PRISM: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices

Yongju Song*
Sungkyunkwan University
South Korea
yongju@skku.edu

Wook-Hee Kim
Konkuk University
South Korea
wookhee@konkuk.ac.kr

Sumit Kumar Monga
Virginia Tech
USA
sumitkm@vt.edu

Changwoo Min
Virginia Tech
USA
changwoo@vt.edu

Young Ik Eom†
Sungkyunkwan University
South Korea
yieom@skku.edu

ABSTRACT

As data generation has been on an upward trend, storing vast volumes of data cost-effectively as well as efficiently accessing them is paramount. At the same time, today's storage landscape continues to diversify, from high-bandwidth storage devices such as NVMe SSDs to low-latency non-volatile memory (*e.g.*, Intel Optane DCPMM). These heterogeneous storage devices have the potential to deliver high performance in terms of bandwidth and latency with cost efficiency, while achieving the performance and cost targets together still remains a challenging problem.

We provide our solution, PRISM, a novel key-value store that utilizes modern heterogeneous storage devices. PRISM uses heterogeneous storage devices synergistically to harness the advantages of each storage device while suppressing their downsides. We devise new techniques to balance the latency-bandwidth tradeoff when reading from SSD. For ensuring multicore scalability and crash consistency of data across heterogeneous storage media, PRISM proposes cross-storage concurrency control and cross-storage crash consistency protocols. Our evaluation shows that PRISM outperforms state-of-the-art key-value stores by up to 13.1× with significantly lower tail latency.

CCS CONCEPTS

• **Information systems** → **Key-value stores**; **Hierarchical storage management**; **Storage class memory**; **Flash memory**.

KEYWORDS

Key-value Stores, Non-volatile Memory, Heterogeneous Storage System

*Dept. of Electrical and Computer Engineering
†Dept. of Electrical and Computer Engineering / College of Computing and Informatics

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575722

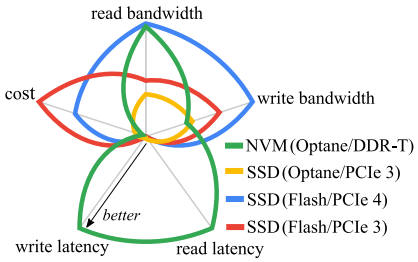
ACM Reference Format:
Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. PRISM: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575722

1 INTRODUCTION

Key-value stores have been a critical component of storage infrastructure in a wide range of applications, including database systems [1, 7, 13, 19, 22, 32, 39, 54, 55, 74, 76, 79, 81], caching systems [28, 67], distributed file systems [11, 17], distributed analytics [20, 31], and serverless platforms [44, 87]. Efficient key-value stores should provide high performance cost-effectively. In particular, the amount of data generated and stored is growing exponentially [23], and many emerging application domains require high performance.

For decades, storage systems have harnessed the storage hierarchy for cost efficiency. The hierarchical approach classifies storage devices into *performance devices* and *capacity devices*. Performance devices offer high performance in all aspects – high bandwidth and low latency, at the expense of high price ($/TB). On the other hand, capacity devices provide high capacity at low price at the cost of low performance. SSD and HDD have manifested the performance and capacity layers, respectively. Most key-value stores [21, 45, 47, 89] have employed hierarchical designs – such as caching, tiering, and layered LSM-tree architecture – to balance the performance and cost using the performance and capacity layers, by placing frequently accessed hot data on performance devices.

However, recent advances in storage hardware technologies blur the clear separation between performance and capacity layers. For instance, flash SSDs with PCIe Gen 4 provide higher bandwidth than byte-addressable NVM (Intel Optane DCPMM) at a 27× lower price, but NVM has two orders of magnitude lower latency than SSD, as shown in Figure 1. Moreover, the upcoming CXL-based persistent memory expansion [25, 69] will provide more performance and cost tradeoff options. Hence, there is no longer an explicit dichotomy between performance and capacity devices in today's storage landscape. Prior studies also made similar observations – "storage jungle" [38] and "non-hierarchical storage" [88]. As a result of recent evolution towards a non-hierarchical, overlapping



Specification		Performance				Cost
Type	Model	Read BW (GB/s)	Write BW (GB/s)	Read Latency (usec)	Write Latency (usec)	$/TB
DRAM	SK Hynix DRAM (16GB/DDR4)	15	15	0.08	0.08	5,427
NVM	Intel Optane DCPMM (128GB/DDR-T)	6.8	1.9	0.30	0.09	4,096
NVM SSD	Intel Optane 905P (960GB/PCIe 3)	2.6	2.2	10	10	1,024
Flash SSD	Samsung 980 Pro (1TB/PCIe 4)	7	5	50	20	150
Flash SSD	Samsung 980 (1TB/PCIe 3)	3.5	3	60	20	100

Figure 1: Heterogeneous storage media. Due to recent advances in SSD and NVM technologies, there is no clear separation between *performance devices* and *capacity devices*. For instance, while byte-addressable NVM offers the shortest access latency, flash SSD (PCIe Gen4) provides the highest bandwidth at a 27.3× lower cost ($/TB). Moreover, PCIe Gen5 SSDs are expected to double the bandwidth and IOPS [5].

storage landscape, many optimizations developed for conventional hierarchical storage systems are neither cost-effective nor optimal.

In this paper, we seek an answer to the question: *"How should we design a key-value store in a non-hierarchical storage landscape?"*. We answer the question by proposing a novel key-value store, Prism, considering storage heterogeneity as a first-class citizen. We leverage each individual storage device's type strengths – flash SSDs have high bandwidth and low cost, while NVMs have low latency and high endurance – to compensate for their respective disadvantages so that we can achieve high performance and cost-effectiveness together. We revisit the performance and cost requirements of each component of key-value store for strategic placement of them on heterogeneous storage devices. We propose the Prism architecture, where each Prism component is placed on the storage device that best satisfies its requirements. Components of Prism are scattered across multiple heterogeneous storage devices, so achieving efficient crash consistency and concurrency control across heterogeneous storage devices is of paramount importance. Thus, we also introduce cross-media crash consistency and concurrency control techniques. As a whole, Prism derives a synergistic interaction between heterogeneous storage devices, allowing them to unleash their full potential.

This paper makes the following contributions:

- We propose Prism, a novel heterogeneous key-value store. For synergistic use of heterogeneous storage devices, Prism consists of Persistent Key Index, Persistent Write Buffer (PWB), Value Storage, Scan-aware Value Cache (SVC), and Heterogeneous Storage Index Table (HSIT). We judiciously place each component based on its performance and cost requirement. Additionally, we propose efficient crash consistency and concurrency control techniques across storage media, and an opportunistic thread combining scheme to achieve high bandwidth and low latency for flash SSD reads. As a whole, Prism components work synergistically to maximize the advantages of individual storage devices and suppress their disadvantages.
- We implemented Prism with DRAM, byte-addressable NVM (Intel Optane DCPMM), and flash SSDs. We thoroughly evaluate Prism against state-of-the-art key-value stores. Our evaluation shows that Prism outperforms other key-value stores by up to 13.1× in throughput. Also, Prism's tail latency is shown to be lower by up to 8.7×.

2 BACKGROUND AND MOTIVATION

2.1 Evolution of Storage Heterogeneity

For decades, the storage hierarchy has consisted of a *performance layer*, providing superior performance in every aspect (*e.g.*, bandwidth, latency) at the cost of higher price, and a *capacity layer*, offering ample capacity at lower price but with lower performance. As such, flash SSDs and HDDs have respectively embodied the performance and capacity layers. This clear division has successfully balanced the performance and cost of a storage system. However, such a dichotomy no longer applies to today's storage devices. Rapid evolution of storage hardware technologies has taken place, including byte-addressable NVM (*e.g.*, Intel Optane DCPMM [12]), ultra-low-latency SSDs (*e.g.*, Intel Optane SSD [40], Samsung Z-SSD [77]), NVMe SSDs with faster PCIe connections (*e.g.*, Gen 4 and Gen 5 [5]), and CXL-based persistent memory expansion [25, 69], as shown in Figure 1. We now compare NVM (Intel Optane DCPMM specifically) and flash SSD along various dimensions such as performance, scalability, and cost.

Performance. NVM provides DRAM-like access latency and enables direct access using load and store instructions, eliminating the storage stack overhead. Flash SSD has much higher latency than NVM (50 us vs. 0.3 us). Also, the storage stack for flash SSD further amplifies its access latency [91]. However, NVM possesses limited bandwidth, which is even lower than PCIe Gen 4 SSDs (6.8GB/s vs. 7GB/s for reads and 1.9GB/s vs. 5GB/s for writes). Moreover, the bandwidth gap will continue to increase as upcoming SSDs supporting PCIe Gen 5 will deliver even higher bandwidth (*e.g.*, 13GB/s for reads, 6.6GB/s for writes [5]).

Scalability. There is a clear limitation in scaling the capacity and bandwidth of NVM by aggregating more NVM DIMMs in a server as memory channels in a processor restrict memory slots [38]. In contrast, the capacity and bandwidth of SSDs are linearly scalable using a RAID controller [35]. The bandwidth gap between SSD and NVM will become wider due to steady developments in PCIe and SSD aggregation.

Capacity and endurance cost. While the cost ($/TB) of current NVM is lower than DRAM, it is 27×- 40× more expensive than flash SSDs. In terms of device lifespan, flash SSD has much lower endurance than NVM (0.6 PBW[1] vs. 292 PBW). Actually, Optane

[1]PBW: Peta Bytes Written.

DCPMM's endurance would be practically unlimited as it takes about 5 years to reach the lifetime writes with the maximum write bandwidth [41].

Insight #1. While NVM provides extremely low latency and very high endurance, it has lower bandwidth than flash SSD. Moreover, it is unrealistic to keep all data in NVM due to its high capacity cost ($/TB), the limitation in capacity and bandwidth scaling compared to flash SSD. In summary, today's storage devices cannot be easily bisected into *performance devices* and *capacity devices*. Prior studies have also made similar observations: *"the storage hierarchy is not a hierarchy"* [88] and *"the hierarchy is becoming a jungle"* [38]. Based on the non-hierarchical nature of today's storage devices, we propose to consider NVM as a latency and endurance device and SSD as a bandwidth and capacity device.

2.2 Managing the Storage Heterogeneity Today

Storage systems have traditionally employed hierarchical designs to balance the performance and cost by leveraging the performance and capacity layers (*e.g.*, SSDs vs. HDDs) in a cost-effective manner. The core question in hierarchical storage system design has been: *How to efficiently identify and place hot data into the performance layer, aiming to maximize hits on performance devices [29, 49, 52, 73, 75, 80, 82, 93].*

For managing such storage hierarchy, *caching* [16, 30, 67, 86] and *tiering* [27, 34, 61, 78] have been widely used. With caching, data is copied to a performance device from a capacity device whenever accessed. With tiering, data is not necessarily promoted to a performance device immediately but rather hot data is identified based on access patterns and promoted periodically.

LSM-tree based key-value stores [21, 45, 47, 89] have been extended to leverage the storage hierarchy, putting recently written (*i.e.*, likely frequently accessed) data into the performance device in the upper layers. For instance, NoveLSM [47] places memtable on NVM. SLM-DB [45] manages a global index in NVM and maintains SSTables on a single level unlike conventional LSM trees. MatrixKV [89] proposes fine-grained column compaction using NVM. SpanDB [21] exploits two types of SSDs (Performance SSD and Capacity SSD) as a legacy storage hierarchy for cost-efficiency. Nevertheless, LSM-tree compaction and the inefficient data traversing still exist, significantly deteriorating the system performance.

Among file systems, Ziggurat [92] and Strata [53] leverage NVM and SSD. Ziggurat [92] dynamically decides where to write to either NVM or SSD based on system call patterns. Strata [53] first writes to NVM log then the log is periodically digested to SSD. However, they still treat NVM and SSD hierarchically so they traverse data layer by layer, resulting in wasting CPU cycles. Also, overall system performance may be bounded by storage devices with the lowest performance.

In sum, it is clear that current hierarchical storage systems have constraints in fully drawing the great potential of modern heterogeneous storage devices.

Insight #2. Hierarchical storage systems worked well for the conventional storage hierarchy consisting of performance and capacity layers. However, the hierarchical approach does not work well on today's heterogeneous storage devices. Traditional hierarchical storage architecture cannot fully leverage each storage medium's individual strengths. Consider an example of placing hot data on NVM: a storage system can leverage the low latency of NVM, but it could end up suffering from NVM's inherent limited bandwidth. In the era of non-hierarchical storage systems, the design of storage systems has diverse aspects to consider beyond simply placing "hot data" on high-performance devices. Rather, a more fundamental question should be: *How should each component in a storage system be designed to synergistically leverage the strengths of each storage device while compensating for the weaknesses of each of them.*

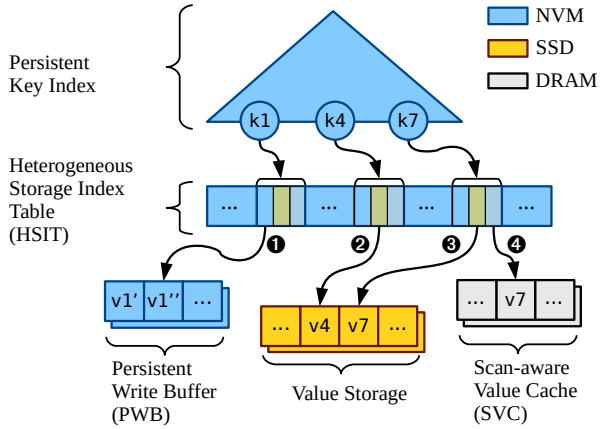


Figure 2: Illustration of Prism storing three key-value pairs: {k1,v1"}, {k4,v4}, and {k7,v7}. The Persistent Key Index maps a key into the location of a corresponding value, which could either be in the Persistent Write Buffer on NVM (v1") or in Value Storage on SSD (v4, v7). Values on SSD can be cached to the Scan-aware Value Cache on DRAM (v7). Prism decouples the Key Index from the value location through the Heterogeneous Storage Index Table, making cross-media crash consistency and concurrency control simple and efficient.

3 DESIGN GOALS OF PRISM

Synergistic interaction of heterogeneous storage devices. We aim to realize the full potential of each storage medium to maximize performance and cost-efficiency. In particular, we exploit the following advantages: (1) SSD – high capacity and bandwidth scaling with lower cost, (2) NVM – DRAM-like latency, high endurance, and better scaling and cost-efficiency than DRAM, and (3) DRAM – lowest latency and highest bandwidth.

CPU efficiency and multicore scalability. Today's storage devices offers high performance so they are no longer the primary performance bottleneck. Instead, the CPU has now become a new major bottleneck. For instance, Lepers *et al.* [57] showed that many optimization techniques (*e.g.*, sorting, compaction, bloom filter) developed for conventional storage devices (*e.g.*, HDD) are counterproductive, wasting CPU cycles and resulting in the CPU becoming the performance bottleneck. In Prism, we consider the CPU-efficient design, by minimizing the overhead of the software stack, especially in critical paths, and aim to provide good multicore scalability with concurrent requests.

Cross-storage media crash consistency & concurrency. Crash consistency is a vital aspect of storage systems. Thus, Prism requires an efficient crash consistency mechanism for its components spread across multiple storage devices. In addition, Prism must be capable of supporting a high level of concurrency to maintain multicore scalability and fully utilize low-latency or high-bandwidth storage devices. In particular, we should avoid scenarios where one slow device becomes a bottleneck in handling crash consistency or concurrency control of the entire system.

4 DESIGN OVERVIEW OF PRISM

We introduce the five key components of Prism– (1) Persistent Key Index on NVM, (2) Value Storage on SSDs, (3) Persistent Write Buffer (PWB) on NVM, (4) Scan-aware Value Cache (SVC) on DRAM, and (5) Heterogeneous Storage Index Table (HSIT) on NVM – as illustrated in Figure 2. These five components are tightly integrated to leverage the advantages of each storage device and compensate for their respective disadvantages.

4.1 Persistent Key Index on NVM

Prism manages a *Persistent Key Index*, which is an NVM-optimized range index that maps a key indirectly to a value location. A lookup operation for a key requires frequent small-size data access [14, 62, 90], so a byte-addressable low-latency media is appropriate. Since the space required for the Key Index grows as the data grows, placing the Key Index on NVM is a reasonable choice for scaling and cost-efficiency. In addition, thanks to the persistency characteristics of NVM, it is possible to avoid the expensive crash consistency mechanism, notably write-ahead logging on SSD, and excessive storage scan operations during the recovery process.

One key challenge in designing Persistent Key Index is achieving high multicore scalability. The problem is more severe in a heterogeneous storage system because, in the worst case, a single slow storage device could determine the overall scalability. Many prior studies [7, 46, 65], including KVell [57], choose the shared-nothing design, which partitions all data structures (*e.g.*, key index, cache) and storage spaces per CPU core to avoid synchronization overhead. However, the shared-nothing architecture is prone to suffer from load imbalance among shards, especially for skewed data (§7.6). We choose the following three techniques to deliver good multicore scalability in Prism:

Central Persistent Key Index. Prism uses a *central* Persistent Key Index to avoid the downsides of the shared-nothing approach which manages a per-shard index. With recent advances in index designs [50, 64, 66], a key index is no longer a scalability bottleneck. In this work, we employ PACTree [50], a state-of-the-art persistent range index. Note that our design is not dependent on a specific key index design, so Prism can replace it with any other range index.

Leveraging low-latency NVM. Storing values on SSD could significantly hamper the scalability of the Persistent Key Index because the inherent high latency of SSDs can increase the time taken to process a write-side critical section in the Persistent Key Index. Hence, we leverage low-latency NVM – *Persistent Write Buffer (PWB)* (§4.3) – for recently-written values, to prevent making SSD a bottleneck without compromising crash consistency.

Decoupling Persistent Key Index from value locations. Storage management tasks (*e.g.*, garbage collection, data migration, etc.) can hamper the scalability of the Persistent Key Index as well. Since a key index entry maintains a value location, it should be updated whenever the value location is changed. This typically requires locking to protect the Persistent Key Index from concurrent accesses even if the Key Index just locks the affected leaf node, resulting in limited multicore scalability. To avoid additional synchronization overhead caused by the storage management tasks, we decouple the Persistent Key Index from the value location by leveraging *Heterogeneous Storage Index Table (HSIT)* (§4.5), allowing values to be moved independently from the Key Index.

4.2 Value Storage on Flash SSDs

Values are usually larger than keys in size, so they account for most of the total storage space and require high storage bandwidth. Such high space and bandwidth requirements correspondingly match well with flash SSDs. Therefore we place the values on flash SSDs separately from the keys on NVM. The main challenges lies in (1) maximizing the SSD bandwidth while (2) minimizing latency and (3) CPU consumption. Achieving all three requirements is a challenging task. Batching more IO requests using asynchronous IO libraries (*e.g.*, libaio [2], SPDK [8]) increases bandwidth utilization, but it also significantly increases tail latency due to queuing effects. By batching fewer requests, latency can be reduced, but frequently issuing IO requests underutilizes the bandwidth and incurs high CPU overhead. To overcome the above challenges, we take different approaches for read and write:

Read operation (lookup, scan). Our target is low-latency reads from SSD while maintaining high SSD bandwidth utilization. We opportunistically adjust the IO batch size for read operations according to thread concurrency. When there are many concurrent read requests to SSD, Prism increases IO batch size for high bandwidth. On the other hand, Prism will reduce the IO batch size for low latency under low levels of concurrency. We discuss the read procedure in detail in §5.3. Besides, Prism also manages a DRAM cache (§4.4) to reduce accessing the SSD for read-hot data.

Write operation (insert, update, delete). We minimize write latency in the critical path and simultaneously maximize the SSD bandwidth off the critical path. Prism first writes the value to low-latency NVM – *Persistent Write Buffer (PWB)* (§4.3) – for immediate durability. Later, in the background, Prism coalesces the values on PWB and writes them back to Value Storage in a log-structured manner which is suitable for SSD [72]. Prism performs all writes to SSD asynchronously with a large batch size to reduce CPU overhead (§5.2).

4.3 Persistent Write Buffer (PWB) on NVM

Unlike traditional logging techniques, Prism writes the values only to the *Persistent Write Buffer (PWB)* on NVM. Prism manages a PWB for each thread to avoid synchronization from concurrent writes. The result is a shorter critical path, reducing write latency significantly and scaling the Persistent Key Index. Any value in the PWB is directly accessible from the Persitent Key Index through the *Heterogeneous Storage Index Table (HSIT)* (§4.5), so PWB plays the fast path for accessing recently written data.

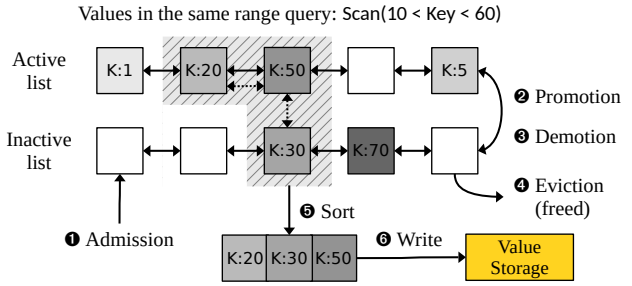


Figure 3: Scan-aware Value Cache (SVC). SVC uses 2Q LRU with an active and inactive list for cache eviction. It sorts and writes values in the same scan range (filled with a hatch pattern) to Value Storage when one of them are evicted. This increases spatial locality of values and improves the scan performance by reducing SSD IO.

Prism writes values to PWB in an append-only manner. Thus, PWB may contain multiple values for the same key (`v1'` and `v1"` for `k1` in Figure 2). In this case, Persistent Key Index always points to the latest value in PWB through the HSIT (❶ in Figure 2). Also, PWB provides crash consistency for values by leveraging the durability of NVM. As data gets written to PWB in an append-only manner, old data within PWB is not overwritten. Thus, guaranteeing data consistency in PWB is both easy and efficient (§5.5).

Reclamation for the PWB is triggered when its utilization reaches a watermark (50% in our evaluation). During reclamation, values in the PWB are asynchronously written to Value Storage on SSD, and the application thread utilizes the remaining space in PWB for further request processing, preventing the thread from blocking during reclamation. Thanks to the append-only write manner, when reclaiming PWB, Prism writes only the latest version of a value (*e.g.*, `v1"` in Figure 2) to Value Storage. This significantly reduces write traffic to the SSD. Additionally, since NVM guarantees outstanding endurance than flash SSD (§2.1), PWB helps to drastically extend the lifespan of SSDs and Prism.

4.4 Scan-Aware Value Cache (SVC) on DRAM

We present *Scan-aware Value Cache (SVC)*, which caches frequently accessed read-hot values in DRAM to mitigate high-latency SSD reads (*e.g.*, choose ❹ over ❸ in Figure 2). SVC does not maintain a separate index to lookup the cache, unlike prior work [52]. Instead, the cached value is directly accessible from the Key Index through HSIT (❹ in Figure 2).

Reducing CPU overhead for managing cache. Prism performs most cache management off the critical path in the background. Unlike conventional cache design serving both reads and writes, Prism segregates writes from the cache, and uses PWB. A value is admitted to SVC when it is neither found in the PWB nor the SVC upon reading (*i.e.*, only on reading the Value Storage on SSD). Upon reading a value from SSD, Prism makes the DRAM copy of the value immediately accessible by atomically updating HSIT in a lock-free manner. After that, it enqueues a request to add the cached value to an LRU list for victim selection. A background thread is in charge of managing the cache, and it processes the requests to add the newly cached value to the LRU list as well as to evict cached values from SVC. Thus, cache management does not happen in the critical path for Prism.

Efficient eviction policy. The SVC uses a 2Q LRU scheme [43] with an *active list* and an *inactive list* (see Figure 3). SVC adds the value to an inactive list when it first reads the value from SSD (❶). Upon accessing the cached value a second time, the value is promoted to the active list (❷). When the active list becomes too long, values from the tail of the active list are demoted to the inactive list (❸). Furthermore, when the cached values are about to go beyond the designated DRAM capacity, SVC evicts values from the inactive list (❹). When a value is evicted from SVC, it is logically deleted by disconnecting it from HSIT. Later, the SVC entry which contains the evicted value is physically freed after ensuring that no thread is accessing it using epoch-based reclamation (§5.4).

Accelerating scan operation. We repurpose SVC to speed up scan operations. Our log-structured Value Storage appends any values to the log, so values on SSD do not preserve spatial locality in the key space. Hence, a scan operation entails more SSD IO. In the worst case, each value in the scan range may reside on different SSD pages. To reduce SSD IO, SVC enhances the spatial locality of scan-intensive values by reorganizing them on SSD. When Prism performs a scan operation, it copies the values in the Value Storage to SVC (filled with a hatch pattern in Figure 3). When one of the value in the same scan range is evicted, SVC sorts and writes them together to the Value Storage (❺, ❻ in Figure 3). To efficiently identify the SVC entries corresponding to the same scan range, Prism chains these SVC entries in a doubly-linked list when scanned. Upon eviction of one of the entries from SVC, Prism traverses the doubly-linked list associated with the evicted entry that was formed during a prior scan operation. Hence, no additional lookup is necessary to find values in the same range in SVC. Conclusively, this way improves spatial locality of the values in the scan-heavy key ranges and reduces SSD IO for subsequent scan operations for the key range. As discussed, a background thread is responsible for performing these tasks.

4.5 Heterogeneous Storage Index Table (HSIT) on NVM

In Prism, values are scattered across heterogeneous storage devices (NVM, SSD, and DRAM), and their location is subject to change over time due to foreground and background activities, such as write buffering, reclamation, and caching. The *Heterogeneous Storage Index Table (HSIT)* is an indirection table, which manages the value location across heterogeneous storage devices. Although the indirection technique has been used in previous studies [59, 60], we exploit HSIT as a foundation of Prism's key innovations, including 1) cross-media concurrency control, 2) lightweight crash consistency, and 3) fast recovery.

Conceptually, HSIT is an array whose entry consists of value addresses in PWB, Value Storage, and SVC, which we call *forward pointer*. The Persistent Key Index maps a key to an array index of HSIT (§4.1), and HSIT always points to the up-to-date value of the corresponding key (see Figure 6). We pack the three forward pointers into 16 bytes since a value can exist in only either PWB or Value Storage. Note that, Prism caches values into SVC from Value Storage, not from PWB, as it can access values on PWB with

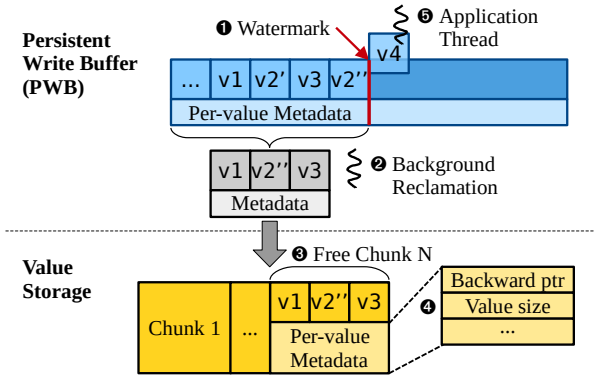


Figure 4: Asynchronous bandwidth-optimized write in Value Storage. Value Storage consists of multiple chunks. When PWB utilization hits the watermark (❶), the up-to-date values (v1, v2", and v3) in the PWB (❷) are asynchronously written to a chunk in Value Storage (❸). Each chunk is fixed-size and stores values with their metadata – each value's backward pointer (*i.e.*, a pointer to an HSIT entry) and size (❹). While reclaiming PWB, an application thread can still write values (v4) to the rest of the PWB (❺).

DRAM-like latency. We place HSIT in NVM since NVM provides low latency, especially for small-sized data, and lightweight crash consistency.

Besides holding value locations, HSIT plays a central role in cross-media concurrency control and crash consistency. We propose a *backward pointer-based crash consistency* mechanism to efficiently guarantee cross-media crash consistency without relying on heavyweight logging. A value in PWB or Value Storage embeds a *backward pointer* to its HSIT entry (see Figure 6). If the backward pointer embedded in the value and the forward pointer in the HSIT entry are *well-coupled* (*i.e.*, they refer to each other), the value is valid and up-to-date. Using this property, Prism efficiently achieves concurrency control, crash consistency, and fast recovery (§5.4, §5.5).

5 DETAILED DESIGN OF PRISM

5.1 Organization of Value Storage

Value Storage is a log-structured store on an SSD containing values. It divides the space into *fixed-size chunks* for space management (❸ in Figure 4). A chunk contains multiple values and metadata associated with each value (❹) – a *backward pointer* and *value size* – for efficient crash recovery. To maximize the write bandwidth of SSD, Prism performs writes in a chunk granularity whose size is 512 KB because large sequential writes are suitable for SSD [72]. Each chunk has a *validity bitmap* in DRAM, where each bit represents whether the corresponding value is valid (*i.e.*, up-to-date) or not (*i.e.*, already outdated). The bitmap is updated whenever a value on the chunk is written to a new location, either on PWB or in another chunk. Since the bitmap is easily reconstructed through HSIT when starting Prism (see §5.5), Prism places and manages the validity bitmaps in DRAM. Meanwhile, Prism accesses Value Storage through an asynchronous IO interface, `io_uring` [3], to dynamically batch IO requests and also manages one Value Storage per SSD to utilize aggregated SSD bandwidth by concurrently accessing multiple Value Storages.

5.2 Asynchronous Bandwidth-Optimized Write

Asynchronous reclamation of PWB. When the PWB space usage crosses the watermark (❶ in Figure 4), Prism triggers reclamation for the PWB. A background reclamation thread first scans the PWB and collects up-to-date, live values (❷). Prism checks for each value if the backward pointer on PWB and the forward pointer on HSIT refer to each other. If so, we call the value on PWB is well-coupled. Note that while Prism requires two NVM reads at this step, it does not give a negative impact on total performance, because reclamation happens in the background and NVM provides low access latency.

Since Prism updates an HSIT entry on every write, the well-coupled value is an up-to-date and live value (*e.g.*, `v2"` over `v2'`). Prism writes the well-coupled, live values (`v1`, `v2"`, and `v3`) into the Value Storage asynchronously chunk by chunk (❸). Once the writing completes, Prism updates the forward pointers in the HSIT entries to point to the new value location on SSD. It also updates the validity bitmap denoting that the newly written values on SSD are valid. Upon updating the HSIT entries, the newly written values are accessible from the Persistent Key Index.

Multiple threads can concurrently write to their designated chunks on the same Value Storage. The reason is that allocating a free (empty) chunk is the only critical section. Upon successful allocation of the chunk, the critical section ends and the thread independently writes the reclaimed values to each assigned chunk. Therefore, no race condition exists when multiple threads in Prism write to the same Value Storage. When there are multiple Value Storages (on multiple SSDs), Prism randomly chooses one of the idle Value Storages, which do not have in-flight requests on asynchronous IO queues. This way, Prism aggressively utilizes the high write bandwidth of SSD.

Garbage collection in Value Storage. When the number of free chunks reaches below a threshold (*i.e.*, the Value Storage utilization exceeds a watermark), Prism triggers the garbage collection on Value Storage. Through garbage collection, Prism secures free chunks by merging *live (i.e., up-to-date) values* in two or more in-use chunks, called *victim chunks*, so they can continue to serve writes from the PWB. Unlike prior work [15, 22, 83], Prism decides whether a value is garbage or not by checking the validity bitmap without expensive key index traversal. Prism uses a greedy policy in choosing victim chunks to be collected. A background garbage collection thread chooses the chunks that have the smallest number of live values (by checking the validity bitmap) as victims. The live values in the victim chunks are copied to a new empty chunk. Once the migration completes, Prism updates the corresponding HSIT entries to point to the new value location and updates the validity bitmaps accordingly. After the HSIT entries are updated, victim chunks will not have any new accesses, allowing them to be recycled as free chunks for future writes to the Value Storage. Note that Prism performs garbage collection within the same Value Storage.

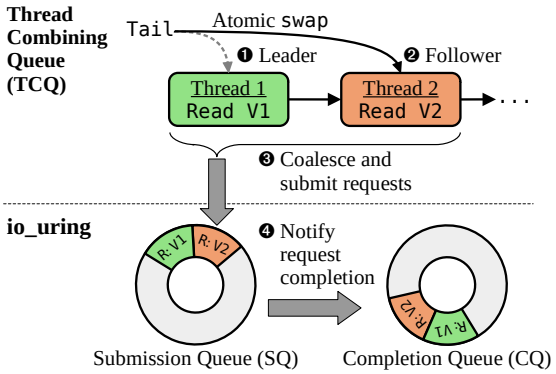


Figure 5: Opportunistic thread combining for optimized read in Value Storage. The leader thread (❶) dynamically coalesces current read requests of other threads (follower, ❷). It submits the coalesced requests to Submission Queue of Value Storage for asynchronous IO operation (❸). Once the IO requests are processed, the OS kernel posts completion messages to the Completion Queue (❹).

5.3 Opportunistic Thread Combining for Optimized Read

When a requested value does not exist in either on SVC or on PWB, Prism has to read the value from Value Storage. The IO batch size in our asynchronous IO approach determines the bandwidth, latency, and CPU overhead. With a large IO batch, we can achieve high read bandwidth and low CPU overhead at the expense of high latency. On the other hand, a small IO batch allows for low read latency while suffering from low read bandwidth and high CPU overhead. Hence the primary challenge is determining the right IO batch size dynamically to achieve low latency, high bandwidth, and low CPU overhead.

To address the challenge, we propose an *opportunistic thread combining* scheme for Value Storage read operations, as illustrated in Figure 5. We use `io_uring` [3] in the Linux kernel for asynchronous IO. Similar to other asynchronous IO frameworks, `io_uring` exposes two queues, namely the *Submission Queue (SQ)* and *Completion Queue (CQ)*. It does not block threads after the submission (*i.e.*, asynchronous IO). A SQ/CQ pair is responsible for a single Value Storage.

Prism dynamically determines the IO batch size based on the number of concurrent reads requested from application threads. More concurrent reads from application threads mean a larger IO batch size for higher read bandwidth and lower CPU overhead. With fewer concurrent reads, a smaller batch size leads to lower read latency. Prism manages a *Thread Combining Queue (TCQ)*, which lines up the concurrent threads requesting read operations to Value Storage. The behavior of arranging incoming threads in TCQ is inspired by the MCS queue lock [70]. A thread is first enqueued into TCQ using an atomic `swap` operation on the TCQ `Tail`. If `Tail` is `null` after the `swap` operation, the thread is at the head of TCQ and takes the *leader* role (❶ in Figure 5). Otherwise, it becomes a *follower* (❷) and passes its read request to the leader. The leader coalesces its own and followers' read requests by traversing the TCQ. The coalesced requests are submitted to SQ (❸) when there is no more followers in the TCQ or it reaches to the coalescing limit (*i.e.*, queue depth: 64). As soon as the follower's request is coalesced by the leader, the follower returns immediately. The completion of the batched read requests is notified later to CQ (❹), and a background completion thread polls the CQ to check whether there are completed IO requests.

In summary, Prism opportunistically combines reads from multiple threads to a single read operation to aggressively utilize the bandwidth and hide the high latency of SSD.

5.4 Cross-Media Concurrency Control

Atomic visibility of a value. In Prism, the Persistent Key Index and HSIT are at the center of concurrency control. These centralized components enforce visibility for a value as every value access goes through them. A write operation in Prism is not visible until after writing the value to the storage media and updating the HSIT and Persistent Key Index. In other words, if an HSIT entry is not accessible through the Persistent Key Index, the value will not be visible to other threads. This implies that other threads will not see any partial writes until the value gets completely written to the storage media. Thus, updating an HSIT entry is a linearization point [37].

Durable linearizable update of an HSIT entry. To guarantee durable linearizability, unpersisted data should not be visible. Accordingly, an update operation in Prism is not visible until the corresponding HSIT entry gets updated. Prism finishes an update request by first making the fresh value persistent, followed by updating the value's address in the HSIT. For insert operation, after performing the above steps, Prism addtionally updates the Key Index to reflect the newly inserted HSIT entry.

In this protocol, when Prism updates an HSIT entry (*e.g.*, forward pointer), it uses atomic instructions (*e.g.*, `compare-and-swap` or `CAS`). This lock-free approach can provide high multicore scalability by preventing HSIT updates from becoming a scalability bottleneck. However, an atomic update of a pointer does not necessarily imply that the updated pointer is persistent on NVM. The update may have only reached the volatile CPU cache and can be lost upon a power failure or crash. To guarantee atomicity and durability of a pointer update, we update the forward pointer in HSIT using the *flush-on-read* technique [26, 84, 85]. Prism encodes a dirty bit to an unused bit into a forward pointer in HSIT entry. A writer records the address of updated value with a dirty bit. The writer flushes the pointer to guarantee durability, and then clears the dirty bit using an atomic 8-byte `CAS` operation. When a reader detects that a forward pointer is dirty (*i.e.*, the pointer is written but not flushed yet), it flushes the pointer first on behalf of the writer, then turns off the dirty bit. In this way, Prism guarantees durable linearizability [42].

No write/write conflicts. All writes in Prism first go to a per-thread PWB, so there are no write/write conflicts. Thus, insert and update operations in Prism follow the concurrency control of the Persistent Key Index.

Safe reclamation of a deleted value and HSIT entry. For delete operations, Prism provides concurrent access to the HSIT, thus making it essential to reclaim HSIT entries safely. A deleted HSIT entry is added to the free list, a linked list for managing freed entries.

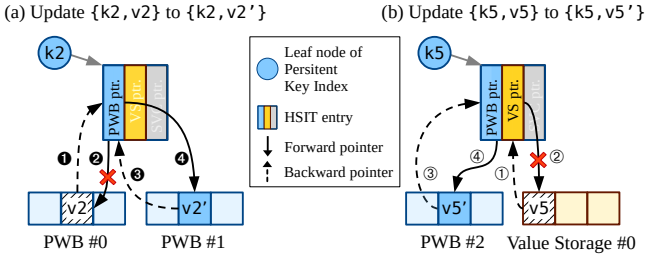


Figure 6: Crash consistent update of values on PWB and Value Storage with HSIT. Prism first writes a value with a backward pointer (❸,③), and then it updates its forward pointer (❹,④), invalidating the old forward pointer (❷,②). Prism efficiently guarantees cross-media crash consistency with our pointer update protocol and append-only PWB write policy.

For reclaiming HSIT entries, we use *epoch-based reclamation* [36, 48, 51, 68]. An epoch is a period such that all threads have finished their current operations from begins. Prism waits for two epochs: The first epoch ensures that no new thread accesses the deleting HSIT entry. The second epoch guarantees that all the references from the previous epoch have completed their access. Thus, the HSIT entry is not accessible after two epochs and can be reclaimed.

5.5 Cross-Media Crash Consistency and Recovery

Crash consistency using HSIT. We assume that the Persistent Key Index ensures its own crash consistency, so Prism only needs to guarantee the crash consistency between the Persistent Key Index and values on PWB and Value Storage. Prism efficiently achieves cross-media crash consistency using HSIT and PWB, as illustrated in Figure 6. Prism first writes a value on PWB with an embedded backward pointer (❸,③) followed by updating a forward pointer in HSIT (❹,④). Once the forward pointer is updated to point to the new value (`v2'`, `v5'`), the old value (`v2`, `v5`) becomes *ill-coupled* (❶ vs. ❹, ① vs. ④). If a crash happens after writing a value, but before updating the forward pointer, the written value is not reachable from the HSIT. Also, suppose a crash happens after writing a value and updating the forward pointer, but before we persist the forward pointer. In this case, the forward pointer is not persisted, so after a restart, the forward pointer still points to the old value, and the newly written value is not reachable. Prism determines whether a value is unreachable by comparing its forward/backward pointers.

Fast recovery. As described in Figure 6, Prism knows where the valid value is stored through only HSIT information. For recovery, Prism first has to perform a full scan of the Persistent Key Index to find reachable HSIT entries. Then, from those HSIT entries, Prism finds out the PWB and Value Storage entries which contain valid values. For PWB, it is sufficient to determine whether the backward pointer and the HSIT entry are well-coupled or not. Meanwhile, for Value Storage, Prism reconstructs the validity bitmap of each chunk of Value Storage and nullifies all pointers in HSIT pointing to SVC. Prism performs the recovery procedure concurrently for randomly partitioned key ranges from the key index. In sum, our cross-media crash consistency mechanism makes Prism can efficiently recover without relying on heavyweight logging.

Table 1: Configurations of key-value stores for evaluation.

Key-value Stores	DRAM Cache	NVM Buffer	Total Cost
Prism	20 GB	16 GB	$170
KVell [57]	32 GB	None	$170
MatrixKV [89]	26 GB	8 GB	$170

Table 2: YCSB workload characteristics.

Workloads	Characteristics
LOAD	Write-only: 100% Inserts
YCSB-A	Write-intensive: 50% Updates and 50% Reads
YCSB-B	Read-intensive: 5% Updates and 95% Reads
YCSB-C	Read-only: 100% Reads
YCSB-D	Read-latest: 5% Updates and 95% Reads
YCSB-E	Scan-intensive: 5% Updates and 95% Scans

6 IMPLEMENTATION

Storage IO interface. We leverage `io_uring` [3] for efficient asynchronous IO to the Value Storage. It is well-known that the `io_uring` reduces the CPU overhead and boosts IO performance by minimizing expensive system calls. Prism uses the XFS filesystem and opens Value Storage files with the `O_DIRECT` flag to enable concurrent disjoint accesses [71].

Persistent Key Index. We choose PACTree [50] for our Key Index for its scalability and performance. PACTree is a persistent index structure that uses asynchronous updates to maximize concurrency. PACTree stores the keys in a packed manner to save the space and bandwidth consumption of NVM. Recall that in Prism, its Key Index can be replaced by another persistent index that supports scan operations because Prism has no dependency on PACTree.

7 EVALUATION

7.1 Evaluation Methodology

Hardware environment. We use a two-socket Intel Xeon machine. Each socket has 20 CPU cores, six 128GB Intel Optane DIMMs, and 96GB DRAM. Storage devices consist of eight Samsung 980 PRO 1TB SSDs with two NVMe RAID Controllers HighPoint SSD7103. For fair comparison, the competitors employ NVM and SSDs in the form of RAID-0 through mdadm [4] and dm-strip [6] to fully exploit the hardware. Also, we allocated their hardware resources at the same cost levels as described in Table 1.

Configuration of key-value stores. For Prism, we created eight Value Storages, one per SSD. Each Value Storage has its own thread for asynchronous IO, handling its own IO queues for request batching (queue depth of 64 as KVell [57]).

We compare Prism with three LSM-tree based key-value stores – MatrixKV [89], RocksDB-NVM [33], and SLM-DB [45] – and one of sharding-based key-value store, KVell [57]. For MatrixKV which leverages NVM and SSD, we set 8 GB NVM space for L0 in the LSM tree, as shown in its paper. RocksDB-NVM is a modified RocksDB storing all SSTables and WAL files in NVM. Certainly, its storage cost spends much higher than Prism, but we use RocksDB-NVM as a reference point showing the maximum performance of LSM-tree

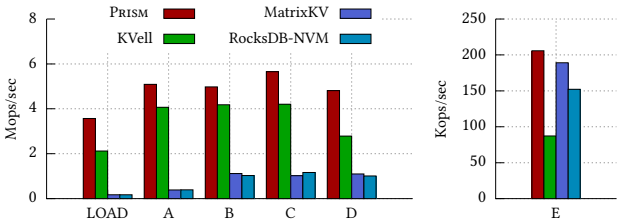


Figure 7: Throughput comparison for YCSB workloads.

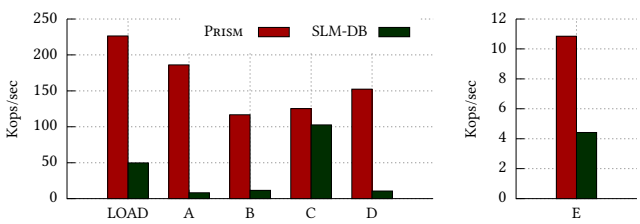


Figure 8: Throughput of Prism and SLM-DB.

Table 3: Latency comparison (μs) for YCSB workloads.

Workloads	Latency	Prism	KVell	MatrixKV	RocksDB-NVM
YCSB-A	Average	44	231	123	141
	Median	2	152	114	128
	99%	145	1262	244	247
YCSB-C	Average	12	49	35	27
	Median	1	44	28	23
	99%	128	185	148	93
YCSB-E	Average	325	456	175	218
	Median	270	484	92	128
	99%	808	1215	1138	1132

Table 4: Latency (μs) of Prism and SLM-DB.

Workloads	Latency	Prism	SLM-DB
YCSB-A	Average	30	122
	Median	2	18
	99%	90	1363
YCSB-C	Average	25	10
	Median	1	4
	99%	96	42
YCSB-E	Average	231	233
	Median	229	89
	99%	796	1394

based approaches. SLM-DB uses a small portion of NVM (64MB) for memtable and the rest for its index structure. Also, the open-sourced SLM-DB only supports single-threaded execution. For the fair comparison with SLM-DB, we also configured Prism to use 64 MB of DRAM (SVC) and 64 MB of NVM (PWB) and ran experiments on a single thread. Lastly, KVell uses DRAM and SSD (without NVM). We configured KVell's DRAM cache size to 32 GB so that KVell and Prism have the same cost. For KVell, we created 16 injector threads to issue queries and three worker threads per SSD with IO queue depth of 64. To control the DRAM usage for fair comparison, we used each key-value store's DRAM cache and turned off the page cache in OS with the `O_DIRECT` flag except for SLM-DB. Since SLM-DB does not support the direct IO, it uses the page cache in OS and consumes more memory. For all other configuration parameters, if not mentioned, we used the same parameters described in their papers.

Workload configuration. We use YCSB [24] workloads with Zipfian distribution (Zipfian coefficient: 0.99) (see Table 2) as real-world workloads with skewed distribution [18]. We set the size of a key-value item to 1 KB. We load 100 million key-value pairs in random order and perform 100 million operations for all workloads except workload E. For Workload E, we perform 20 million scan operations with an average scan length of 50. Meanwhile, due to the instability of SLM-DB, we index 8 million key-value items and conduct 2 million operations for SLM-DB evaluation.

7.2 Prism vs. MatrixKV and RocksDB-NVM

Write-intensive workload. Figure 7 and Table 3 show the throughput and latency comparison with total 40 threads, respectively. Prism outperforms MatrixKV and RocksDB-NVM by up to 13.1× with up to 3.2× lower average latency under Workload A. Even when we configured Prism to use the identical DRAM cache (26GB) and NVM buffer (20GB) sizes with MatrixKV, Prism outperforms MatrixKV by up to 10.8×. Even with NVM, MatrixKV and RocksDB-NVM still suffer from expensive compaction operations. In contrast, Prism does not require level-compaction. Furthermore, Prism's PWB, being a per-thread write buffer, avoids contention among threads and coalesces small writes into large block sizes, resulting in superior performance.

Read-intensive workload. Prism shows 4.8–5.5× higher throughput than LSM-tree based key-value stores in read-intensive Workloads B, C, and D because managing values without keys in the SVC results in more efficient caching. Moreover, the lookup operation in Prism efficiently uses CPU resource, not traversing multiple levels to find a key-value pair as done in LSM-tree based approaches. As shown in Table 3, Prism's latency is lower than MatrixKV and RocksDB-NVM primarily due to efficient caching in the SVC and asynchronous low-latency reads from the Value Storage. Meanwhile, MatrixKV and RocksDB perform IO submission and completion synchronously unlike Prism, so they suffer from high latency.

Scan-intensive workload. In Workload E (95% scan and 5% write operations), Prism outperforms MatrixKV and RocksDB-NVM in throughput (up to 1.4×) and tail latency. SVC writes evicted values in a range query to the same chunk in Value Storage. This makes future scan operations more efficient. Also, as MatrixKV and RocksDB-NVM are hierarchical storage systems, a scan operation may traverse every level of the LSM-tree to find the values for the keys in the range query. This traversal overhead deteriorates the scan operations.

7.3 Prism vs. KVell

Write-intensive workload. In Workload A, Prism outperforms KVell by 1.3× while delivering 8.7× lower tail latency. This is because Prism leverages PWB on NVM to reduce latency and SSD write amplification. KVell delivers good performance via batching IO requests, but queuing amplifies and worsens tail latency significantly (see Table 3). Moreover, KVell has to read-modify-write to update data when the data is not cached on DRAM. Thus, it causes more frequent accesses to SSD, thereby increasing latency significantly.

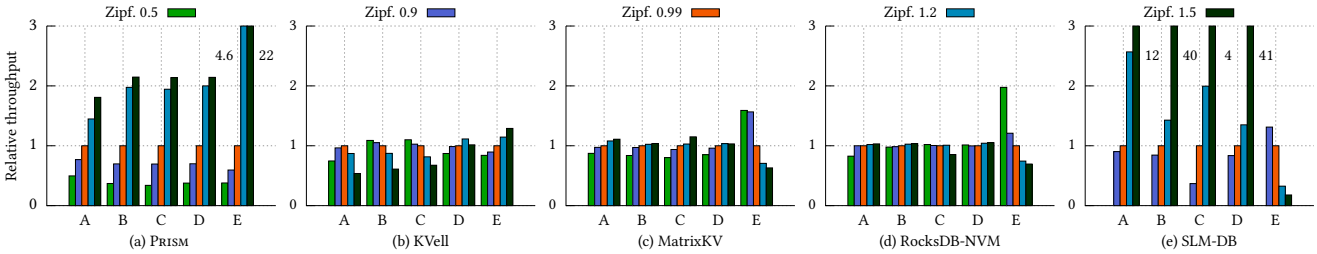


Figure 9: Relative throughput of key-value stores with varying Zipfian coefficients from 0.5 to 1.5. (Normalized to 0.99)

Read-intensive workload. For Workloads B and C, Prism outperforms KVell by 1.2× and 1.3×, respectively. This performance gain comes from our SVC design that caches individual values while KVell's cache manages its data in page granularity (4KB). Also, some IO workers in KVell can become the performance bottleneck under high data skew as KVell partitions the entire key space using hashing. Furthermore, KVell always enqueues requests to worker threads for IO batching even in the case the requested data is already cached in DRAM. The queuing effect amplifies the latency for cached data (see Table 3). In contrast, Prism directly accesses the SVC on DRAM via HSIT. The tail latency of Prism is also lower than KVell by 1.5× in Workload C. Prism submits read requests to the SSDs even when the IO queue of each thread is not full, thereby minimizing the idle time of SSDs. Unlike Prism, KVell's worker threads not only submit IO requests to storage devices but also traverse the indexes, which adds up to the latency. In Workload D, Prism outperforms KVell by 1.7×. Prism has a high probability of reading data from the PWB as it handles write requests on the PWB.

Scan-intensive workload. Prism provides better throughput and latency than KVell by 2.3× and 1.5×, respectively, as shown in Workload E of Figure 7 and Table 3. KVell incurs more IOs to the SSD for a given key range. However, Prism's SVC efficiently merges values in the same key range into the same chunk, reducing SSD IO.

7.4 Prism vs. SLM-DB

We ran SLM-DB and Prism only in a single-threaded environment because the open-source version of SLM-DB does not support multi-threading. Note that SLB-DB consumes more memory than Prism (and all other tested key-value stores) because it does not support direct IO (`O_DIRECT`) so it leverages the page cache in OS.

Write-intensive workload. Prism outperforms SLM-DB by up to 22.7× in write-intensive workloads, as depicted in Figure 8 and Table 4. Although SLM-DB has a single-level storage layer, it still requires compaction operations from memtable to SSD that degrade its performance. Also, it lacks design considerations for asynchronous IO batching to exploit the high bandwidth of SSDs, thereby not utilizing the potential of heterogeneous storage devices.

Read-intensive workload. For YCSB Workloads B, C, and D, Prism achieves an order of magnitude higher throughput than SLM-DB, up to 14.4× in Workload D. This is because Prism can handle read requests within a short critical path in NVM. For Workload C, SLM-DB shows lower average and tail latency than Prism because SLB-DB uses the OS page cache (not supporting direct IO) so it consumes more memory (*i.e.*, not apple-to-apple comparison with Prism). Even though SLM-DB leverages the OS page cache, Prism shows comparable performance to SLM-DB.

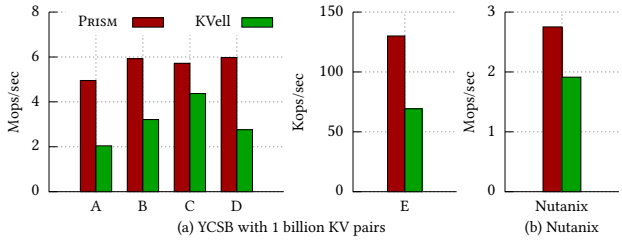


Figure 10: Performance in YCSB with a 1-billion KV pairs and Nutanix production workloads.

Scan-intensive workload. Prism delivers throughput up to 2.5× that of SLM-DB in Workload E, as shown in Figure 8. Our SVC reduces the number of IOs issued to Value Storage.

7.5 Performance under Other Workloads

YCSB workloads with 1 billion KV items. We also conducted performance evaluations using larger YCSB workloads containing a 1 TB dataset (1B KV pairs), as shown in Figure 10(a). As mentioned earlier, Prism uses a smaller DRAM cache than KVell, which consists of only DRAM and SSDs to ensure identical hardware costs for both systems. Despite it, Prism achieves 1.3× higher performance than KVell under the YCSB-C workload. The efficacy of our SVC and opportunistic thread combining are the key contributors on improving read performance. Overall, the experimental results show that Prism outperforms KVell by up to 2.42× for this workload.

Nutanix production workloads. Besides YCSB workloads, we compared Prism and KVell with the production workloads from Nutanix, as presented in Figure 10(b). This workload tends to be rather write-intensive: 57% Updates, 41% Reads, and 2% Scans. In this case, Prism shows 1.44× higher performance.

7.6 Understanding Prism Performance

Impact of data skewness. A recent study shows that real-world workloads exhibit strong access skew [18, 87]. Figure 9 shows the performance of Prism and other key-value stores with varying data skewness. The relative performance is normalized to the performance when the Zipfian coefficient is 0.99. Prism effectively manages the read/write hot data by leveraging PWB and SVC, resulting in throughput improves as skewness increases in all YCSB

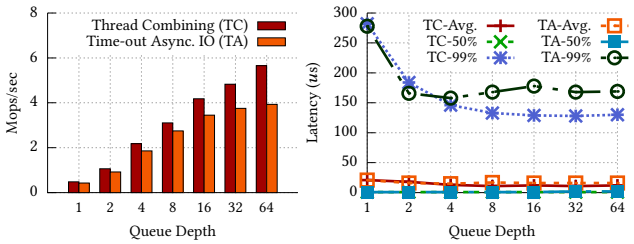


Figure 11: Impact of PRISM's opportunistic thread combining for optimized read with varying queue depth.

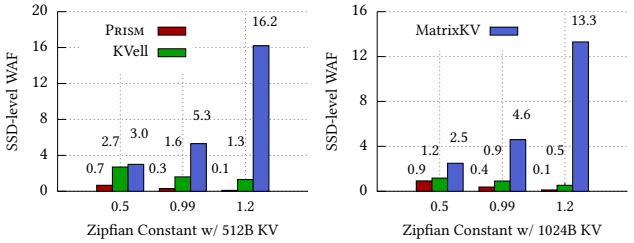


Figure 12: Write amplification to SSD with varying data skewness.

workloads. We also investigate performance trends of LSM-based key-value stores that adopt a hierarchical structure. They show better performance as data skewness increases, as shown in Figure 9. This performance gain comes from the increased chance of accessing data in the memtable or their internal block cache within the memory layer. KVell's throughput drops as skewness increases load imbalance and creates a few hot spots. Hence, some worker threads are overloaded and become the performance bottleneck under skewed data access. This load imbalance is an inherent problem of partitioning-based architectures [17, 63].

Opportunistic thread combining for read. We evaluate the effectiveness of our thread combining technique using YCSB Workload C while varying the queue depth (QD, the coalescing limit). Figure 11 presents the throughput and latency as QD varies in two different settings; One is using our thread combining technique (§5.3, abbreviated to TC) and another is using timeout-based asynchronous IO processing (abbreviated to TA). TA waits for subsequent read requests for a certain period (100μs in this evaluation) and submits the requests to storage if there is no more incoming request. The experimental results show that the performance gap between TC and TA gets larger as QD increases. Also, thread combining with QD of 64 delivers up to 11.7× higher throughput and 1.9× lower response time than when using a single QD. This confirms PRISM's thread combining can handle I/O requests from multiple threads at once with high SSD IO utilization.

Write amplification (WAF). We measure the write amplification in SSDs for updating a 100GB dataset with variable sizes of key-value pairs (512B and 1KB). Figure 12 shows that PRISM has the lowest SSD-level write amplification as PWB absorbs small IOs and merges them into large chunk-sized IOs to the SSD. MatrixKV has high write amplification, up to 162× of PRISM, due to compaction operations of the LSM-tree. KVell also shows high write amplification, up to 13× of PRISM, because KVell performs IO operations in page granularity. As data skewness increases, write amplification

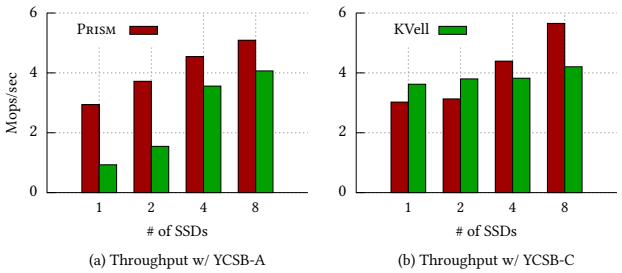


Figure 13: Throughput with varying the number of SSDs.

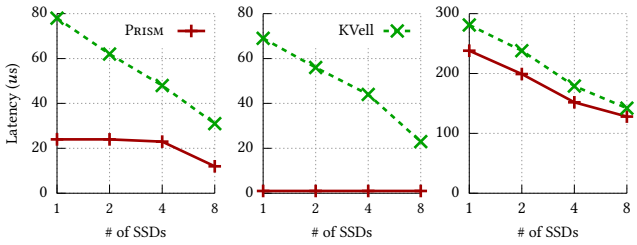


Figure 14: Latency (μs) with varying the number of SSDs.

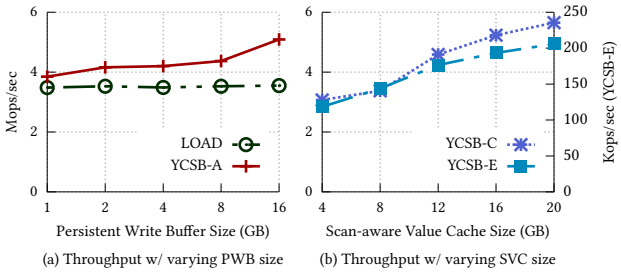


Figure 15: Performance impact of varying PWB/SVC sizes.

decreases for PRISM and KVell due to increased opportunities to coalesce IO requests for the same data. In contrast, MatrixKV shows higher write amplification as the skewness increases due to the compaction in LSM-tree.

Number of SSDs. We measured the throughput of PRISM and KVell with write-intensive Workload A and read-intensive Workload C while varying the number of aggregated SSDs in Figure 13. In the write-intensive Workload A, PRISM provides higher throughput than KVell irrelevant to the number of SSD attached, thanks to PRISM's PWB and scalable centralized components. Even in the read-intensive Workload C, PRISM delivers better throughput and latency. Only in the case of the number of SSDs being less than 4, KVell provides higher throughput than PRISM as shown in Figure 13(b). This is because KVell employs special threads that inject IO requests into the queue to batch read requests more aggressively. Although, KVell offers better read throughput in the case of a small number of SSDs, note that PRISM which exploits opportunistic thread combining for handling read requests, always provides lower latencies than KVell, as shown in Figure 14.

Size of Persistent Write Buffer. The size of PWB is closely related to the write performance of PRISM as every value is written first to the PWB. As shown in Figure 15(a), in LOAD workload,

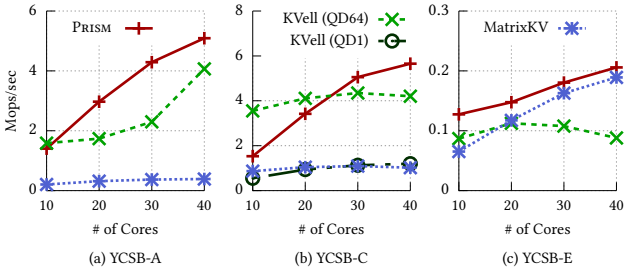


Figure 16: Comparison of multicore scalability.

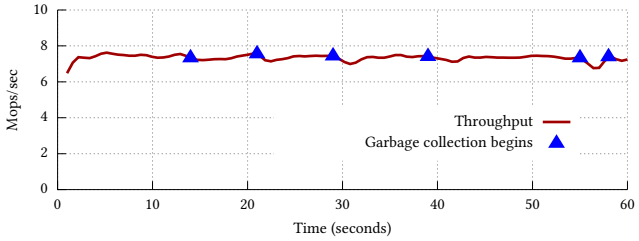


Figure 17: Impact of garbage collection for YCSB-A.

we can find out PRISM's performance shows stable because PRISM successfully migrates values from PWB to Value Storage using the background thread. In Workload A, as the size of the PWB increases, the throughput also increases because PWB absorbs more write requests. Considering the large capacity of NVM, modern heterogeneous storage systems can sufficiently carry out higher write performance.

Size of Scan-aware Value Cache. We evaluated the performance of lookup and scan operations while varying the size of SVC. Figure 15(b) shows that a larger SVC improves performance. Even with 4GB SVC (only 20% of 20GB SVC), PRISM shows 55% read and scan performance compared to 20GB SVC. We confirmed that PRISM's value-granule caching is more effective than page-granule caching used in prior work [57].

Multicore scalability. We measure the multicore scalability of PRISM with varying the number of cores. PRISM scales near linearly as the number of cores increases in all workloads due to its efficient concurrency control and lightweight data consistency mechanisms, as shown in Figure 16. Note that, in Workload C, PRISM always shows lower latency than KVell in all settings (either QD is 64 or 1).

Impact of garbage collection in Value Storage. Garbage collection (GC) in PRISM reclaims free space for Value Storage when there is no enough free space in each Value Storage. In our evaluation, GC does not significantly affect the PRISM's performance, as shown in Figure 17 as PRISM supports non-blocking access to values in Value Storage via HSIT. Moreover, GC is performed in each Value Storage independently.

Impact of individual techniques. We conduct a performance breakdown to evaluate the impact of each technique proposed. Our asynchronous bandwidth-optimized writes (§5.2) deliver up to 23% performance improvement for write-intensive workloads. Furthermore, it minimizes the performance fluctuations due to garbage collection, as shown in Figure 17. Using opportunistic thread combining (§5.3) gains 11.7× performance improvement in read-only workloads. SVC enhances the lookup and scan throughput by up to 9.6× and 4.4×, respectively. Moreover, its accelerated scan operation (§4.4) derives a performance improvement of nearly 10% rather than not being used.

Size of NVM space. We measure the space overhead of our NVM components: Persistent Key Index and HSIT. For 100 million key-value pairs, PRISM requires a total of approximately 5.4 GB of NVM space. We believe it is a reasonable size considering the large capacity of NVM.

Recovery time. We measured the recovery time in PRISM and KVell. We injected system crashes after inserting 100 GB of a dataset, similar to previous studies [56, 58]. PRISM spends about 6.9 seconds to recover all key-value pairs, while KVell which consists of DRAM and SSDs takes 10.4 seconds. KVell needs to scan the entire SSD, so recovery time can seriously deteriorate depending on the performance and number of storage devices.

8 DISCUSSION

We found that NVM has many characteristics that are helpful for designing heterogeneous storage systems. It provides higher capacity than DRAM, very high endurance compared to SSD, and low latency. Through the comparison evaluations between PRISM and DRAM-SSD configuration (*i.e.*, KVell), we demonstrate the usefulness of NVM. As seen in §7, PRISM outperforms conventional systems significantly for write-intensive workloads as PWB absorbs writes to the SSD. Also, PRISM shows significantly lower SSD WAF than KVell. That is, using NVM can significantly extend the span of SSD lifetime. Lastly, NVM enables fast recovery without scanning the entire SSD, thanks to the persistency characteristics of NVM, resulting in PRISM 's recovery being much faster than KVell.

We believe the lessons from PRISM apply to other emerging storage media. Practically, today's storage media is more diverse than we anticipate: (1) QLC SSD, ZNS SSD, and remote flash for compute-storage disaggregation – high bandwidth and low capacity cost, (2) CXL-based (battery-backed) NVM – low access latency, byte-addressability, and non-volatility. Lastly, (3) Samsung's recently released Memory-Semantic CXL SSD – byte-addressable and block-addressable access with CXL.mem and CXL.io, respectively. Considering the modern storage landscape, we set out to answer the questions: how to design an efficient heterogeneous storage system while utilizing the low latency byte-addressable storage and how to hide the high latency of slow storage devices and software overhead by leveraging concurrent access. Above all, we present novel approaches to cross-media concurrency control, crash consistency, and recovery, enabling efficient amalgamation of the diverse storage devices within a single system.

9 CONCLUSION

We presented PRISM, a key-value store built for modern heterogeneous storage devices. PRISM copes with the increasing diversity of storage devices and exploits the capabilities of each storage device (*e.g.*, low latency of NVM, high bandwidth of SSD). At the center of PRISM's design lies a careful interplay between a low latency storage device and a high bandwidth storage device that enables PRISM to achieve the best of both worlds. PRISM maintains multicore scalability and crash consistency across heterogeneous storage

devices using efficient concurrency control and crash consistency protocols. Our extensive evaluation shows significant performance improvements against the state-of-the-art techniques for various real-world workloads.

10 DATA-AVAILABILITY STATEMENT

The data that support the findings of this study are publicly available in Zenodo at https://doi.org/10.5281/zenodo.7215748 [9] and Github at https://github.com/cosmoss-jigu/prism [10].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and feedback. This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2022R1F1A1070065 and NRF-2021R1A6A3A 03046359) and by the National Science Foundation under the grants CCF-2153748.

REFERENCES

[1] 2015. MariaDB. https://mariadb.org/.
[2] 2018. libaio. https://pagure.io/libaio.
[3] 2019. Efficient I/O with io uring. https://kernel.dk/io_uring.pdf.
[4] 2021. mdadm. https://raid.wiki.kernel.org/index.php/A_guide_to_mdadm.
[5] 2021. Samsung teases a PCI 5.0 SSD that can hit 13,000 MB/s read speeds. https://www.engadget.com/samsung-teases-pci-50-ssd-that-can-hit-13000-m-bs-read-speeds-070308575.html?src=rss&fbclid=IwAR2wZDeVVwFY8Yfa5-gJTHr5DT-DwBH-zTA1yRhV0JtIc3nlXs8iwKc5jng.
[6] 2022. Device Mapper. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/striped.html.
[7] 2022. MongoDB. https://www.mongodb.org/.
[8] 2022. Storage Performance Development Kit. https://spdk.io/doc.
[9] 2023. cosmoss-jigu/prism: v0.9.0-asplos23-ae. https://doi.org/10.5281/zenodo.7215748.
[10] 2023. Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. https://github.com/cosmoss-jigu/prism.
[11] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Ontario, Canada, 353–369.
[12] Anandtech. 2018. Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here! https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here
[13] Apache. 2023. Welcome to Apache HBase™. https://hbase.apache.org/.
[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. London, England, UK.
[15] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proc. VLDB Endow.* 14, 9 (may 2021), 1544–1556.
[16] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. 2017. Adaptsize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, 483–498.
[17] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-Based Databases. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland, 301–316.
[18] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, CA, 209–223.
[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages.
[20] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*. San Francisco, CA, USA, 1087–1098.
[21] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Virtual, 17–32.
[22] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. *FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory*. Association for Computing Machinery, New York, NY, USA, 1077–1091.
[23] Cisco. 2016. Cisco Global Cloud Index: Forecast and Methodology, 2016–2021. https://www.cisco.com/c/en/us/solutions/executive-perspectives/annual-internet-report/index.html
[24] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*. ACM, Indianapolis, Indiana, USA, 143–154.
[25] CXL Consortium. 2020. Compute Express Link: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/.
[26] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-free concurrent data structures. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. Boston, MA, 373–386.
[27] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.* 3, 1–2 (2010), 1414–1425.
[28] Dormando. 2019. memcached - a distributed memory object caching system. https://memcached.org/.
[29] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*. London, UK, 1–16.
[30] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, 65–78.
[31] Facebook. 2016. Dragon: A distributed graph query engine. https://engineering.fb.com/data-infrastructure/dragon-a-distributed-graph-query-engine/.
[32] Facebook. 2017. LogDevice: a distributed data store for logs. https://engineering.fb.com/core-data/logdevice-a-distributed-data-store-for-logs/.
[33] Facebook. 2022. RocksDB. http://rocksdb.org/.
[34] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. 2011. Cost Effective Storage Using Extent Based Dynamic Tiering. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX Association, 1–14.
[35] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS.. In *CIDR*.
[36] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel and Distrib. Comput.* 67, 12 (2007), 1270–1285.
[37] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.
[38] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. 2022. SSDs Striking Back: The Storage Jungle and Its Implications on Persistent Indexes. In *Proceedings of the 12th Annual Conference on Innovative Data Systems Research, CIDR*. 9–12.
[39] influxdata. 2023. InfluxDB: Real-time visibility into stacks, sensors and systems. https://www.influxdata.com/.
[40] Intel. 2018. Intel® Optane™ SSD 905P Series 960GB, 2.5in PCIe x4, 3D XPoint™. https://www.intel.com/content/www/us/en/products/sku/147529/intel-optane-ssd-905p-series-960gb-2-5in-pcie-x4-3d-xpoint/specifications.html.
[41] Intel. 2019. Intel® Optane™ Persistent Memory 128GB Module. https://www.intel.co.kr/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf.
[42] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *Proceedings of the 30st International Conference on Distributed Computing (DISC)*. Paris, France, 313–327.
[43] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*. San Francisco, CA, 439–450.
[44] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383* (2019).
[45] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*. Boston, MA, 191–205.

[46] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: A High-performance, Distributed Main Memory Transaction Processing System. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*. Auckland, Newzeland, 1496–1499.

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

[48] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. 2019. MV-RLU: Scaling Read-Log-Update with Multi-Versioning. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Providence, RI, USA). ACM, Providence, RI, 779–792.

[49] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, 385–398.

[50] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*. Virtual, 424–439.

[51] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland.

[52] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*. Virtual, 773–787.

[53] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, 460–477.

[54] Cockroach Labs. 2023. CockroachDB: Scale fast, survive anything, thrive everywhere. https://www.cockroachlabs.com/.

[55] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40.

[56] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Ontario, Canada, 462–477.

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

[58] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*. Los Angeles, CA, 574–587.

[59] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *Proc. VLDB Endow.* 6, 10 (aug 2013), 877–888.

[60] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-Tree for New Hardware Platforms. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*. Brisbane, Australia, 302–313.

[61] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. 2014. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, 501–512.

[62] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal, 1–13.

[63] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 143–157.

[64] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 1–16.

[65] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. 2014. Rethinking main memory OLTP recovery. In *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE)*. Chicago, IL, 604–615.

[66] Ajit Mathew and Changwoo Min. 2020. HydraList: A Scalable In-Memory Index Using Asynchronous Updates and Partial Replication. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*. Tokyo, Japan, 1332–1345.

[67] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. ACM, 243–262.

[68] Paul E. McKenney, Jonathan Appavoo, Andy Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. 2002. Read-Copy Update. In *Ottawa Linux Symposium (OLS)*. 168–183.

[69] Chris Mellor. 2022. SMART brings Optane memory to AMD and Arm. https://blocksandfiles.com/2022/04/13/smart-brings-optane-memory-to-amd-and-arm/.

[70] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems* 9, 1 (feb 1991), 21–65.

[71] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. 2016. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*. Denver, CO, 71–85.

[72] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*. San Jose, California, USA, 1–16.

[73] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. 2015. SQLite Optimization with Phase Change Memory for Mobile Applications. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*. Hawaii, USA, 1454–1465.

[74] Oracle. 2023. MySQL: The world's most popular open source database. https://www.mysql.com/

[75] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. 2017. SQL Statement Logging for Making SQLite Truly Lite. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*. TU Munich, Germany, 513–525.

[76] PingCAP. 2023. TiKV : A distributed transactional key-value database. https://tikv.org/.

[77] Samsung. 2018. Z-SSD. https://semiconductor.samsung.com/ssd/z-ssd/

[78] Mohit Saxena, Michael M. Swift, and Yiying Zhang. 2012. FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. Association for Computing Machinery, 267–280.

[79] Scylla. 2023. ScyllaDB: The Real-Time Big Data Database. https://www.scylladb.com/

[80] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. 2017. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Xi'an, China.

[81] SQLite. 2017. LSM Key/Value Storage in SQLite3. https://charlesleifer.com/blog/lsm-key-value-storage-in-sqlite3/

[82] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harad a, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*. Houston, TX, USA.

[83] Jing Wang, Youyou Lu, Qing Wang, Minhui Xie, Keji Huang, and Jiwu Shu. 2022. Pacman: An Efficient Compaction Approach for Log-Structured Key-Value Store on Persistent Memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA.

[84] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*. Paris, France, 461–472.

[85] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. 2022. FliT: A Library for Simple and Efficient Persistent Algorithms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '22)*. Association for Computing Machinery, 309–321.

[86] Wikipedia. 2022. Cache (computing). https://en.wikipedia.org/wiki/Cache_(computing).

[87] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2018. Anna: A KVS for Any Scale. In *34th IEEE International Conference on Data Engineering, ICDE 2018*. IEEE Computer Society, 401–412.

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

[89] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In

Proceedings of the 2020 USENIX Annual Technical Conference (ATC). Boston, MA, 17–31.

[90] Adar Zeitak and Adam Morrison. 2021. Cuckoo Trie: Exploiting Memory-Level Parallelism for Efficient DRAM Indexing. In *Proceedings of the 28th Symposium on Operating Systems Principles SOSP, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 147–162.

[91] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. 2018. 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)*. 477–492.

[92] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: a tiered file system for non-volatile main memories and disks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*. Boston, MA, 207–219.

[93] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *Proceedings of the 2021 ACM SIGMOD/PODS Conference*. Xian (Virtual), China, 2195–2207.

Received 2022-07-07; accepted 2022-09-22