- **How much time is needed to prepare workflow (approximately)?:** 10-15 minutes
- **How much time is needed to complete experiments (approximately)?:** The experiment time significantly varies (approximately 2-60 minutes) depending on the performance of cache device and backend storage and the test scenario.
- **Publicly available?:** We plan to publicly release the main codes of our proposed framework through GitHub.

A.3. Description

A.3.1. How to access We plan to release codes through GitHub [38].

A.3.2. Hardware dependencies Using the same system hardware as we specified in the paper would result in similar output to what we showed. However, using any other server configuration with minimal hardware requirements makes it still possible to run our scripts, but would naturally lead to different absolute performance, and system resource utilization numbers.

Recommended configuration: The same as specified in Section 4 of this paper.

Minimal configuration: A server with at least an 8-core CPU (total 16 logical cores), one server-grade SSD as the backend (in addition to the OS disk), and 32GB DRAM, from which 10GB are usable as the I/O cache device. Note that this configuration leads to different absolute performance numbers compared to our paper results, and thus is only suitable to check the ability to run our scripts.

A.3.3. Software dependencies Running the experiments require the benchmarking tool FIO (version 3.8), CentOS 7 with kernel updated to 5.4.52, OpenCAS version 21.3, Lanconnected branch of EnhanceIO, and DM-Cache integrated in CentOS 7. The user needs to ensure that both python version 2.7.5 (for EnhanceIO), and python version 3.6.8 (for OpenCAS), in addition to libaio (for FIO) are installed in their system.

A.4. Installation

No installation is needed for the main scripts but before you can use scripts, make sure that your system meets the prerequisites as follows. The first prerequisite is FIO tool. First, ensure *libaio* is installed in your system. Then, download FIO version 3.8 source code from its repository [10], use the specified three simple commands for installation. Then install OpenCAS [12], and EnhanceIO [1] following the instructions in each cache repository (DM-Cache is already installed in CentOS 7 and no further action needed).

A.5. Evaluation and Expected Results

Once you run scripts for a specific test scenario, our framework generates various results in a single output directory. One of the main summarized results shows the average IOPS, average latency, and tail latency during the test. If the hardware and software environments of the test are same as the paper, the expected output results would match what we have shown in the paper; otherwise, they may become significantly different.

A.6. Experiment Customization

Our framework is easily customizable thanks to its organized structure and bash scripting platform. Example customizable parameters include (1) I/O cache module, (2) the block device provided as the I/O cache device and the block device provided as the backend, (3) cache-specific parameters (size, policy, etc.) (4) workload specific parameters (block size, locality, etc.), and (5) FIO-replayable real traces of interest.

References

[1] lanconnected fork of git repository for enhnanceio. https://github.com/lanconnected/EnhanceIO, visited on Oct 5 2022.

[2] The linux kernel user's and administration guide: Cache. https://www.kernel.org/doc/html/v5.4/admin-guide/device-mapper/cache.html, visited on Oct 5 2022.

[3] Saba Ahmadian, Onur Mutlu, and Hossein Asadi. Eci-cache: A high-endurance and cost-efficient i/o caching scheme for virtualized platforms. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(1):1–34, 2018.

[4] 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)*, DATE'19, pages 1196–1201, USA, 2019. IEEE.

[5] Amazon. Amazon online shop. http://www.amazon.com, visited Oct 4, 2022.

[6] Youmin Chen, Youyou Lu, Pei Chen, and Jiwu Shu. Efficient and consistent nvmm cache for ssd-based file system. *IEEE Transactions on Computers*, 68(8):1147–1158, 2018.

[7] Wen Cheng, Chunyan Li, Lingfang Zeng, Yingjin Qian, Xi Li, and André Brinkmann. Nvmm-oriented hierarchical persistent client caching for lustre. *ACM Transactions on Storage (TOS)*, 17(1):1–22, 2021.

[8] 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*, ATC'16, pages 379–392, USA, 2016. USENIX.

[9] EnhanceIO Community. Stec inc git repository of enhnanceio. https://github.com/stec-inc/EnhanceIO, visited on Sept 28 2022.

[10] FIO Community. Fio repository. https://github.com/axboe/fio, visited on Sep 22, 2022.

[11] Oltpbench Community. Oltpbench repository, 2021. https://github.com/oltpbenchmark/oltpbench visited on Jan 30 2023.

[12] OpenCAS Community. Git repository of opencas, 2021. https://github.com/Open-CAS/open-cas-linux, visited on Jan 30 2023.

[13] OpenCAS Community. Open cache acceleration software, 2022. https://open-cas.github.io/, visited on Jan 30 2023.

[14] DELL. Dell memory upgrade - 128gb - 3200mhz intel optane™ pmem 200 series, 2022. https://www.dell.com/en-us/work/shop/dell-memory-upgrade-128gb-3200mhz-intel-optane-pmem-\200-series/apd/ab614354/memory, visited on Sept 22 2022.

[15] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, December 2013.

[16] Argenis Fernandez. Sql server's io block size, 2016. https://blog.purestorage.com/purely-technical/what-is-sql-servers-io-block-size/, visited on Oct 5 2022.

[17] Sebastien Godard. iostat tool manual, 2022. https://man7.org/linux/man-pages/man1/iostat.1.html visited on Jan 30, 2023.

[18] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. Fast miss ratio curve modeling for storage cache. *ACM Transactions on Storage (TOS)*, 14(2):1–34, 2018.

[19] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving flash-based disk cache with lazy adaptive replacement. *ACM Transactions on Storage (TOS)*, 12(2):1–24, 2016.

[20] alper Ilkbahar. The future of intel optane persistent memory, 2020. https://newsroom.intel.com/wp-content/uploads/sites/11/2020/12/The-Future-of-Intel%C2%AE-Optane%E2%84%A2-Persistent-Memory-Alper-Ilkbahar-.pdf, visited on Sept 24 2022.

[21] Insight.com. Intel optane 200 256gb ddr-t persistent memory module, 2022. https://www.insight.com/en_US/shop/product/P23535-B21/HEWLETT+PACKARD+ENTERPRISE/P23535-B21/Intel-Optane-Persistent-Memory-200-Series---DDR-T--\-module---256-GB---DIMM-288-pin---3200-MHz---PC4-\25600/#, visited on Sept 28 2022.

[22] Moor Insights and Strategy. Storage-optimized machine learning: Impact of storage on machine learning, 2018. https://www.purestorage.com/content/dam/pdf/en/white-papers/protected/wp-moor-storage-optimized-machine-learning.pdf, visited on Sept 22 2022.

[23] Intel. Achieve greater insight from your data with intel optane persistent memory, 2021. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html, visited on Sept 28 2022.

[24] Intel. Intel optane ssd p5800x series, 2021. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html, visited on Sept 24 2022.

[25] Yichen Jia, Zili Shao, and Feng Chen. Slimcache: An efficient data compression scheme for flash-based key-value caching. *ACM Transactions on Storage (TOS)*, 16(2):1–34, 2020.

[26] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'19, pages 345–360, USA, 2019. USENIX.

[27] 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.

[28] Sangwook Kim, Hwanju Kim, Sang-Hoon Kim, Joonwon Lee, and Jinkyu Jeong. Request-oriented durable write caching for application performance. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, FAST'15, pages 193–206, USA, 2015. USENIX.

[29] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, September 2010.

[30] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*, Systor '17, pages 1–11, New York, NY, USA, 2017. ACM.

[31] Geonhee Lee, Hyeon Gyu Lee, Juwon Lee, Bryan S Kim, and Sang Lyul Min. An empirical study on nvm-based block i/o caches. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys'18, pages 1–8, New York, NY, USA, 2018. ACM.

[32] Weiguang Liu, Jinhua Cui, Junwei Liu, and Laurence T Yang. Mlcache: a space-efficient cache scheme based on reuse distance and machine learning for nvme ssds. In *2020 IEEE/ACM International Conference On Computer Aided Design*, ICCAD'20, pages 1–9, USA, 2020. IEEE.

[33] 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, USA, 2021. USENIX.

[34] MICRON. Micron 5100 series sata nand flash ssd datasheet, 2016. https://www.mouser.com/datasheet/2/671/5100_ssd-1283974.pdf, visited on Jan 30 2023.

[35] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys'09, pages 145–158, New York, NY, USA, 2009. ACM.

[36] Newegg. Newegg online shop. http://www.newegg.com, visited on Sept 28 2022.

[37] Newegg. Micron 5100 eco 3.84tb solid state drive-2.5inch internal-sata (sata/600), 2022. https://www.newegg.com/micron-3-8tb-5100/p/1E4-006U-00015, visited on Oct 4 2022.

[38] HPDS Research. Repository of i/o cache performance analysis framework, 2023. https://github.com/HPDSResearch/io_cache_performance_analysis_framework, visited on Feb 14 2023.

[39] Reza Salkhordeh, Shahriar Ebrahimi, and Hossein Asadi. Reca: An efficient reconfigurable cache architecture for storage systems with online workload characterization. *IEEE Transactions on Parallel and Distributed Systems*, 29(7):1605–1620, 2018.

[40] Spiceworks. Data storage trends in 2020 and beyond, 2020. https://www.spiceworks.com/marketing/reports/storage-trends-in-2020-and-beyond/, visited on Sept 28 2022.

[41] Hui Sun, Shangshang Dai, Jianzhong Huang, and Xiao Qin. Co-active: A workload-aware collaborative cache management scheme for nvme ssds. *IEEE Transactions on Parallel and Distributed Systems*, 32(6):1437–1451, 2021.

[42] Shivani Tripathy, Debiprasanna Sahoo, Manoranjan Satpathy, and Madhu Mutyam. Fuzzy fairness controller for nvme ssds. In *Proceedings of the 34th ACM International Conference on Supercomputing*, SC'20, pages 1–12, New York, NY, USA, 2020. ACM.

[43] Shucheng Wang, Ziyi Lu, Qiang Cao, Hong Jiang, Jie Yao, Yuanyuan Dong, and Puyuan Yang. {BCW}: Buffer-controlled writes to hdds for ssd-hdd hybrid storage server. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, FAST'20, pages 253–266, USA, 2020. USENIX.

[44] Perf Kernel Wikipedia. Perf: Linux programming with performance counters, 2022. https://perf.wiki.kernel.org/index.php/Main_Page, visited on Jan 30, 2023.

[45] 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)*, FAST'21, pages 307–323, USA, 2021. USENIX.

[46] Yu Zhang, Ke Zhou, Ping Huang, Hua Wang, Jianying Hu, Yangtao Wang, Yongguang Ji, and Bin Cheng. A machine learning based write policy for ssd cache in cloud block storage. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, DATE'20, pages 1279–1282, USA, 2020. IEEE.