Eliminating Double I/O Amplifications of Hashing Indexes for Persistent Memory

Anonymous Author(s)
Submission Id: 98

ABSTRACT

Persistent memory (PMem) brings new design considerations in realizing high-performance and scalable hashing indexes. We uncover that existing hashing indexes specially designed for PMem still suffer double I/O amplifications along the access path, stemming mainly from frequent bucket probings and the mismatch in access granularity, which introduce up to 41.5× increase of the access traffic to PMem. We present ElimDA, a scalable hashing index that eliminates the double I/O amplifications to PMem. ElimDA proposes a three-layer index structure that aggregates inserted key-value items in CPU caches and then organizes them in DRAM, which will be flushed in batches later to eliminate the amplification caused by the mismatch in access granularity. ElimDA also maintains fingerprints in DRAM to reduce unnecessary bucket probings to PMem. We implement ElimDA on PMem with persistent and volatile CPU caches. Extensive testbed experiments show that ElimDA improves the insert throughput by 314.8% and the search throughput by 269.4% compared to state-of-the-art hashing indexes for PMem.

1 INTRODUCTION

Hashing indexes that can achieve constant lookup time complexity (i.e., $O(1)$) have been cornerstones in building a wide spectrum of data-intensive applications, spanning across memory databases [5, 12, 46, 47], key-value stores [2, 3, 16, 53], and data deduplication [26, 56, 66, 67]. Recently, the emergence of *persistent memory* (PMem), which provides large capacity, data persistence, and DRAM-comparable access performance, has exhibited great potential to complement or even revolutionize the memory architecture. Intel Optane DC persistent memory (Optane PMem) [42, 51, 70, 71, 74] is the first commercialized product of PMem. It has been deployed in data centers [9, 28] (e.g., Google Cloud [28] and Windows Azure Cloud [9]), which allow users to allocate virtual machines with several TB memory to scale up in-memory data analytics [25, 44, 45, 50]. While Intel recently initiated the winding down of Optane Memory Business [20], industry is still continuing the exploration of PMem deployment via employing Compute Express Link (CXL) based on NAND flash [1, 4], devising byte-addressable SSD as PMem (e.g., Samsung 2B-SSD [10]), and developing other PMem products (e.g., Everspin STT-MRAM product [23] and Renesas MRAM family [63]). Hence, the design of hashing indexes on PMem becomes vital to enable accesses to TB-scale memory space with DRAM-like latency.

Designing hashing indexes for PMem usually has to incorporate two considerations. First, since PMem has limited write endurance (e.g., PCM can only endure $10^7 - 10^9$ writes [27]) and high write energy [70, 74], a number of write-optimized hashing indexes [54, 57, 79, 80] are proposed, either to achieve constant insert and update [1] time complexity (e.g., path hashing [79] and level hashing [80]), or to reduce the resizing overhead (e.g., level hashing [80], Dash [54], and CCEH [57]), with the common objective of reducing unnecessary writes to PMem. Second, PMem delivers DRAM-comparable access performance, making traditional lock-based concurrent control mechanisms become expensive and cumbersome; this motivates the design of lock-free hashing indexes with multi-level structure (e.g., clevel hashing [17]), multiple updating queues (e.g., HiKV [72]), bucket atomic snapshot (e.g., PCLHT [51]), and persistent lock-free sets (e.g., SOFT [81]).

In this paper, we examine the performance of existing hashing indexes on PMem using YCSB benchmark [18]. We find that while being supposed to reduce unnecessary writes to PMem, most existing hashing indexes still incur up to 41.5× increase of the access traffic to the underlying PMem media. This traffic amplification problem usually comes from two fundamental yet unresolved limitations: (L1) the frequent *bucket probings* performed in key searches, which have to walk through the bucket and iteratively retrieve the key-value item stored in each slot for comparison; and (L2) the mismatch between the key-value item size and the access granularity of PMem, which triggers additional traffic to complement the internal buffer embedded in PMem (see §2.3 for more details). We call this problem "*double I/O amplifications*", as the traffic is amplified in two independent aspects along the access path: *increased requests to PMem* (L1) and *complemental traffic within PMem* (L2).

We resolve this problem by presenting ElimDA, a scalable hashing index that eliminates the double I/O amplifications for PMem. ElimDA appends the inserted key-value items in CPU caches and opportunistically flushes them in batch to PMem, thereby saturating the internal buffer of PMem and removing the I/O amplification caused by the mismatch in access granularity. ElimDA also maintains a shrunk index structure in DRAM, whose responsibility is to settle the inserted key-value items with reduced bucket probings. Directly appending incoming key-value items in CPU caches without checking key existence will introduce *duplicate inserts* (i.e., inserting multiple key-value items with the identical key). ElimDA promises search correctness via pinpointing the first appearance for the requested key and employs fingerprinting [58] to reduce unnecessary bucket probings in key searches. Finally, ElimDA compacts the *duplicate items* (i.e., the items with the identical key) during the *segment split* (i.e., a resizing operation that allocates more buckets), so as to accelerate future searches and reduce PMem footprints. Our contributions are summarized as follows.

- We carefully examine state-of-the-art hashing indexes on PMem and identify that among them, the bucket probings and the mismatch in access granularity collectively amplify access traffic to PMem (§2.3).

[1] Insert (a new item) and update (an existing item) are two interfaces called by upper-layer applications.

- We design a scalable hashing index named ElimDA, which eliminates the double I/O amplifications to PMem via several key design techniques: (i) an *insert-ahead logging* scheme that aggregates incoming key-value items in CPU caches and lazily flushes them to both DRAM and PMem; (ii) a *reversed search* approach that employs fingerprinting to quickly pinpoint the valid value of the requested key; and (iii) a *segment split* mechanism that periodically removes duplicate items to reduce unnecessary search efforts. We also discuss the recovery strategy to promise crash consistency (§3).
- We demonstrate the generality of ElimDA by implementing it on PMem with persistent and volatile caches (§3.7).
- We conduct extensive experiments and compare ElimDA against a variety of state-of-the-art hashing indexes for PMem, showing that ElimDA improves insert throughput by 314.8% and search throughput by 269.4% (§4).

We also provide artifacts that describe the testbed, the software, the running commands, and the experimental data, via the following anonymized link: **https://anonymous.4open.science/r/artifact-2337**. We will release the source code of the ElimDA prototype for public use and verification in the final version of this paper.

2 BACKGROUND

We start with the introduction on PMem (§2.1) and elaborate on existing representative hashing indexes specially designed for PMem (§2.2). We also summarize the limitations when deploying hashing indexes on PMem (§2.3).

2.1 Persistent Memory

Persistent memory is a promising candidate of next-generation storage devices, which can directly connect CPU processors through *integrated memory controller* (iMC), so as to offer byte-addressable data persistence with close-to-DRAM speed. In recent years, the emergence of a variety of PMem (e.g. PCRAM [62], STT-MRAM [69], RRAM [73], and Intel Optane PMem [39]) has created new opportunities for redesigning key-value store.

The Intel Optane PMem [38] is one of the first off-the-shelf persistent memory devices implemented based on the 3D-XPoint technology [6, 29, 30, 60]. Figure 1 shows the architecture overview of the Optane PMem. The Optane PMem supplies two operation modes, namely *AppDirect Mode* and *Memory Mode* [32]. The AppDirect Mode provides persistent storage and allows applications to directly access the persistent memory through standard storage calls from file systems, while the Memory Mode supplies DRAM-like memory without any persistence guarantee. Although Intel announces the shutdown of its Optane business [20], it is worth noting that our work is not specifically designed for Optane PMem.

Except the limited write endurance and high write latency, the real PMem device further exhibits two interesting features (F1 and F2) that may affect system designs.

F1 (Constant access granularity). While being assumed to be byte-addressable, commercial PMem accesses data in a constant access granularity in reality (e.g., 256 B in Optane PMem and 4KB in Samsung 2B-SSD), which behaves much more like a block device. Once the size of accessed data is smaller than the media access granularity, PMem has to read additional to complement the accessed

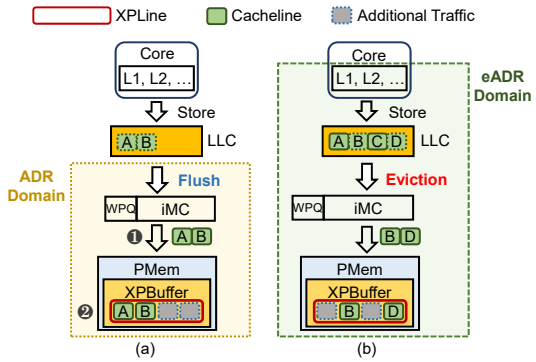


Figure 1: The ADR and eADR persistent domains in the Optane PMem.

data, thereby amplifying traffic to the underlying PMem media. For example, Optane PMem embeds an internal write-combining buffer (called "XPBuffer") to aggregate incoming small writes, and hence has an inherent media access granularity (called "XPLine") with the size of 256 B by default [68]. When flushing two cachelines (with 64 B each) to PMem (Step ❶ in Figure 1), PMem will perform read-modify-write by reading additional data (marked in dashed lines) to complement the XPBuffer (Step ❷), hence doubling the access traffic. It is reported that the 64-B writes (aligned with the 256 B unit sizes) only achieve one-fourth of the write throughput of the 256-B writes for the Optane PMem [74].

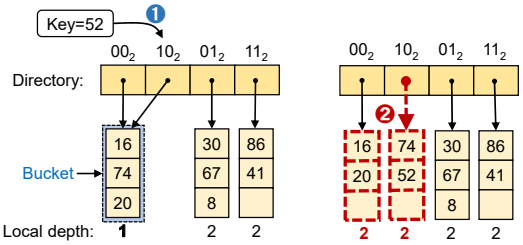
F2 (Persistent CPU caches). With the increasing size of working data sets, traditional SRAM-based *Last Level Cache* (LLC) is hard to scale to accommodate the whole data sets due to its limited memory density. Recently, several emerging non-volatile memory technologies (e.g., PCRAM [43, 52, 78], STT-MRAM [69], and Optane PMem [39]) have been developed to replace SRAM in LLC, so as to provide large cache space with persistence guarantee. For example, Optane PMem provides two different domains for persistent guarantee, namely *Asynchronous DRAM Refresh* (ADR) and *enhanced Asynchronous DRAM Refresh* (eADR) [37, 41]. Specifically, ADR ensures that all pending writes in the *write pending queue* (WPQ) of the iMC will be persisted to PMem even encountering a power loss (Figure 1(a)), while eADR further extends the persistent guarantee to the data resided in CPU caches (Figure 1(b)).

In this paper, we mainly elaborate on the design techniques for the PMem with persistent CPU caches. Nevertheless, we show that our work can be effortlessly extended to other PMems with volatile CPU caches and sustain its effectiveness (see §3.7 and Exp#5 in §4.2).

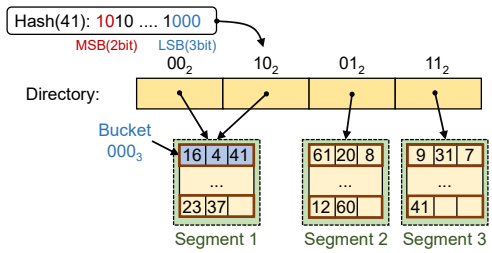
2.2 When Hashing Indexes Meet PMem

We first introduce extensible hashing [57] that is scalable to support efficient resizing operations and summarize existing efforts to combine hashing indexes with PMem.

Extensible hashing: Extensible hashing owns a dynamic structure that can grow and shrink gracefully without having to perform full-table rehashing [24, 57]. Generally, the structure of extensible hashing comprises a number of *buckets* (with multiple slots to store key-value items) and a *directory* (i.e., the bucket addressing table) that maintains entries pointing to the buckets. Extensible hashing has two parameters to define the numbers of entries and buckets,



(a) Insert a key to the bucket of 10_2. (b) Split the first bucket.



(c) Organize buckets into segments in CCEH [57]. It uses MSBs of a hashed key (e.g., 10) to index the segment and employs LSBs (e.g., 000) to locate the corresponding bucket.

Figure 2: An index structure of extensible hashing.

called *global depth* (denoted by g) and *local depth* (denoted by l). The global depth denotes the number of *index bits* selected from a key (usually the *most significant bits* (MSBs)) to index the entry of the associated bucket in the directory, while the local depth of a bucket represents the length of the common index bits used within the same bucket. Figure 2(a) depicts an index structure of extensible hashing with three buckets and a directory with four entries; here, the index bits of the first entry is 00 with the global depth of 2, and the first bucket (with the local depth $l_1 = 1$) is pointed by two entries (i.e., 00_2 and 10_2, which has a common bit '0'). Therefore, the global depth g implies the most entries that a directory can store at present (i.e., 2^g) and the local depth of a bucket is always no larger than the global depth.

When a bucket overflows, extensible hashing compares its local depth with the global depth to decide the extension strategy. Once the local depth is smaller (implying that it is pointed by multiple entries), extensible hashing simply splits this bucket into two buckets, increases their local depths both by one, and disperses the stored items across the two buckets. For instance, when inserting an item (Step ❶) to the first bucket (Figure 2(a)), extensible hashing will split the first bucket and relocate the items (Step ❷, Figure 2(b)). On the other hand, if the local depth of the overflowed bucket is equal to the global depth, extensible hashing will increase the global depth by one to double the entries in the directory (i.e., using one more bit in the index bits), and then split the overflowed bucket into two buckets.

To further reduce the number of entries in a directory, CCEH [57] organizes multiple buckets into a *segment* and uses each entry to locate a segment (Figure 2(c)). It still uses MSBs of a hashed key to index the entry that points to a segment now and employs *least significant bits* (LSBs) to help an item to find the corresponding bucket in a segment. When a bucket overflows, CCEH performs the segment split by generating a new segment and relocating items from the corresponding segment where the overflowed bucket resides to the new segment.

Hashing indexes for PMem: Although conventional hashing indexes (e.g., extendible hashing [24] and linear hashing [49]) can achieve constant lookup complexity, they still fall short of the direct deployment for PMem, as they do not incorporate the characteristics of PMem (e.g., limited write endurance, asymmetric read/write performance, and data consistency) into their designs. To leverage the persistent nature and reduce writes to PMem, a number of hashing indexes have been proposed for PMem in recent years. Here, we elaborate on some representative hashing indexes for PMem.

CCEH [57] is a variant of extendible hashing [24] designed for minimizing the number of cacheline accesses to PMem. To achieve this, it configures cacheline-sized buckets, such that reading a bucket can be accomplished by a cacheline access. Besides, CCEH also groups multiple buckets into a segment so as to ease directory management.

Level hashing [80] is a write-optimized and high-performance hashing index for PMem. It couples a two-layer index table with cuckoo hash [59, 65], such that each inserted item will be provided two optional buckets. Level hashing ensures that a search/update/delete operation needs to probe at most four buckets. It further optimizes the resizing operation by only rehashing the items in 1/3 of the buckets.

Dash [54] is a hashing index that achieves a high *load factor* (i.e., the number of stored items that triggers the resize operation divided by the expected capacity of a hashing index) and good scalability. It uses fingerprints [58] to avoid unnecessary PMem reads and employs an optimistic flavor of bucket-level locking to reduce PMem writes for search operations.

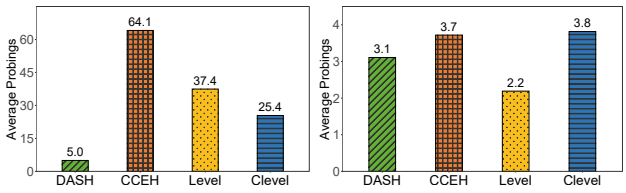
Clevel hashing [17] is a lock-free concurrent hash table based on level hashing. It uses background threads to perform resizing operations, and proposes lock-free concurrent control to avoid lock contentions and improve concurrency.

2.3 Limitations of Existing Efforts

We examine the performance of existing hashing indexes when they are deployed on PMem. We find that they are still prone to trigger a considerable number of additional PMem reads, stemming mainly from the bucket probings (L1). We also identify that each request—once its operated size is small—will amplify the read/write traffic to PMem media again, caused by the mismatch between the request size and the access granularity of PMem media (L2). We call this problem "double I/O amplifications".

Experimental setup: We analyze four representative hashing tables designed for PMem: Dash [54], CCEH [57], level hashing [80], and clevel hashing [17]. We use YCSB benchmark [18] to generate 10 million insert and *positive search* (i.e., when searched keys all exist) operations with the value size of 8 B (a common value size [54, 57, 80]), whose keys follow the uniform distribution. More details about the testbed configurations can be referred to §4.1. We also study the impact of different configurations in §4.3.

Limitation 1 (L1): Frequent key lookups introduce a lot of additional PMem reads. To increase the load factor and reduce resizing operations, existing hashing indexes popularly use linear probings to skim through a bucket, and find the requested item



(a) Number of bucket probings for inserts.

(b) Number of bucket probings for positive searches.

Figure 3: L1 (Frequent key lookups introduce a lot of additional PMem reads). We perform 10 million insert and positive search operations.

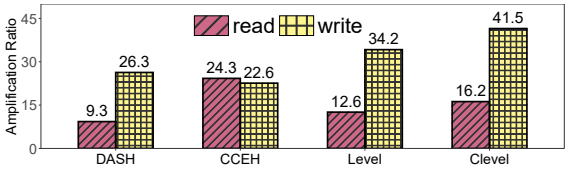


Figure 4: L2 (The mismatch between the key-value item size and the access granularity of PMem aggravates traffic amplification).

(in searches) or locate an unoccupied slot for an inserted item (in inserts). For example, level hashing [80] needs to probe at most four buckets to read/insert a key-value item, and CCEH [57] also has to try at most four cachelines in each insert. Figure 3(a) and (b) further show the number of bucket probings introduced in 10 million insert and positive search operations, respectively.

We find that the four hashing indexes have to probe 5.0-64.1 slots and 2.2-3.8 slots for each insert and positive search operation, respectively. As the keys are stored in PMem, the introduced bucket probings inevitably incur excessive PMem reads. We also find that Dash causes the fewest bucket probings (i.e., 5.0) on average to serve an insert (Figure 3(a)), since it employs fingerprinting [58] and SIMD [19] to reduce unnecessary PMem reads. Besides, the inserts will introduce more bucket probings than the positive search operations. For example, CCEH needs to probe 64.1 buckets on average for an insert (Figure 3(a)), which is 16.2 times higher than that when performing a positive search operation (Figure 3(b)). This is because an insert operation has to traverse all the occupied slots to check the existence of a given key beforehand and then record the given key-value item. Even worse, for the inserts, the additionally generated bucket probings will produce mixed read/write traffic to PMem, which is observed to under-utilize the access bandwidth of PMem [74].

Limitation 2 (L2): The mismatch between the key-value item size and the access granularity of PMem aggravates traffic amplification. Although always assumed to be a byte-addressable memory device, PMem in production systems actually has a fundamental access granularity (e.g., the XPBuffer with the size of 256 B in Optane PMem) to enable persistent storage. Hence, the accesses with the operated size smaller than 256 B will trigger additional traffic for PMem to complement the internal buffer, resulting in low bandwidth utilization (Figure 1(a)). More severely, even for the accesses to sequential keys that are supposed to constitute large sequential data, the random nature of hashing indexes will transform them into random visits again, making it hard to fully saturate the bandwidth of PMem.

To characterize the degree of traffic amplification, we define the *amplification ratio*, calculated as the number of bytes finally written to (resp., read from) PMem media [2] divided by the size of the inserted (resp., requested) key-value items. Figure 4 shows the amplification ratios when inserting and searching 10 million key-value items, respectively. It implies that all the four hashing indexes introduce 9.3-24.3× read amplification and 22.6-41.5× write amplification. Notice that the write amplification is still larger than 16 (i.e., the ratio of the size of XPBuffer (256 B) and the size of key-value item (16B)), because the hashing indexes also trigger additional small writes to PMem (e.g., updates to the global context in clevel hashing [15] and data movements in level hashing under hash collisions [80]).

3 ELIMDA DESIGN

We present ElimDA, a scalable hashing index that eliminates double amplifications with fast recovery for PMem. The main idea behind ElimDA is to (i) reduce bucket probings to PMem by redirecting key searches to DRAM, and (ii) relieve the traffic amplification in PMem by transforming small-random accesses into large-sequential ones and flushing them to PMem.

3.1 Overview

ElimDA employs the following design techniques to address the limitations aforementioned (§2.3). It first proposes an *insert-ahead logging* scheme (§3.3), which directly appends inserted key-value items to a *collecting log* in persistent CPU caches without having to traverse buckets to settle them in PMem. By doing so, ElimDA can reduce a tremendous number of PMem reads (i.e., L1 addressed). ElimDA can also be extended to the PMem with volatile CPU caches (§3.7). Notice that for the update requests, ElimDA first searches the location of the operated item and then performs in-place updates as in previous indexes (e.g., clevel hashing [15] and CCEH [57]).

Directly appending the inserted key-value items without checking for key existence may introduce incorrect searches in future, since there may exist multiple values associated with the identical key caused by the duplicate inserts. To promise search correctness, ElimDA designs a *reversed search* approach (§3.4), which seeks the value currently associated with a given key inserted for the first time, ensuring that the search correctness will not be affected by the duplicated inserts. ElimDA also reclaims the storage space occupied by the duplicate inserts during the segment split (§3.5), thereby reducing PMem footprints and saving unnecessary search efforts in future.

To address the mismatch in access granularity, ElimDA always flushes key-value items to PMem in its access granularity, hence relieving the traffic amplification to PMem media (i.e., L2 addressed).

3.2 Three-Layer Index Structure

Figure 5 shows the index structure of ElimDA, which comprises five major components maintained across three layers.

Cache layer: In the cache layer, ElimDA keeps a number of *collecting logs*, each of which has the same size as the access granularity

[2] We use `PMWatch` [36] to monitor the traffic loaded to PMem media.

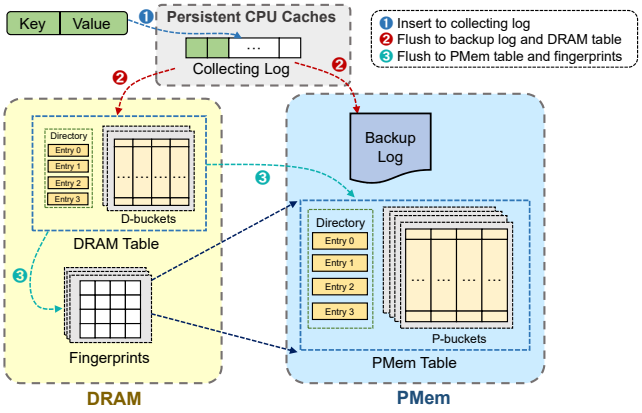


Figure 5: The three-layer index structure of ElimDA.

of PMem [3]. When a new key-value item arrives (Step ❶ in Figure 5), ElimDA appends it to a collecting log based on the MSBs of the key [4]. When a collecting log becomes full, ElimDA then flushes it to both PMem and DRAM (Step ❷), ensuring that these newly inserted key-value items are persisted (in PMem) and can be fast pinpointed (in DRAM). Since a full collecting log has the same size as the access granularity of PMem, the flush operation to PMem can eliminate the traffic amplification caused by the mismatch in access granularity (§2.1). Because the CPU caches are persistent (§2.1), the newly inserted key-value items are persisted once being appended to a collecting log. For each insert, ElimDA further uses a 64-bit atomic *compare-and-swap* (CAS) operation [14] to ensure its atomicity. We also discuss the extension of ElimDA to the PMem with volatile CPU caches in §3.7.

DRAM layer: ElimDA maintains a *DRAM table* and *fingerprints* in DRAM for fast lookups. The DRAM table is an index structure of extensible hashing [57] and is used to temporally index the key-value items that are flushed from CPU caches but not persisted in the index structure of PMem (i.e., the PMem table that we will introduce later). For each key-value item coming from a full collecting log, ElimDA inserts it to the DRAM table based on its MSBs (to locate the segment) and LSBs (to find the associated bucket, see §2.2). Hence, compared to the collecting log that simply stores the unordered key-value items, the DRAM table offers higher search efficiency.

On the other hand, although extensible hashing can pinpoint the bucket for a given key, it still suffers expensive bucket probings: whenever a search/update/insert/delete operation arrives, extensible hashing has to linearly probe each slot of the associated bucket for key lookups, thereby incurring excessive PMem reads. The *negative searches* (i.e., the wanted key does not exist in this bucket) further aggravate the read amplification, since they must traverse all the slots of a bucket. ElimDA cuts down most of the reads to PMem via maintaining fingerprints of keys in DRAM, each of which is a small portion (e.g., one or two bytes [17, 54]) of the key stored in PMem; it compares a given key with the fingerprints stored in DRAM, so as to speculatively predict the possible slots in PMem for being accessed. The items within a bucket of the DRAM table will further be flushed to the PMem table (Step ❸) once the bucket becomes full; after the flush operation completes, ElimDA deletes the items in the full bucket of the DRAM table to make room for future inserts.

Maintaining the DRAM table and fingerprints can bring forth two advantages. First, ElimDA can perform most of the key searches in DRAM, thereby reducing a tremendous number of PMem reads. Second, by leveraging the low access latency of DRAM (typically 69.7% lower than PMem [68]), ElimDA can greatly accelerate key searches.

PMem layer: In the PMem layer, ElimDA stores a *backup log* and a *PMem table* for data persistence. The backup log stores a replica of the key-value items in the full collecting logs when these logs are flushed from CPU caches to PMem. The backup log ensures the consistency of the key-value items of the DRAM table before they are finally persisted to the PMem table. On the other hand, the PMem table is the structure of extensible hashing that stores most of the inserted key-value items (except those stored in the collecting logs and the DRAM table). This design leverages the large storage capacity offered by PMem.

3.3 Insert-Ahead Logging

To reduce bucket probings in key-value inserts, ElimDA proposes an insert-ahead logging scheme, whose main idea is to maintain a number of collecting logs in CPU caches to aggregate the incoming key-value items. Here, we consider the PMem with persistent CPU caches and hence the inserted key-value item is persisted once being appended to the log. We also extend the insert-ahead-logging to another PMem with volatile CPU caches in §3.7 and evaluate its performance in Exp#5 of §4.2. We set the size of a collecting log to L bytes, which has the same size as the access granularity of PMem. For example, if deploying ElimDA on Optane PMem, we can set L to 256 B (i.e., the access granularity in Optane PMem), such that each collecting log is supposed to store 16 key-value items, where each item comprises an 8-B key and another 8-B value. We also discuss the support of variable-size values in §3.7.

Lock-free logging: ElimDA performs lock-free logging to fully leverage the fast access performance of CPU caches. When a new key-value item arrives, ElimDA selects a collecting log based on the MSBs of the key. In each log, ElimDA maintains the number of slots being occupied (denoted by N_{occupy}) and the associated tag bits to record its status. Once trying to append a key-value item to a collecting log, ElimDA first checks if the collecting log has reached its capacity limit. If it still has available slots, then ElimDA increases N_{occupy} by one using a 64-bit atomic *compare-and-swap* (CAS) operation [14]. ElimDA then appends the key and the associated value to the log. Finally, ElimDA flips the corresponding bit in the tag to indicate that the insert has been completed using a CAS operation.

Flushing items to DRAM and PMem: When a collecting log is full, ElimDA flushes the key-value items stored in it to DRAM and PMem in different ways. ElimDA first flushes the full collecting log to a backup log in PMem for data persistence. Since the collecting log has the same size as the access granularity of PMem, this

[3]For example, in Optane PMem, we can set the size of a collecting log to 256 B (i.e., the access granularity of Optane PMem) and allocate 8,192 collecting logs in CPU caches, hence occupying 2 MB of the capacity in total. We evaluate the performance of ElimDA with different sizes of occupied cache space in §4.3.
[4]For example, if we allocate 8,192 collecting logs in CPU caches, then we can use the first 13 bits of a key to determine the associated collecting log.

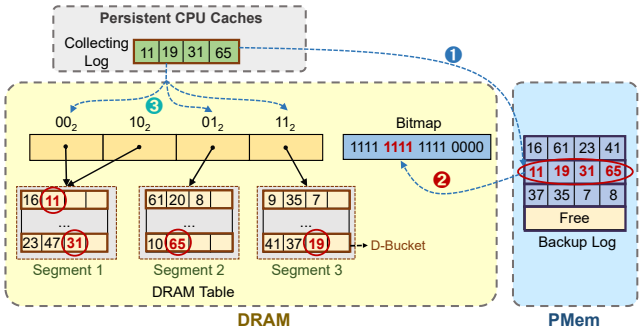


Figure 6: An example of flushing a collecting log to DRAM and PMem. Suppose that a bucket of the DRAM table can store four key-value items.

flush operation will not incur additional traffic to PMem. Figure 6 shows an example to clarify the flush process. ElimDA will locate a physically contiguous region in the backup log to accommodate the key-value items of the full collecting log (Step ❶ in Figure 6). It keeps a bitmap in DRAM, in which a bit '0' indicates that the corresponding slot in the backup log is available to store an incoming item. After the flush operation completes, ElimDA then updates the bitmap and sets the corresponding bits to '1's (Step ❷).

After flushing the items of the full collecting log to PMem, ElimDA then inserts each key-value item of the full collecting log to a DRAM table again, which is an index structure of extensible hashing. Keeping a DRAM table enables fast lookups for the key-value items of the backup log. Figure 6 shows that four key-value items of a full collecting log are inserted into different buckets of the DRAM table (Step ❸).

Support of concurrent control: ElimDA further supports inserting to a full collecting log even though when it is being flushed to DRAM and PMem. Specifically, ElimDA allocates a dedicated *shadow log* (with the same size as the collecting log) for each thread. Once a collecting log becomes full, ElimDA then swaps the associated shadow log with the full collecting log by interchanging their address pointers in an atomic operation without any data movement; at this time, the shadow log will serve as a collecting log to accommodate the incoming inserts. ElimDA then flushes the items of the full collecting log to DRAM and PMem, and changes this collecting log to the shadow log again for future use. This design ensures that the incoming inserts can be continuously appended to the shadow log without being blocked by the flush operation, thereby improving write concurrency.

3.4 Reversed Search

As ElimDA directly appends key-value items without checking key existence, there may be multiple values associated with the same key caused by the duplicate inserts. To ensure search correctness, ElimDA proposes a *reverse search* approach, whose main idea is to search the requested key across the PMem table at first, followed by the DRAM table and the collecting logs. The main objective of the reverse search is to find the associated value with the given key inserted for the first time, so as to promise search correctness.

Fingerprinting for searches over the PMem table: To accelerate key searches across the PMem table, ElimDA employs fingerprinting [58] to reduce unnecessary search efforts. Specifically, ElimDA extracts a portion of each key (e.g., 16 bits) to serve as the fingerprint and stores all the fingerprints in DRAM for fast lookups. The number of bits used in fingerprinting can be tunable to tailor the DRAM footprints and the search accuracy. Hence, to search a key-value item across the PMem table, ElimDA compares it with the fingerprints stored in DRAM in advance: it pinpoints all the fingerprints of the keys within the same bucket as the requested one (based on the LSBs), compares them with the key's fingerprint, and retrieves those items that have the same fingerprint from the PMem table. In this way, ElimDA can perform most key probings in DRAM and greatly narrow the search range before accessing PMem.

Searches across DRAM table and collecting logs: If we cannot find the requested key in the PMem table, we then turn to search it across the DRAM table and the collecting logs. To find a key in the DRAM table, we simply locate the possible segment and bucket based on the MSBs and LSBs of the hashed key, respectively. ElimDA then sequentially compares each key stored in the bucket with the requested one. Since DRAM is byte-addressable, this operation does not introduce traffic amplification. Finding a given key in collecting logs is similar, except that we use the MSBs of the hashed key to pinpoint the collecting log that possibly stores the requested key-value item (§3.3). We finally leaf through the items in the collecting log and see if the key exists. Note that in some rare cases that the key-value items may have been flushed to the backup log but have not been inserted to the DRAM Table, at this time ElimDA will finally scan the requested key in the corresponding shadow log to ensure search correctness. We also evaluate the operation latency in §4.2 (Exp#4).

3.5 Bucket Relocation and Segment Split

Recall that the DRAM table is the index structure of extensible hashing and accepts key-value items coming from the collecting logs (§3.3). To suppress the traffic amplification, ElimDA unifies the structures of the DRAM table and the PMem table, implying that they have the same configurations of the index structures (e.g., the global depth and the local depth for each bucket), except the number of items comprised in a bucket. Specifically, we set a bucket of the DRAM table (called "D-bucket") to comprise N_d items and configure a bucket of the PMem table (called "P-bucket") to store N_p items, satisfying the following two conditions: (i) N_p should be a multiple of N_d (i.e., we can assemble multiple D-buckets to fill a P-bucket); and (ii) the size of a D-bucket should be a multiple of the access granularity of PMem. By doing this, ElimDA can directly flush a full D-bucket from the DRAM table to the corresponding P-bucket of the PMem table without performing extra inserts and this flush operation eliminates traffic amplification to PMem. Figure 7 illustrates the index structures of the DRAM table and the PMem table, both of which have four directory entries (i.e., the global depth is 2) and the same mapping between entries to segments (e.g., both of the entries 00_2 and 10_2 point to Segment 1 in these two tables).

Bucket relocation: When a D-bucket is full, ElimDA will flush all the items belonging to this full D-bucket to the free space of

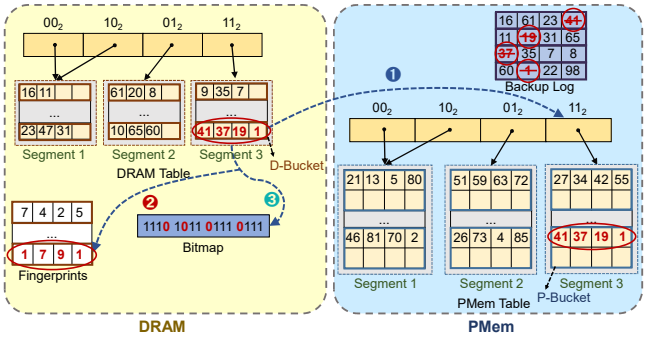


Figure 7: An example of flushing a full D-bucket to the corresponding P-bucket. Suppose that a D-bucket comprises four items ($N_d = 4$) and a P-bucket has eight items ($N_p = 8$).

the corresponding P-bucket (since these two tables have the same structure), so as to make room for accommodating the future key-value items flushed from the full collecting logs to the DRAM table. As a full D-bucket comprises N_d items (a multiple of the access unit of PMem), flushing it to the P-bucket will not introduce traffic amplification. For example, the last D-bucket of the DRAM table in Figure 7 will be flushed to Segment 3 of the PMem table (Step ❶), which is still pointed by the same directory entry (i.e., 11_2). After the relocation process finishes, ElimDA adds the fingerprints of the corresponding keys retrieved from the DRAM table (Step ❷) to facilitate future searches over the PMem table (§3.4). Finally, ElimDA simply flips the corresponding bits of the bitmap to invalidate the replicas of the corresponding items in the backup log (Step ❸), implying that these key-value items have been removed from DRAM and successfully stored in the PMem table. Note that the bucket relocation may generate fragmentations in the backup log; however, the space occupied by the invalidated items can be reused for accommodating the ones flushed from CPU caches once the number of sequential invalidated items in the backup log is equal to that of the items that a collecting log comprises.

Segment split: When a P-bucket in the PMem table overflows, ElimDA will perform the segment split operation, which allocates P-buckets to the newly generated segments in the PMem table, such that more key-value items can be accommodated. The segment split comprises the following steps to eliminate the duplicated items with reduced key comparisons. ElimDA first reads the key-value items of the overflowed P-bucket into DRAM from the top (older) to down (newer). It then compares each retrieved key-value item with the ones that have been relocated to the newly allocated P-buckets and abandons it if the key has already existed in the new P-buckets. Since the new P-buckets always store fewer items that are also distinct throughout the segment split, this operation can remove duplicated items with much fewer key comparisons. To keep in line with the new structure of the PMem table, the DRAM table will also perform segment split, ensuring that their structures are still consistent after split.

3.6 Recovery

ElimDA can guarantee data consistency in the face of unexpected system failures. Once system failures occur, the key-value items in CPU caches and PMem are still available, while the components in DRAM (i.e., the DRAM table, fingerprints, and bitmap) will be lost. ElimDA rebuilds these unavailable components in DRAM in the following steps. ElimDA first compares the keys in the backup log with the ones in the PMem table. There are two possibilities. On one hand, once a key-value item both appears in the backup log and the PMem table, ElimDA then infers that this item has been relocated from the DRAM table to the PMem table and hence the corresponding bit in the bitmap should be '0'. On the other hand, once a key-value item appears in the backup log but is not stored in the PMem table, then ElimDA learns that this item still has a replica in the DRAM table without being flushed to PMem and hence sets the corresponding bit in the bitmap to '1' (§3.1). ElimDA then recovers the fingerprints by reading the corresponding bits of each key-value item in the PMem table and rebuilds the DRAM table based on the bitmap and the backup log.

3.7 Extensions

Extension to other PMems: We show that ElimDA can also be extended to another PMem with volatile CPU caches (e.g., ADR-enabled Optane PMem) with minor modifications. For the PMem with volatile CPU caches , ElimDA directly writes the newly incoming key-value items to the backup log at first and then inserts them to the DRAM table. By doing this, ElimDA can maintain data persistence and good search performance. Although directly appending the new key-value items to the backup log will introduce write amplification to PMem, ElimDA can also relieve this amplification, as the appended key-value items exhibit sufficient locality (i.e., they will be appended to the logs that are physically sequential in PMem). This manner can favor XPBuffer — the embedded write-combining buffer in Optane PMem — to collect and combine items into a 256 B internal write without inducing additional complementary traffic [74]. Actually, ElimDA can also be extended to other PMem products once they exhibit the same mismatch problem in access granularity.

Support for variable-size of key-value items: Till now, ElimDA mainly discusses the support of fixed-length of values. We can also extend ElimDA for coping with the key-value items with variable sizes. One commonly used approach is to replace the variable-size key-value items with the fixed-length pointers [7, 54, 57, 58, 80]. To suppress the traffic amplification, ElimDA separates the storage of the value and their pointers: it appends the incoming values to another dedicated pool (called "value pool"), and stores the associated keys and the value pointers in the collecting log. Hence, for a requested key, ElimDA can fetch the value pointer through scanning index tables and collecting logs, followed by retrieving the associated value from the value pool. We also evaluate the performance of ElimDA under variable sizes of key-value items in Exp#8 (see §4.2).

Integrated with Intel CAT: We can further enhance the retention of items in CPU caches by using Intel Cache Allocation Technology (Intel CAT) [33], which provides software-programmable control over CPU cache space. In particular, ElimDA can employ Intel CAT to specify a dedicated CPU cache space, which cannot be used by other applications except ElimDA (also called "Cache Pseudo-Locking" [34]). By doing this, ElimDA can increase the probability

to keep items in CPU caches before the corresponding collecting log is full.

4 EVALUATION

We conduct extensive testbed experiments to evaluate ElimDA and summarize our major findings as follows.

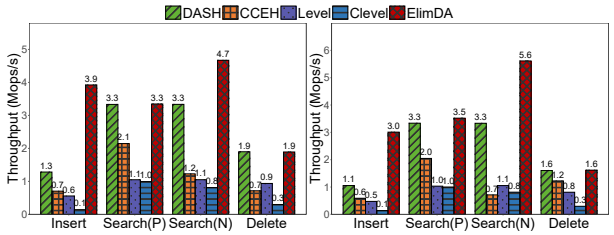
- ElimDA improves the throughput by 68.8-635.7% on average for general operations (Exp#1 and Exp#2).
- ElimDA achieves the best scalability under individual and mixed operations (Exp#2 and Exp#3), and reduces 9.2-90.8% latency for general operations (Exp#4).
- ElimDA still outperforms state-of-the-art hashing indexes when it is deployed on the PMem with volatile CPU caches (Exp#5).
- ElimDA significantly reduces 53.1-93.7% of the storage traffic to PMem (Exp#6) with reasonable DRAM footprints (Exp#7).
- ElimDA retains its effectiveness under variable-size key-value items (Exp#8) and gains a large load factor (Exp#9).
- ElimDA can achieve fast recovery speed (Exp#10) and is also effective when given limited cache space (e.g., 0.5 MB, Exp#11).
- ElimDA still outperforms other hashing indexes (by increasing up to 34.7× of throughput) under different number of PMem DIMMs (Exp#13).

4.1 Setup

Testbed: We conduct all the experiments on a Linux server equipped with two 2.10 GHz Intel Gold 5318Y CPUs (with 24 cores each), 256 GB of DRAM memory, and 1,024 GB of Optane PMem (4×128 GB Optane PMem per socket) configured in the interleaved AppDirect Mode. To avoid NUMA effects [48], we pin threads to one NUMA node and only allow them to access the local DRAM and PMem. Each processor has a shared 36 MB persistent *last-level cache* (LLC). The machine runs Ubuntu 20.04 with the kernel version of 5.4.0.

Configuration parameters: We implement ElimDA using *Persistent Memory Development Kit* (PMDK) with C++ bindings [40], which provides primitives for crash-safe PMem management and synchronization. We set the sizes of keys and values to both 8 B; we further consider the variable value sizes in Exp#8. Each D-bucket is supposed to store 16 key-value items (i.e., with the size of 256 B in total) and every P-bucket can store 256 items (i.e., with the size of 4,096 B in total). The cacheline size is 64 B in our testbed. We set the size of a collecting log to 256 B, which is the same as the access granularity of the Optane PMem. We allocate 8,192 collecting logs (with the size of 2 MB in total) in persistent CPU caches when evaluating ElimDA for eADR-enabled Optane PMem. We also investigate the sensitivity of ElimDA by merely varying a configuration parameter while keeping others unchanged (§4.3). We emphasize that when deploying ElimDA on other PMems (with different media access granularities), we can still eliminate the traffic amplification caused by granularity mismatch via adjusting the size of a collecting log accordingly.

Comparison hashing indexes: We compare ElimDA against another four representative scalable hashing indexes for PMem, namely level hashing [80], clevel hashing [17], CCEH [57] [5], and Dash [54], which optimize hashing indexes from different angles, in terms of write optimization (i.e., level hashing and CCEH), concurrent control (i.e., clevel hashing), and fast recovery (i.e., Dash). For fair comparison, all the hashing indexes are set to contain 60,000 key-value items at the very beginning; once a bucket of a hashing index overflows, the hashing index will be expanded by performing resizing operations to accommodate more inserted items.



(a) Uniform workloads (b) Skewed workloads

Figure 8: Exp#1 (Performance of general operations).

Workloads: We generate various workloads for evaluation using YCSB benchmark [18], which comprises four basic operations: inserts, positive searches (i.e., search existing items), deletes, and negative searches (i.e., search non-existing items). We consider both the uniform and the skewed (i.e., Zipfian(0.99)) key distributions.

4.2 Experiments on ElimDA Property

Exp#1 (Performance of general operations): We start our evaluation by measuring the performance of the four general operations. For each operation, we launch one thread to continuously issue 50 million requests (also used in HiKV [72] and Viper [11]) to the items organized by the five hashing indexes. We then measure the throughput of each operation when the requested keys follow the uniform and the skewed distributions, respectively. We show the results in Figure 8 and make three findings.

First, ElimDA outperforms most of the four hashing indexes for different operations under all workloads. Specifically, ElimDA increases 489.4%, 80.9%, 247.9%, and 68.8% of the throughput on average in insert, positive search, negative search, and delete operations, respectively. ElimDA gains the most improvement on inserts because of the following two reasons. On one hand, ElimDA directly appends the new incoming key-value items in collecting logs and reduces the probings when settling them in the smaller DRAM table (§3.3). On the other hand, ElimDA always aggregates small writes into large sequential writes to PMem (e.g., flushing the full collecting logs (§3.3) and the full D-buckets (§3.5) to PMem), resolving the mismatch problem in access granularity and eliminating the write amplification to PMem.

Second, level hashing and clevel hashing introduce poor performance in inserts, since both of them have to rehash 1/3 of the buckets in a resizing operation. As a comparison, CCEH, Dash, and ElimDA are designed upon extensible hashing, which simply relocates the items in an overflowed bucket by performing resizing operations. Clevel hashing further needs to allocate new space outside the table to store each new item [17] and results in the lowest insert throughput.

[5] We choose the CCEH-PMDK version (https://github.com/DICL/CCEH/tree/master/CCEH-PMDK) in the evaluation and add an unique check to avoid duplicate inserts.

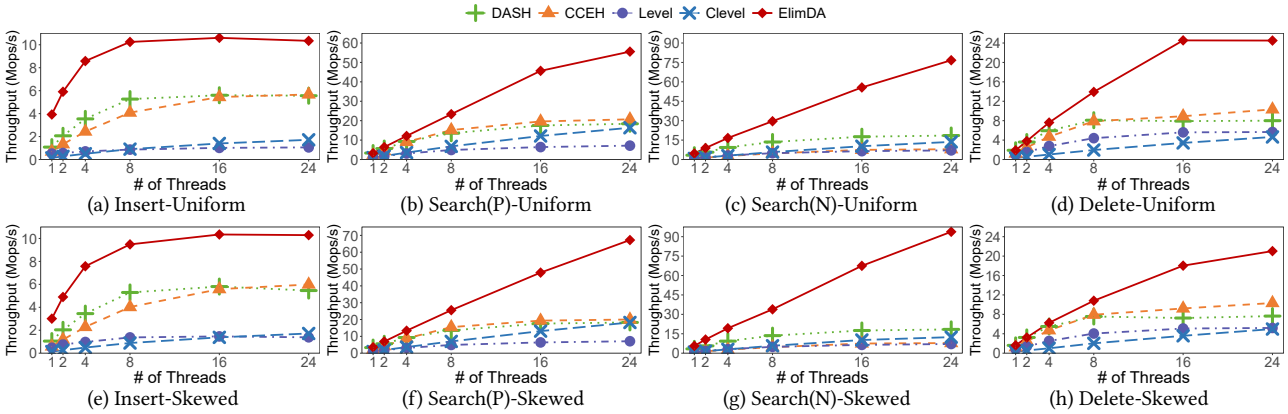


Figure 9: Exp#2 (Scalability)

Third, ElimDA gains near the same performance on the positive search and delete operations compared to Dash. The underlying reason is that both ElimDA and Dash employ fingerprinting to save unnecessary probings in searches, with the difference that ElimDA stores the fingerprints in DRAM to absorb small reads. However, in the experimental settings, ElimDA stores more key-value items in a P-bucket (i.e., 256 items) than Dash, which instead merely keeps 14 items in a bucket, thereby incurring more traffic in reading fingerprints. We further demonstrate that ElimDA gains higher performance on these two operations under multiple threads (Exp#2).

Exp#2 (Scalability): We also evaluate the scalability of the five hashing indexes via increasing the number of threads from 1 to 24, which collectively issue 50 million of the four operations under different key distributions. Here, we measure the accumulated throughput of threads.

Figure 9 shows the throughput when the keys follow the uniform distribution (i.e., Figure 9(a)-(d)) and the skewed distribution (i.e., Figure 9(e)-(h)), respectively. First, all the hashing indexes gain higher throughput with the increase of the number of threads, where ElimDA always achieves the highest throughput by improving up to 635.7% of the throughput. The reason is that ElimDA eliminates the double I/O amplifications by filling the internal buffer of PMem and reducing bucket probings, hence fully exploiting the media bandwidth of PMem.

Second, the throughput of ElimDA shows a linear increase with the number of threads for the positive search and negative search operations (Figure 9(b)(c)(f)(g)). The reasons are two-fold. On one hand, ElimDA provides lock-free searches while CCEH and level hashing add locks in search operations; hence, the search operations of each thread in ElimDA can be performed simultaneously without being blocked by each other. On the other hand, by placing fingerprints in DRAM, ElimDA can access the fingerprints without incurring any traffic amplification to PMem. While Dash also enables lock-free searches, it places fingerprints in PMem, which in turn amplifies the access traffic to PMem and limits its scalability. Besides, the insert throughput of ElimDA (Figure 9(a)(e)) does not linearly increase with the number of threads, since the inserts will be blocked by the segment split and we will explore lock-free split in future work.

Third, the hashing indexes can gain higher throughput for the positive search and negative search operations under the skewed key distributions (Figure 9(f)(g)) than under the uniform key distribution (Figure 9(b)(c)), since the requests under the skewed distribution are likely to be directly served by the cached items without touching PMem.

Exp#3 (Performance under mixed workloads): We further evaluate the performance under mixed workloads generated by the YCSB benchmark [18]. We select four typical workloads with different read-write ratios: (i) `YCSB-A`, which comprises 50% of read (search) operations and another 50% of write (update) operations; (ii) `YCSB-B`, which has 95% of read (search) operations and 5% of write (update) operations; (iii) `YCSB-D`, which issues 95% of read operations to the latest keys and another 5% of insert operations; and (iv) `YCSB-F`, which consists of 50% of write (insert) operations and 50% of read (search) operations. We do not perform the other two workloads generated by the YCSB benchmark (i.e., `YCSB-C` and `YCSB-E`), since `YCSB-C` purely performs read operations (i.e., it is identical to perform positive searches) and `YCSB-E` performs scan operations that are not supported by hashing indexes. We do not evaluate level hashing and Dash under `YCSB-A` and `YCSB-B` workloads, since they do not support the update operations. We increase the threads from 1 to 24 and launch 50 million requests in each workload. Figure 10 shows the throughput under the uniform key distribution.

ElimDA achieves the highest throughput under all four mixed workloads. Specifically, ElimDA improves the throughput by 175.2%, 229.3%, 199.8%, and 199.3% on average under `YCSB-A`, `YCSB-B`, `YCSB-D`, and `YCSB-F` workloads, respectively. Besides, the throughput of ElimDA almost linearly increases with the number of threads under the first three workloads (Figure 10(a)(b)(c)), as their operations (reads and updates) will not introduce segment split. This experiment implies that ElimDA maintains good scalability under mixed workloads.

Exp#4 (Latency): To study the latency introduced by the three-layer design in ElimDA, we continuously insert 50 million requests (following the uniform distribution) for each operation (i.e., insert, positive search, negative search, and delete) and record the latency of each request, from the time when it is issued to the time when

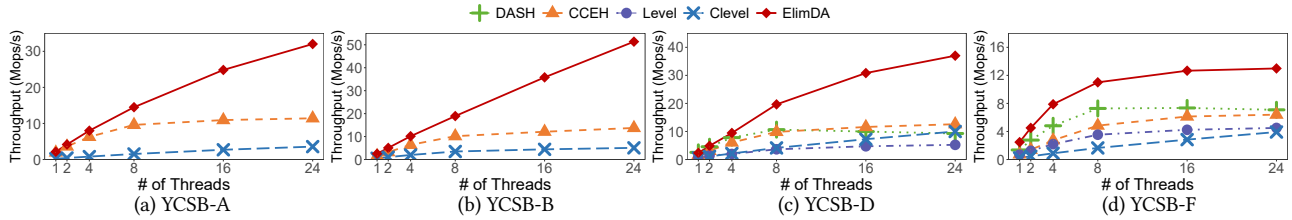
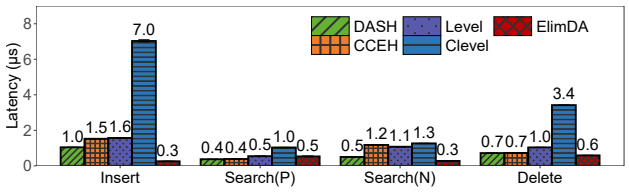
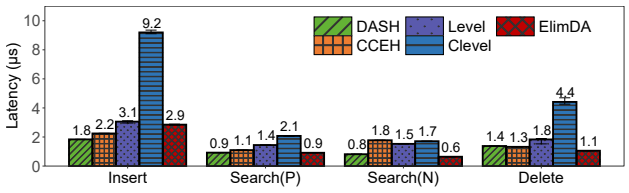


Figure 10: Exp#3 (Performance under mixed workloads).



(a) Average latency



(b) Tail latency

Figure 11: Exp#4 (Latency).

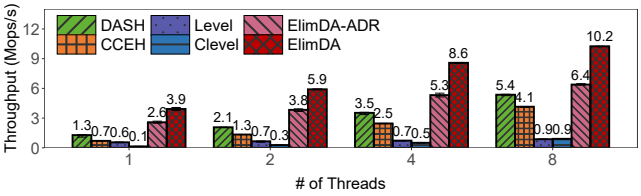


Figure 12: Exp#5 (Generality).

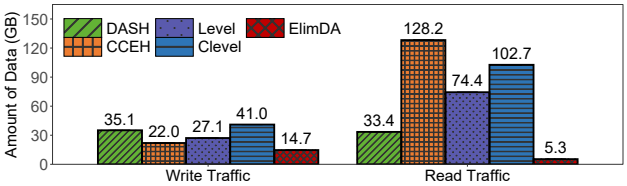


Figure 13: Exp#6 (Storage traffic).

the response is received. We mainly focus on the average latency and the P99 latency.

Figure 11(a) shows that ElimDA gains the lowest average latency in most cases. Specifically, compared to the other four hashing indexes, it reduces the average latency by 90.8%, 9.2%, 72.4%, and 60.1% for insert, positive search, negative search, and delete operations, respectively. In particular, for an incoming insert, ElimDA can directly append it to the collecting log without trying the optional slots as in level hashing and clevel hashing, hence reducing the insert latency. Second, ElimDA slightly increases the average latency of positive searches compared to DASH and CCEH, since ElimDA has to traverse the PMem table, DRAM table, and the collecting log for locating the requested item; however, ElimDA maintains fingerprints in DRAM and employs SIMD to accelerate searches.

Figure 11(b) shows the P99 latency of the five hashing indexes. Compared to DASH and CCEH, ElimDA slightly increases the tail latency in inserts, but reduces tail latency in the other three operations. The underlying reason is that ElimDA needs to flush the collecting log—once it becomes full—to both PMem and DRAM; at this time, the last insert of this full collecting log will complete until all the items of the collecting log is successfully flushed, hence prolonging the tail latency of inserts. However, ElimDA performs most of key searches in DRAM using fingerprinting, hence accelerating the key searches in the other three operations and shortening their tail latencies.

Exp#5 (Generality): We further study the generality of ElimDA. In previous experiments, we run ElimDA atop the PMem with persistent CPU caches and measure its performance. Here, we deploy ElimDA atop ADR-enabled Optane PMem, which is another representative of PMem with volatile caches (denoted as ElimDA-ADR). ElimDA and ElimDA-ADR merely differ in the inserts, where ElimDA-ADR has to diligently persist each incoming key-value item to PMem, while ElimDA simply appends it to the collecting log in CPU caches. Hence, we continuously insert 50 million items using different number of threads and measure the resulting throughput for both ElimDA and ElimDA-ADR.

Figure 12 shows that the throughput of ElimDA-ADR generally grows with the number of threads (from 2.6 Mops/s to 6.4 Mops/s) and achieves around 63.7% of the throughput of ElimDA. The reason is that ElimDA-ADR has to frequently use the flush instructions (e.g., `cflush` and `clwb`) and memory barrier, ensuring that the incoming key-value items can be timely persisted in the right order. Besides, ElimDA-ADR still outperforms the other four hashing indexes under the four operations by improving the 314.4% on average of the insert throughput. We also validate that ElimDA-ADR achieves the comparable performance as ElimDA under other operations and omit it due to the page limit.

Exp#6 (Storage traffic): To assess the efficacy of ElimDA on reducing the storage traffic to the underlying PMem media, we measure the storage traffic finally written to (resp., read from) PMem media when inserting 50 million key-value items.

Figure 13 shows that ElimDA can reduce 53.1% of the write traffic to PMem on average compared to other four hashing indexes. The underlying reason is that ElimDA aggregates the incoming key-value items using insert-ahead logging (§3.3) and flushes the data (with the same size of the access granularity of PMem), hence avoiding the unnecessary read-modify-write operations performed to PMem media. Moreover, ElimDA also cuts down 93.7% of the read traffic to PMem on average, it employs fingerprinting and item

Table 1: Exp#7 (Space overhead).

	Number of inserted items (million)				
	20	40	60	80	100
DRAM Space (GB)	0.35	0.63	0.99	1.18	1.59
PMem Space (GB)	3.08	6.17	11.82	12.33	12.33
Ratio	11.36%	10.21%	8.37%	9.57%	12.89%

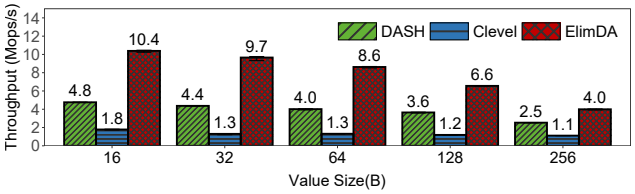


Figure 14: Exp#8 (Performance for variable-size key-value items).

flushing to mitigate bucket probings and granularity mismatch, respectively.

Exp#7 (Space overhead): We also measure the DRAM and PMem space needed by ElimDA via varying the number of inserted items from 20 million to 100 million. Table 1 conveys two findings. First, ElimDA needs larger DRAM and PMem space when more key-value items are inserted, since it needs more buckets (both D-buckets and P-buckets) to index the items. Second, the ratio of DRAM space and PMem space keeps stable (from 8.37% to 12.89%) across different number of inserted items and results in the average of 10.48%. The reason is that ElimDA only maintains a table (called DRAM table) and fingerprints in DRAM for fast lookups (§3.2), whose space is roughly $\frac{1}{16}$ of the PMem space occupied by ElimDA. We believe that the ratio of DRAM and PMem consumption (i.e., around 1:10) in ElimDA is tolerable, which is significantly lower than that of our testbed configuration (1:4). In practice, we can continuously reduce the ratio by storing all the values in PMem if the value size becomes larger.

Exp#8 (Performance for variable-size key-value items): We also evaluate the performance via varying the value size from 16 B to 256 B. For each value size, we insert 50 million key-value items using 24 threads. We do not evaluate level hashing and CCEH in this experiment, since they do not support variable-size of key-value items. Figure 14 delivers two observations.

First, the throughput of the three hashing indexes generally declines with the increasing value size, since PMem needs more time to persist the inserted items with larger value size. Second, compared to Dash and clevel hashing, ElimDA still achieves a 98.3% and 477.2% increase in throughput under different value sizes, respectively. The reason is that these three hashing indexes store an 8-B key and another 8-B value pointer in PMem for each item, where Dash and clevel hashing will introduce double traffic amplifications caused by the bucket probings and the mismatch in access granularity. Since small-value objects occupy a majority in many production workloads [8, 13, 16, 58, 61] (e.g., more than 90% of value sizes are smaller than 34 B in ZippyDB [13], and 70.0% of values are smaller than 300 B in Facebook's Memcached deployment [8]), we believe ElimDA can assist existing production systems to gain higher performance in small-value-dominated scenarios.

Exp#9 (Load factor): We evaluate the changing of the load factor by continuously inserting 240 million key-value items. With the inserts proceeding, each hashing index will experience resizing operations once a P-bucket overflows.

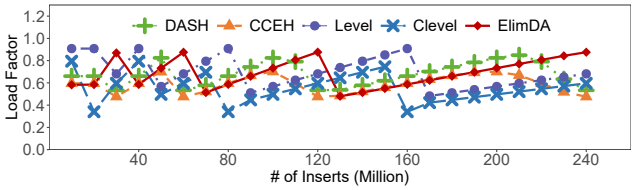


Figure 15: Exp#9 (Load factor).

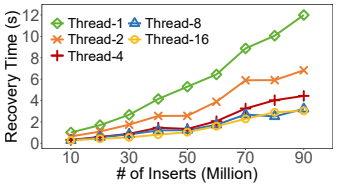


Figure 16: Exp#10 (Recovery performance).

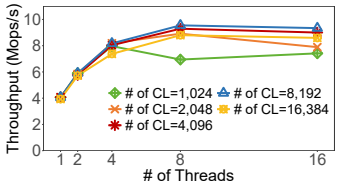


Figure 17: Exp#11 (Impact of number of collecting logs).

Figure 15 conveys two observations. First, the load factor of ElimDA fluctuates from 48.0% to 87.6% and achieves the average of 68.9%, which outperforms another three hashing indexes (i.e., clevel hashing, CCEH, and Dash). Level hashing gains the largest load factor (69.5% on average), since it employs cuckoo hashing [59, 65] to provide two optional buckets for settling a new item, which increases the load factor at the expense of more bucket probings (see Exp#1). Second, clevel hashing has the smallest load factor (55.2% on average), since it removes the one-step movement (i.e., copying an inserted item into its another candidate bucket) originally adopted in level hashing, reducing the storage utilization.

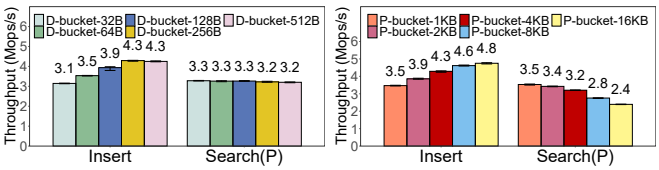
Exp#10 (Recovery performance): Once a power loss occurs, ElimDA has to rebuild the components stored in DRAM, including the DRAM table and the fingerprints (§3.6). Figure 16 depicts the recovery time under different number of threads, where the number of inserts is changed from 10 million to 90 million. We can observe that ElimDA can quickly recover data when encountering power failures (e.g., recover 50 million items with merely 1.2 seconds under 8 threads). We can also find that the reduction of the recovery time becomes marginal even increasing the number of threads from 8 to 16. This is because ElimDA uses locks to ensure the correctness of data recovery. The recovery with multiple threads aggravates lock contentions, which in turn compensates for the reduction of the recovery time.

4.3 Experiments on Sensitivity

We also study the sensitivity of the configured parameters and measure their impact on the performance.

Exp#11 (Impact of the number of collecting logs): We first assess the impact of the number of collecting logs. We vary the size of collecting logs from 0.25 MB (i.e., with 1,024 collecting logs) to 4 MB (i.e., with 16,384 collecting logs). We insert 50 million key-value items and measure the resulting throughput under different number of threads.

Figure 17 shows that the throughput of ElimDA generally increases with the number of collecting logs, from 6.5 Mops/s (when there are 1,024 collecting logs) to 7.4 Mops/s (when there are 16,384



(a) Varying D-bucket size. (b) Varying P-bucket size.

Figure 18: Exp#12 (Impact of bucket size).

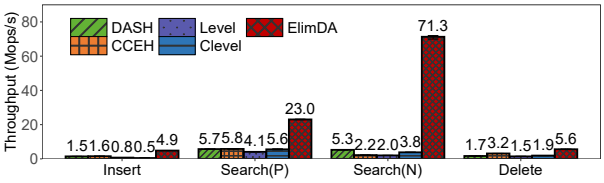


Figure 19: Exp#13 (Impact of number of DIMMs).

collecting logs). However, when the number of threads is fixed, the throughput also exhibits small fluctuations when the number of the collecting logs changes. The reason is that fewer collecting logs make inserts from different threads more likely to arrive at the same bucket. Although ElimDA implements lock-free logging to remove lock overhead, more threads inserting into the same log still reduce ElimDA's scalability.

Exp#12 (Impact of bucket size): We further study the impact of the bucket size. Recall that ElimDA maintains two tables in DRAM and PMem, which comprise D-buckets and P-buckets with different capacities, respectively. In this experiment, we perform two tests to study the impact of bucket size: (i) we first fix the size of P-bucket to 64 KB and measure the performance when the size of D-bucket is varied from 32 B to 512 B; and (ii) we set the size of the D-bucket to 256 B and vary the size of the P-bucket from 1 KB to 16 KB.

Figure 18 shows the throughput of insert and positive search operations under different sizes of D-bucket and P-bucket. We find that the insert performance first increases with the D-bucket size and then stabilizes (Figure 18(a)), as the small-sized D-bucket (smaller than 256 B) will introduce access amplification.

Besides, when the size of P-bucket becomes larger, the throughput of positive searches declines (Figure 18(b)), as ElimDA needs to probe more slots to find the requested item. As a comparison, the insert throughput climbs with the P-bucket size, as a larger P-bucket needs fewer segment splits.

Exp#13 (Impact of number of DIMMs): We finally run ElimDA on a single Optane DIMM to investigate its performance when the accumulated bandwidth of Optane PMem shrinks. We launch 24 threads to simultaneously perform four operations (insert, positive search, negative search, and delete) to saturate the PMem bandwdith. We measure the accumulated throughput of threads.

Figure 19 shows that even though the bandwidth of Optane PMem is limited, ElimDA still outperforms its competitors by improving the throughput by 0.75×-34.7× across different operations. That is because ElimDA reduces the traffic amplification to PMem through insert-ahead logging (§3.3) and employs fingerprinting (§3.4) to quickly pinpoint the requested key-value item without probing the bucket in PMem, improving the bandwidth utilization of PMem.

5 RELATED WORK

We classify the closest related work into two branches: hashing indexes for PMem and tree-based indexes for PMem.

Hashing indexes for PMem: To obtain fast and constant indexing performance, a massive number of hashing indexes are designed for PMem [21, 35, 64, 79, 80]. Path hashing [79] organizes storage buckets into an inverted complete binary tree, where the buckets in the bottom levels are shared by the ones in the top levels. By doing so, path hashing minimizes the writes to NVM and alleviates hash collisions. As a variant of path hashing, level hashing [80] provides a sharing-based two-level hash table for achieving the constant-scale time complexity with the improved load factor. Clevel hashing [17] further designs a multi-level structure that allows the resizing operation to be performed in the background without affecting concurrent queries. However, clevel hashing requires logging or CoW to promise crash consistency. CCEH [80] is built atop extensible hashing [24] to optimize the dynamic rehashing (resizing) performance with failure-atomicity guaranteed. It employs cacheline-sized buckets to minimize the number of cacheline accesses and groups many buckets into a segment to ease directory management. PCLHT [51] is the persistent version of Cache-Line Hash Table (CLHT) [22]. It sets the bucket size to a cacheline size, ensuring that each update to the hash table requires only one cacheline access. Dash [54] adopts fingerprinting to reduce PMem reads and uses verifications to avoid PMem writes, so as to realize dynamic and scalable hashing on PMem. However, Dash still misses the mismatch between PMem access granularities and key-value item size. Halo [31] places the entire hash index in DRAM to avoid persistence overhead caused by rehashing; it also adpots a log-structured storage layout in PMem. However, Halo is hard to scale to large hashes because of its huge DRAM footprints.

As a comparison, ElimDA aims to eliminate the I/O amplifications to PMem, caused by bucket probings and the mismatch in access granularity.

Tree-based indexes for PMem: In addition to the hashing indexes, some tree-based indexes are also proposed for accelerating the range queries. NV-Tree [75] reduces the consistency overhead of B$^+$-tree in NVM via solely enforcing consistency on leaf nodes and keeping their entries unsorted. FPTree [58] is a hybrid PMem-DRAM persistent and concurrent B$^+$-Tree, which persists leaf nodes in PMem and places inner nodes in DRAM, such that the inner nodes can be reconstructed upon unexpected crashes. In view of the different characteristics of PMem and DRAM, HiKV [72] designs a hybrid index structure: it maintains the hash index in PMem for fast indexing and keeps the B$^+$-tree in DRAM to support range queries. However, HiKV has to synchronize the two indexing structures for each update. FlatStore[16] keeps a hash index and a B$^+$-Tree in DRAM; it then adopts a compacted per-core oplog to guarantee persistence. ChameleonDB [77] is a multi-shard index, where each shard is a multi-level LSM-like structure and has an in-DRAM memTable to aggregate KV items. To support range queries with variable-sized keys, ROART [55] is designed based on the radix tree, which postpones the leaf splits and organizes multiple leaf nodes into a leaf array. While most of existing tree-based index structures mainly concern improving the access throughput, μTree [15] reduces the tail latency via putting the entire B$^+$-tree in DRAM and

maintaining an additional list layer in PMem for data persistence and crash recovery. NBTree [76] serializes leaf node operations using atomic primitives and achieves lock-free accesses to eADR-enabled PMem. Different from these studies, ElimDA mainly focuses on designing PMem-friendly hashing indexes.

6 CONCLUSION

Persistent memory (PMem) opens up new opportunities to re-architect memory hierarchy. Hashing indexes are key components to realize fast accesses to PMem. We find that existing hashing indexes are prone to incurring access amplifications, because of bucket probings and mismatch in access granularity. We present ElimDA, a new scalable hashing index that eliminates the double I/O amplifications to PMem. ElimDA allocates collecting logs in CPU caches to aggregate incoming key-value items for fully utilizing the bandwidth of PMem. It then flushes the collected key-value items to DRAM and PMem for both fast lookups and data persistence. ElimDA further employs fingerprinting to reduce unnecessary PMem reads in key lookups. We show that ElimDA dramatically improves the access throughput compared to state-of-the-art hashing indexes for PMem.

REFERENCES

[1] Last week Intel killed Optane. Today, Kioxia and Everspin Announced Comparable Tech: Rumors of Storage-Class Memory's Demise may have been Premature. https://www.theregister.com/2022/08/02/kioxia_everspin_persistent_memory/.

[2] Memcached - A Distributed Memory Object Caching System. https://memcached.org/.

[3] Redis. https://redis.io/.

[4] Samsung's Memory-Semantic CXL SSD Brings a 20X Performance Uplift. https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift.

[5] Mongodb. https://www.mongodb.com/, 2022.

[6] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman, et al. Cascade Lake: Next Generation Intel Xeon Scalable Processor. *IEEE Micro*, 39(2):29–36, 2019.

[7] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. BzTree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of A Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[9] Girish Bablani. Introducing new product innovations for SAP HANA, Expanded AI collaboration with SAP and more. https://azure.microsoft.com/en-us/blog/introducing-new-product-innovations-for-sap-hana-expanded-ai-collaboration-with-sap-and-more/, 2019.

[10] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 425–438, 2018.

[11] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An Efficient Hybrid Pmem-Dram Key-Value Store. *Proceedings of the VLDB Endowment*, 14(9):1544–1556, 2021.

[12] Dhruba Borthakur. The Hadoop Distributed File System: Architecture and Design. *Hadoop Project Website*, 11(2007):21, 2007.

[13] Zhichao Cao and Siying Dong. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*, 2020.

[14] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *Proc. of USENIX FAST*, 2020.

[15] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. uTree: A Persistent B+-Tree with Low Tail Latency. *Proceedings of the VLDB Endowment*, 13(12):2634–2648, 2020.

[16] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1077–1091, 2020.

[17] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. Lock-Free Concurrent Level Hashing for Persistent Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812, 2020.

[18] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[19] Intel Corporation. Intel 64 and IA-32 architectures softwaredeveloper's manual.2015.

[20] Intel Corporation. Intel Reports Second-Quarter 2022 Financial Results. https://www.intc.com/news-events/press-releases/detail/1563/intel-reports-second-quarter-2022-financial-results, 2022.

[21] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386, 2018.

[22] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The Secret to Scaling Concurrent Search Data Structures. *ACM SIGARCH Computer Architecture News*, 43(1):631–644, 2015.

[23] Everspin. Spin-transfer Torque DDR Products. https://www.everspin.com/spin-transfer-torque-ddr-products, 2022.

[24] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H Raymond Strong. Extendible Hashing—A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems (TODS)*, 4(3):315–344, 1979.

[25] John Hubbard Flavio Fomin. Deploying a Cost-Efficient, High-Performance vSAN Cluster Best Practices Guide. https://www.intel.com/content/www/us/en/architecture-and-technology/deploying-vsan-cluster-best-practices-guide.html, 2021.

[26] David Geer. Reducing the Storage Burden via Data Deduplication. *Computer*, 41(12):15–17, 2008.

[27] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Software wear management for persistent memories. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 45–63, 2019.

[28] Google. Available first on Google Cloud: Intel Optane DC Persistent Memory. https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory, 2018.

[29] Philipp Gotze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. Data Structure Primitives on Persistent Memory: An Evaluation. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pages 1–3, 2020.

[30] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform Storage Performance with 3D XPoint Technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[31] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. Halo: A Hybrid PMem-DRAM Persistent Hash Index with Fast Recovery. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1049–1063, 2022.

[32] Alper Ilkbahar. Intel Optane DC Persistent Memory Operating Modes Explained, 2018.

[33] Intel. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html?wapkw=intel%20cat, 2016.

[34] Intel. User Interface for Resource Control feature. https://www.kernel.org/doc/html/v5.9/x86/resctrl_ui.html#cache-pseudo-locking, 2016.

[35] Intel. Pmemkv. http://pmem.io/pmemkv/index.html, 2019.

[36] Intel. Pmwatch. https://github.com/intel/intel-pmwatch, 2020.

[37] Intel. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, 2021.

[38] Intel. Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2021.

[39] Intel. Intel® Optane™ Persistent Memory 200 Series Brief. https://www.intel.la/content/www/xl/es/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html, 2021.

[40] Intel. Persistent Memory Development Kit. https://pmem.io/, 2021.

[41] Intel. Intel® Xeon® Processor Scalable Family Technical Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html, 2022.

[42] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714*, 2019.

[43] Gangyong Jia, Guangjie Han, Jinfang Jiang, and Li Liu. Dynamic Adaptive Replacement Policy in Shared Last-Level Cache of DRAM/PCM Hybrid Memory for Big Data Storage. *IEEE Transactions on Industrial Informatics*, 13(4):1951–1960, 2016.

[44] Flavio Fomin John Hubbard. Boost VMware vSphere Efficiency with Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/vsphere-optane-pmem-best-practices-guide.html, 2019.

[45] Madhuri Kaniganti. Next Generation SAP HANA Large Instances with Intel® Optane™ drive lower TCO. https://azure.microsoft.com/en-us/blog/next-generation-sap-hana-large-instances-with-intel-optane-drive-lower-tco/, 2020.

[46] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. Oracle Database In-Memory: A Dual Format In-Memory Database. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1253–1258. IEEE, 2015.

[47] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[48] Christoph Lameter. NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51, 2013.

[49] Per-Ake Larson. Dynamic Hash Tables. *Communications of the ACM*, 31(4):446–457, 1988.

[50] Quoc-Thai V Le. Enhance VMware* VMs with Intel® Optane™ DC Persistent Memory. https://www.intel.com/content/www/us/en/developer/articles/technical/enhance-vmware-vms-with-intel-optane-persistent-memory.html, 2019.

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

[52] Won Jun Lee, Chang Hyun Kim, and Seon Wook Kim. A Last-Level Cache Management for Enhancing Endurance of Phase Change Memory. In *2021 36th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*, pages 1–4, 2021.

[53] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to Achieve A Billion Requests per Second Throughput on A Single Key-Value Store Server Platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 476–488, 2015.

[54] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. *Proceedings of the VLDB Endowment*, pages 1147–1161, 2020.

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

[56] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying Data Deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*, pages 12–17, 2008.

[57] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, 2019.

[58] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386, 2016.

[59] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[60] Ivy B Peng, Maya B Gokhale, and Eric W Green. System Evaluation of the Intel Optane Byte-Addressable NVM. In *Proceedings of the International Symposium on Memory Systems*, pages 304–315, 2019.

[61] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware Thread Management. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 145–160, 2018.

[62] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-Change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[63] RENESAS. MRAM Family Overview. https://www.renesas.cn/cn/zh/document/ovr/mram-family-overview?r=587766, 2022.

[64] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-hashmap: A Persistent and Concurrent Hashmap for Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, pages 1–8, 2015.

[65] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, and Pengfei Zuo. SmartCuckoo: A Fast and Cost-Efficient Hashing Index Scheme for Cloud Storage Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 553–565, 2017.

[66] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device Mapper Target for Data Deduplication. In *2014 Ottawa Linux Symposium*. Citeseer, 2014.

[67] Yoshihiro Tsuchiya and Takashi Watanabe. Dblk: Deduplication for Primary Block Storage. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, 2011.

[68] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Building blocks for persistent memory. *The VLDB Journal*, 29(6):1223–1241, 2020.

[69] Mengxing Wang, Wenlong Cai, Daoqian Zhu, Zhaohao Wang, Jimmy Kan, Zhengyang Zhao, Kaihua Cao, Zilu Wang, Youguang Zhang, Tianrui Zhang, et al. Field-Free Switching of a Perpendicular Magnetic Tunnel Junction through the Interplay of Spin–Orbit and Spin-Transfer Torques. *Nature electronics*, 1(11):582–588, 2018.

[70] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and Modeling Non-Volatile Memory Systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508. IEEE, 2020.

[71] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–19, 2019.

[72] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, 2017.

[73] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the Challenges of Crossbar Resistive Memory Architectures. In *2015 IEEE 21st international symposium on high performance computer architecture (HPCA)*, pages 476–488. IEEE, 2015.

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

[75] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[76] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. NBTree: A Lock-Free PM-Friendly Persistent B+-Tree for eADR-enabled PM Systems. *Proceedings of the VLDB Endowment*, 15(6):1187–1200, 2022.

[77] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. ChameleonDB: a Key-Value Store for Optane Persistent Memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 194–209, 2021.

[78] Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–21, 2012.

[79] Pengfei Zuo and Yu Hua. A Write-Friendly and Cache-Optimized Hashing Scheme for Non-Volatile Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):985–998, 2017.

[80] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.

[81] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient Lock-Free Durable Sets. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–26, 2019.