Exploiting Persistent CPU Cache for Scalable Persistent Hash Index

Bowen Zhang†, Shengan Zheng*‡, Liangxu Nie†, Zhenlin Qi†, Linpeng Huang*†, Hong Mei†
†*Department of Computer Science and Engineering, Shanghai Jiao Tong University*
‡*MoE Key Lab of Artificial Intelligence, AI Institute, Shanghai Jiao Tong University*
{bowenzhang, shengan, nieliangxu, qizhenlin, lphuang, meih}@sjtu.edu.cn

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00295

***Abstract*—Byte-addressable persistent memory (PM) has been widely studied in the past few years. Recently, the emerging eADR technology further incorporates CPU cache into the persistence domain. The persistent CPU cache is promising to optimize the write performance of PM-based storage systems and facilitate the design of concurrent crash-consistent data structures. In this paper, we propose Spash, a highly scalable persistent hash index for PM systems with persistent CPU cache. Spash fully exploits the benefits of persistent CPU cache to implement a durable linearizable index with low PM access overhead and high concurrency. Spash employs a fine-grained extendible hash architecture and a metadata-free segment design to minimize the number of PM accesses. Moreover, Spash adopts adaptive in-place updates and compacted-flush insertions, which dramatically conserve scarce PM write bandwidth by absorbing a large amount of PM write in the persistent CPU cache. Furthermore, Spash proposes a two-phase concurrency protocol and a collaborative staged doubling mechanism, which leverage the persistent CPU cache and hardware transactional memory to achieve lock-free concurrency and durable linearizability. Spash outperforms the other state-of-the-art persistent hash indexes in YCSB workloads by up to 19.6×.**

***Index Terms*—Persistent Memory, Persistent CPU Cache, Hash Index**

I. Introduction

Byte-addressable persistent memory (PM) promises data durability with DRAM-comparable performance. The release of Intel's first generation Optane DC Persistent Memory Module (DCPMM) [1] in 2019 has spawned a slew of research proposals and prototype systems [2]–[9]. These studies mostly assumed that CPU cache is volatile since the ADR (Asynchronous DRAM Refresh) [10] mechanism of Optane DCPMM only guarantees the persistence of the memory and *write pending queues* (WPQ). Two years later, Intel unveiled the second generation Optane DCPMM [11], along with a new platform feature called eADR (extended ADR) [12]. eADR further pushes the boundaries of the persistence domain to incorporate CPU cache. During a power outage, eADR guarantees the persistence of the CPU cache by flushing the data of the CPU cache to PM with the reserved energy.

Persistent CPU cache offers new opportunities to fully unleash the write and concurrent performance of PM-based systems. First, PM-based systems are anticipated to squeeze the most write performance out of PM by effectively leveraging the persistent CPU cache. PM has a relatively low write bandwidth, which is around 3× lower than its read bandwidth and 5× lower than DRAM write bandwidth. Moreover, when the CPU cache is volatile, PM-based systems must flush each modified cacheline to PM synchronously to ensure data durability, which increases the write latency and exacerbates the scarce PM write bandwidth [13]. With persistent CPU cache, synchronous flush instructions are no longer necessary, thereby reducing PM write latency on the critical path. Moreover, PM write bandwidth is also conserved substantially by eliminating the repetitive flushes to hot PM regions.

Second, persistent CPU cache facilitates the design of concurrent crash-consistent data structures with high scalability. The key challenge for concurrent PM-based data structures is the guarantee of durable linearizability [14], which preserves the linearizability of data structures even in the case of power failure. Previously, the volatile CPU cache created a gap between an update being globally visible in CPU cache and durable in PM. This can lead to inconsistency when concurrent threads access data that is not yet durable [15]–[17]. Persistent CPU cache eliminates this gap, thereby presenting an excellent opportunity to optimize the concurrency control in PM programming. In particular, we can leverage HTM (Hardware Transactional Memory) [18]–[21] to design lock-free data structures with the guarantee of durable linearizability. HTM provides a simple and efficient interface to atomically make the effect of multiple-word updates visible in a lock-free manner. With the persistent CPU cache, HTM can further ensure the persistence of those updates, which makes HTM well-suited for building data structures with durable linearizability.

In the past few years, PM-based persistent indexes [22]–[30] have received increasingly high interest. However, most of them are designed for platforms with volatile CPU cache. With persistent CPU cache, we found that simply removing flush instructions from those indexes does not yield a significant performance improvement for two reasons: (1) Existing designs do not fully leverage the persistent CPU cache to reduce PM writes. Without a cache-friendly design to increase write locality, PM writes are not reduced by removing flush instructions since dirty cachelines will eventually be evicted to PM media. Worse even, random cacheline evictions might cause write amplification due to the mismatch between the access granularity of PM media and cacheline-size, further degrading the write performance. (2) Previous PM-based indexes employ inefficient protocols to ensure durable linearizability.

*Linpeng Huang and Shengan Zheng are the corresponding authors.

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00295

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.

The majority of them adopt locks to coordinate concurrent accesses, preventing them from achieving high concurrency. Moreover, they produce excessive PM accesses to maintain various metadata (e.g., bitmap, lock, and version) to guarantee durable linearizability, resulting in performance degradation.

In this paper, we propose a highly scalable persistent hash index named Spash to address the above challenges. Spash fully exploits the benefits of persistent CPU cache to reduce the consumption of PM write bandwidth and the overhead associated with ensuring durable linearizability.

First, Spash integrates persistent CPU cache and HTM to achieve durable linearizability with low PM access overhead and high concurrency. Specifically, Spash employs a fine-grained extendible hash structure, which utilizes a volatile directory to divide hash space into multiple metadata-free segments. This architecture not only reduces PM accesses but also achieves strong compatibility with HTM transactions. To ensure durable linearizability and lock-free concurrent accesses, Spash employs a two-phase concurrency protocol and a collaborative staged doubling mechanism based on HTM. The two-phase concurrency protocol decouples hash operations into the preparation phase and transaction phase, minimizing data conflicts in the HTM transactions. Collaborative staged doubling divides the directory doubling of Spash into numerous minor stages that can be accomplished by either a doubling thread or concurrent threads. This mechanism obviates the capacity abort of HTM transactions and the blocking of concurrent threads during the directory doubling.

Furthermore, we propose an adaptive in-place update and a compacted-flush insertion mechanism for Spash to enhance the utilization of PM write bandwidth with persistent CPU cache. The adaptive in-place update absorbs the frequent updates to hot key-value entries in the CPU cache, thereby minimizing PM writes. Additionally, it adaptively flushes the updates of cold key-value entries to reduce write amplification resulting from the random cacheline eviction. Moreover, the compacted-flush mechanism compacts the consecutive small insertions in the persistent CPU cache and asynchronously flushes them back to PM in XPLine-sized (the access granularity in PM media) chunks, thereby reducing write amplification.

In summary, the contributions of this paper include:

- We thoroughly investigate the characteristics of PM and the impact of persistent CPU cache on PM systems. Based on this, we conclude several valuable design principles for PM systems on platforms with persistent CPU cache.
- We propose Spash, the first persistent hash index that fully exploits the benefits of persistent CPU cache. Spash integrates the persistent CPU cache and HTM to minimize the PM access overhead and enable lock-free concurrency.
- For Spash, we further propose adaptive in-place update and compacted-flush insertion, which fully leverage the persistent CPU cache to improve write performance.
- We conduct comprehensive evaluations on the performance of Spash and compare it with the state-of-the-art persistent hash tables. In YCSB evaluation, Spash outperforms other counterparts by up to 19.6×.

II. Background and Motivation

In this section, we begin by introducing the background of persistent memory in Section II-A. Then, we conduct an in-depth analysis of how persistent CPU cache influences the flush strategies for PM writes in Section II-B. Finally, in Section II-C, we explore the opportunities for designing highly scalable concurrent crash-consistent data structures with persistent CPU cache.

A. Persistent Memory

Persistent memory (PM) (e.g., MRAM [31], Re-RAM [32], PCM [33]) provides many attractive features, such as byte addressability, and DRAM-comparable performance. In the past decade, numerous use cases of PM, such as databases [34]–[36], key-value store [37]–[39], and file systems [40], [41], have been studied in both academia and industry.

Intel Optane DCPMM is the first commercially available PM product, which has two generations so far, AEP (Apache Pass, 2019) and BPS (Barlow Pass, 2021). The key distinction between AEP and BPS is their persistence domain. AEP-equipped platforms support the ADR mechanism [10], which ensures that data residing in the *write pending queues* (WPQ) can be flushed into the PM after a power failure. However, the CPU cache is still excluded from the persistence domain. Programmers need to issue explicit cacheline flush instructions (e.g., `clwb`, `clflush`, and `clflushopt`) and memory barriers (e.g., `mfence`, and `sfence`) to ensure data persistence. BPS-equipped platforms introduce a new mechanism called eADR [12], which further incorporates CPU cache into the persistence domain with enhanced backup energy. With the persistence guarantee of CPU cache, cacheline flush instructions are no longer necessary for data persistence.

Through a comprehensive evaluation of Optane DCPMM, we conclude the unique PM characteristics as follows:

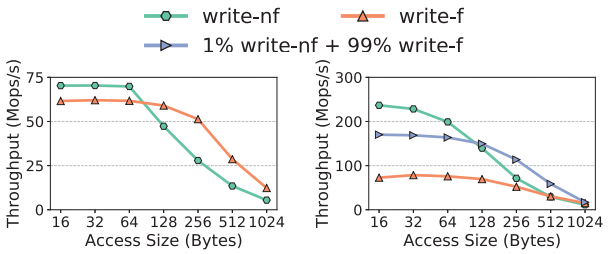
Observation 1: PM has lower write bandwidth and higher read latency than DRAM. Previous works [42] found that the access granularity of physical media (3D-XPoint) in Optane PM is 256-byte (XPLine). Therefore, accessing PM with blocks that are a multiple of XPLine-aligned can maximize the utilization of memory bandwidth. In our testbed (detailed configuration is described in Section VI), PM writes with 256-byte access granularity can achieve peak bandwidth (∼15GB/s), which is 5× lower than DRAM (∼75GB/s). Although PM read has a much higher bandwidth (∼50GB/s), the latency of PM read (∼350ns) is 3.5× higher than DRAM (∼100ns).

Hence, for PM-based systems, we have ***DP (Design Principle) 1: Avoid excessive PM accesses and take the XPLine-access granularity into consideration.***

B. The Flush Strategies for PM Writes

As mentioned in Section II-A, the persistent CPU cache obviates the need for cacheline flush instructions to ensure the durability of PM writes. Therefore, programmers have the flexibility to decide whether and when to use cacheline flush instructions for optimizing performance. In this section, we conduct a thorough analysis of the impact of persistent CPU

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.



(a) Uniform access distribution (b) Zipfian access distribution

Fig. 1. The PM write performance under different flush strategies. (56 threads)

cache by evaluating the performance of `write-f` (`store` that is followed by a flush instruction) and `write-nf` (`store` that is not followed by a flush instruction) under various access granularities and access patterns. Our evaluation leads to three observations:

Observation 2: Using `write-nf` for cold memory regions with more than 1 cacheline causes write amplification. Figure 1(a) shows the performance of PM write under uniform access distribution. In the uniform access pattern, the majority of writes access the cold memory regions and cannot hit CPU cache. In such a situation, we observe that the `write-nf` has lower throughput than `write-f` when the access size is larger than 1 cacheline (64-byte). This is due to the write amplification caused by the asynchronous cacheline eviction [42], [43] and the XPLine access granularity of PM. When we modify the cold memory chunks larger than 1 cacheline, multiple cachelines are likely to be evicted to PM in random order if we do not explicitly use flush instructions. Therefore, the large sequential writes will be divided into multiple cacheline-sized random writes, which mismatches the internal access granularity of PM (XPLine), leading to the write amplification. In contrast, the flush instructions after the write operations allows PM to aggregate those sequential writes in a small XPBuffer [42], maximizing the utilization of PM write bandwidth.

Observation 3: Using `write-nf` for hot memory regions saves PM write bandwidth. Figure 1(b) shows the performance of PM write under the zipfian access distribution (skewness=0.99), in which most memory accesses are concentrated in small hot regions. We discover that using `write-nf` has a significantly higher throughput than using `write-f`. The reason is that the writes on the hot memory region are likely to hit CPU cache, without consuming PM write bandwidth. The flush instructions for the hot memory region produce unnecessary data movement between CPU cache and PM media. Meanwhile, we found that for access sizes larger than 64-byte, using `write-nf` exclusively for the top 1% hot memory region outperforms using 100% `write-nf`. This is because using `write-f` on the cold memory regions that are larger than 1 cacheline can avoid the write amplification, which is already mentioned in ***Observation 2***.

Observation 4: Using `write-nf` is the best choice when the access size is less than a cacheline (64-byte). As shown in Figure 1(a) and 1(b), using `write-nf` achieves the highest throughput when the access size is less than a cacheline. The reason is that `write-nf` can not only exploit the CPU cache to conserve PM bandwidth but eliminate the latency of the flush instructions. Meanwhile, `write-f` can not alleviate write amplification when the access size is within a cacheline.

Therefore, for the usage of flush instructions in PM-capable platforms with the persistent CPU cache, we have ***DP2: Use `write-f` for the writes on cold memory regions that are larger than 1 cacheline. Use `write-nf` for the writes on hot memory regions or the small writes within 1 cacheline.***

C. Concurrent Crash-consistent Data Structures

In this section, we first present the influence of persistent CPU cache on the design of concurrent data structures featuring durable linearizability. Following this, we explore the potential of integrating persistent CPU cache and Hardware Transactional Memory (HTM) to enhance the scalability of concurrent crash-consistent data structures.

1) Durable linearizability: The correctness standard for PM-based concurrent crash-consistent data structures is durable linearizability [14], [15], which consists of the following two aspects:

- **Concurrent consistency**. In the absence of a crash, concurrent operations in PM-based data structures should ensure linearizability just as the DRAM-based data structures. Linearizability requires that each concurrent operation take effect atomically at some point during its execution [44].
- **Crash consistency**. After a crash, all previously completed operations should remain completed and their effects should remain visible after a crash. The operations not completed upon a crash should provide all-or-nothing semantics.

For PM-capable platforms with volatile CPU cache, it is challenging to preserve durable linearizability since there is a gap between data being visible in CPU cache and being durable in PM. The durable linearizability might be violated if a crash happens when the effect of the operations is visible but not durable. In such a situation, the effect of those operations can be observed before the crash but not after it, which violates the crash consistency.

Fortunately, the gap between visibility and durability is eliminated with the persistent CPU cache, which facilitates the design of concurrent crash-consistent data structures. The persistent CPU cache makes it possible to adopt hardware transactional memory (HTM) to effectively implement lock-free and durable linearizable data structures in PM.

2) Hardware Transactional Memory: HTM is a hardware approach to making a batch of writes within a transaction atomically visible in CPU cache. Taking Intel's Restricted Transactional Memory (RTM) [45] as an example, programmers can execute a transaction by wrapping the critical section with the `_xbegin()` and `_xend()`. If the transaction commits without any conflicts with concurrent memory operations, the writes in the transaction will atomically become visible in CPU cache. Otherwise, any conflicts detected by the hardware will result in a transaction abort, which rolls back all operations and restarts the transaction. Compared with the lock-free

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.

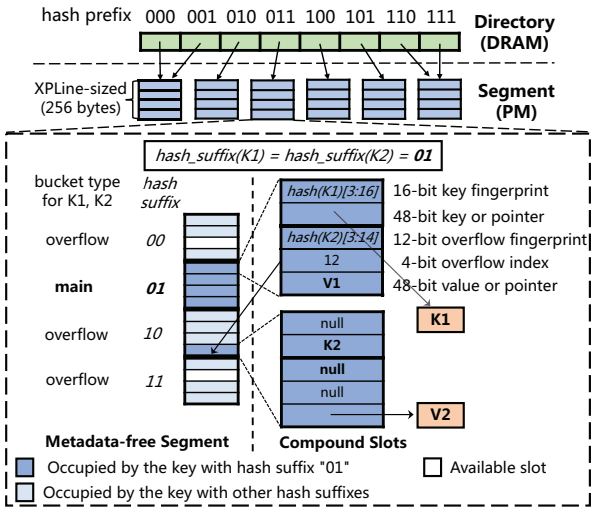


Fig. 2. The overall architecture of Spash.

data structures relying on the atomic memory operations (e.g., CAS) that only ensure the atomicity of 8-byte writes, HTM offers the assurance of the atomicity for a batch of writes (within the L1 Cache capacity), greatly simplifying the lock-free design.

Previously, HTM was incompatible with PM in the platforms with volatile CPU cache. To achieve durable linearizability, programmers need to issue flush instructions in the HTM transaction, ensuring the durability of the writes. However, the flush instructions will trigger the abortion of the HTM transaction [46], leading to an incompatibility between HTM and PM programming. If the flush instructions are executed after the completion of HTM transaction, the non-durable effect of transaction operations can be seen by concurrent threads, which violates the requirement of durable linearizability.

With the persistent CPU cache, HTM can ensure both visibility and durability of the writes without using flush instructions since the data in the CPU cache is guaranteed to be persisted. Therefore, we can use the simple interface provided by HTM to implement durable linearizable data structures, which dramatically reduces the complexity of design and programming. Meanwhile, the lock-free nature of HTM also improves the scalability of the data structures.

Nevertheless, there are still some challenges that prevent us from directly using HTM for all memory operations [19]. The first one is the capacity limitation. As HTM uses the CPU's private cache to track its read/write set, the transaction with large memory footprints will suffer the *capacity abort*. The second one is data conflicts. HTM transactions may experience frequent *conflict aborts* due to long code paths or high contentions. Hence, additional designs are needed to address the above two problems.

III. Spash

In this section, we present Spash, a scalable persistent hash index that fully exploits the benefits of persistent CPU cache.

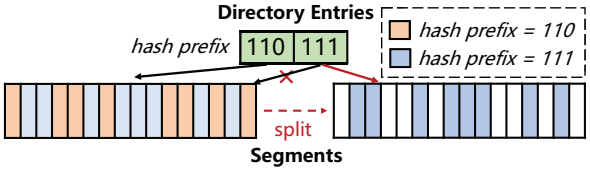


Fig. 3. The segment split of Spash.

We begin by describing the hash structure in Section III-A, which minimizes the PM accesses in hash operations and facilitates the HTM concurrency control (Section IV). To further leverage the persistent CPU cache to save scarce PM write bandwidth for Spash, we introduce adaptive in-place update in Section III-B and compacted-flush insertion in Section III-C. Then, we summarize the execution flow of Spash and propose a pipeline optimization in Section III-D.

A. Hash Structure

For Spash, we propose a fine-grained extendible hash structure, metadata-free segment design, and compound slots, which mainly aim to achieve the following three design goals. First, we minimize the number of PM accesses based on ***DP1*** rather than reducing the flush instructions in the previous works since Spash targets platforms with persistent CPU cache. Second, the hash structure design has to consider both the limitations and the benefits of HTM as we mentioned in Section II-C2 since we employ HTM to ensure durable linearizability. Third, the hash structure should have a high memory utilization and alleviate the performance degradation caused by resizing.

Fine-grained Extendible Structure. As shown in Figure 2, Spash is a variant of extendible hash [7], [22], which employs a DRAM-based directory to divide the hash space into multiple fine-grained (XPLine-sized) segments in PM. Through the directory, each hash operation uses the highest few bits of the key hash (hash prefix) to locate the address of the segment, which stores the target key-value entries. Spash achieves dynamical expansion and shrinkage by splitting or merging the fine-grained segments. As depicted in Figure 3, the key-value entries with the hash prefix "11" are initially stored in the left segment, which is shared by directory entries "110" and "111". Once this segment reaches capacity, it initiates a split operation to rehash the key-value entries with the hash prefix "111" into a newly allocated segment and redirects the pointer in the directory entry "111" to the new segment. The directory doubling is triggered when the directory cannot accommodate the new segment, which will be discussed in Section IV-B. Conversely, segment merging is the reverse process of segment splitting.

The fine-grained extendible hash structure accomplishes the design goals of Spash in the following ways. First, the fine-grained resizing (split/merge) has better compatibility with HTM-based concurrency control (see details in Section IV) and low PM access overhead. Large-grained segment resizing [7], [22] and level-based resizing [23], [47] in the

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.

previous PM-based hash indexes are not suitable for HTM-based concurrency control due to their large memory footprint, which increases the possibility of data conflicts or even triggers the capacity abort. Therefore, fine-grained segment facilitates HTM-based concurrency control during the resizing. Meanwhile, the resizing-incurred performance degradations in the previous works [7], [22], [23], [47] will also be addressed in Spash due to the lightweight resizing. Moreover, the XPLine-sized segment in Spash matches the internal access granularity of PM media, which improves the bandwidth utilization of resizing.

Second, the PM access overhead during the base hash operation (Insert/Update/Delete/Search) is also reduced. During a base operation, the directory access is absorbed in DRAM and the segment access only incurs an XPLine-sized PM access in the worst case.

Metadata-free Segments. To further reduce the number of PM accesses during the hash operations without sacrificing memory utilization, we propose a metadata-free segment structure with the support of HTM and persistent CPU cache.

As shown in Figure 2, Spash directly divides a segment into four cacheline-sized buckets without maintaining any metadata in the header of the segment. To process a query, Spash locates the main bucket of a requested key in the target segment according to the lowest 2 bits of the key hash (e.g., bucket 01 for $K1$). Then, Spash inserts/updates/deletes/searches a key-value entry in the main bucket, which only incurs one cacheline-sized PM access. The additional PM accesses on the metadata are eliminated in Spash. Specifically, the various metadata (e.g., lock, version, and bitmap) used in the previous works [7], [22], [23] for crash consistency and concurrent consistency is unnecessary for Spash, since the durable linearizability of Spash is guaranteed by HTM and persistent CPU cache (see details in Section IV). Meanwhile, for other metadata to accelerate searching (e.g., fingerprints), Spash offloads them to compound key-value slots, which will be discussed later.

To avoid segment splitting immediately when one of its buckets is full, we propose circular probing to improve the memory utilization of segments with few additional PM accesses. For insertions in Spash, if the main bucket of the requested key is full, the key-value entry can be inserted into the other three buckets, named overflow buckets. Specifically, to find a free slot for an insertion, Spash starts the probing procedure from its main bucket and proceeds in a circular order to check the overflow buckets. For example, $(K2, V2)$ in Figure 2 is inserted into the first free slot of the overflow bucket 10 if its main bucket 01 is full. This design ensures that the segments in Spash do not split until all of its buckets are full. The additional probing overhead is tolerable in the fine-grained segments since it happens infrequently and the probing buckets reside in the same XPLine as the main bucket.

Compound Slots. Spash applies compound key-value slots to support variable-sized key-value entries and reduce unnecessary PM reads during the search operations. As shown in Figure 2, each bucket contains four 16-byte key-value slots, which store inline keys and values for small-sized key-value entries (e.g., K2, V1) while storing pointers to the actual key and value for large-sized key-value entries (e.g., K1, V2). Since 48-bit is sufficient for a pointer to indicate the memory address, Spash reserves the highest 16-bit of both key and value fields to reduce PM reads during the search operations.

To accelerate the search for key-value entries in the overflow buckets, Spash stores their hints in the value fields of the main bucket. As shown in Figure 2, the main bucket of $K2$ stores the lowest 3-14 bits of $K2$'s hash value (`overflow fingerprint`) and the position of $K2$ in the segment (`overflow index`). The `overflow fingerprint` can be used to filter the unmatched key without probing the overflow buckets, and the `overflow index` can directly identify the position of the target key-value entry. Moreover, Spash stores the `key fingerprint` (the lowest 3-16 bits of the hash value) in the key field to reduce PM reads for searching the large-sized key-value entries. In this way, hash operations can filter the unmatched keys through `key fingerprint`, reducing PM reads caused by the pointer dereference.

PM Overhead Analysis. For inlined key-value entries, search operations in Spash produce a cacheline-sized PM read to access a main bucket in most cases. In rare cases ($\sim 9\%$), search operations need to access an extra cacheline-sized overflow bucket that is stored in the same XPLine as the main bucket. Compared with the search operation, the write operations (Insert/Update/Delete) in Spash produce one more cacheline-sized PM write to modify a key-value slot. For large-sized key-value entries, all hash operations should further access the key or value via the pointers in the key-value slot.

B. Adaptive In-place Update

The real-world applications often have obvious hotspots, as well as variable-sized key-value entries [29], [48]–[51]. In this section, we propose an adaptive in-place update strategy, which leverages persistent CPU cache to maximize the utilization of PM write bandwidth in such a situation. We first describe our update strategies for the key-value entries with various hotnesses and sizes. Then, we introduce our lightweight hotspot detector, which identifies the hotness of the key-value entries with low overhead.

Update Policy. Spash employs in-place update for variable-sized key-value updates. Compared with the out-of-place update that was employed in most previous works [16], [28], [47], in-place updates for frequently accessed key-value entries are likely to be completed in CPU cache, conserving scarce PM write bandwidth. Meanwhile, in-place update produces fewer PM writes than out-of-place update since it does not need to install a new pointer in the hash structure.

Driven by the ***DP2*** mentioned in Section II-B for the usage of flush instructions, Spash employs different flush strategies to accommodate the key-value entries with different hotnesses and sizes. As shown in Table I, for hot key-value entries, Spash employs in-place update without using flush instructions. It is because that in-place updates to hot data are likely to hit CPU cache, which reduces writes to PM media. The flush

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.

TABLE I
ADAPTIVE FLUSH STRATEGY FOR THE UPDATES OF KEY-VALUE ENTRIES WITH DIFFERENT HOTNESSES AND SIZES.

Hotness	Hot	Cold	
Size	All	$\leq$ 64-byte	$>$ 64-byte
Flush strategy	*w/o flush*		*w/ flush*

instructions should be removed since they cause repeated data movements between CPU cache and PM, wasting PM write bandwidth. For cold key-value entries, if their sizes are larger than 64 bytes, Spash issues an asynchronous flush instruction after each update. The reason is that those cold key-value entries will hardly be accessed again in a short time. The flush instruction can force multiple modified cachelines to sequentially write back to PM, making the best use of PM write bandwidth. Otherwise, dirty cachelines will randomly write back to PM according to the cache replacement policy, leading to the write amplification. If the sizes of the cold key-value entries are smaller than 64 bytes, Spash omits flush instructions. The reason is that actively flushing a single cacheline won't alleviate write amplification.

We fully leverage the persistent CPU cache to overcome two obstacles that prevent the use of in-place update in the previous PM-based indexes. First, in-place update is incapable of atomically modifying and persisting the key-value entries that are larger than 8-byte, resulting in anomalies, such as partial write [52]–[54], and dirty read [16], [55], which violate the requirement of durable linearizability. Spash guarantees the atomic persistence of in-place updates by integrating HTM transactions and persistent CPU cache, which will be discussed in Section IV in detail. To avoid transaction aborts, cacheline flush instructions (e.g., `clwb`) can asynchronously be executed after the completion of the HTM transaction. Note that `ntstore` should not be used for cold key-value updates since it is incompatible with HTM transactions [46]. Second, for platforms with volatile CPU cache, the hot key-value entries are not benefit from in-place update. The reason is that each update must be synchronously flushed to PM for persistence, which consumes significant PM write bandwidth even when the writes hit CPU cache. With persistent CPU cache, Spash adaptively decides whether to issue flush instructions based on the hotnesses and sizes of the updated key-value entries for the best utilization of PM write bandwidth.

Hotspot Detector. To support the adaptive update strategy, we propose a lightweight hotspot detector to identify the hotness of the key-value entries. The basic idea of our hotspot detector is to maintain a hot-key list in DRAM, which contains the frequently accessed keys. Spash decides which update strategy to use by checking if the requested key is in the hot-key list.

Spash proposes the hash partitioning approach to reduce the overhead of scanning the hot-key list. Spash divides the hash structure into multiple partitions according to the highest p bits of the hash prefix. For each partition, the hot-key list maintains q local hot keys, which is updated according to the LRU replacement policy (The detailed choice of p and q will be discussed in Section VI). In this way, the overhead of identifying the hotness of the key can be reduced dramatically since Spash only needs to scan the local hot-keys of the partition where the requested key resides. The size of the hot-key list is small enough in our implementation to fit in the CPU cache, which also reduces the overhead of our hotspot detector. Moreover, our hotspot detector can accurately find out hot keys through the hash partition. Due to the randomness of the hash function, the union of the local hot-keys of each partition can effectively represent the global hot-keys.

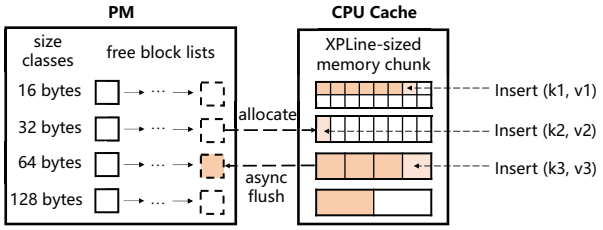


Fig. 4. Compacted-flush insertion.

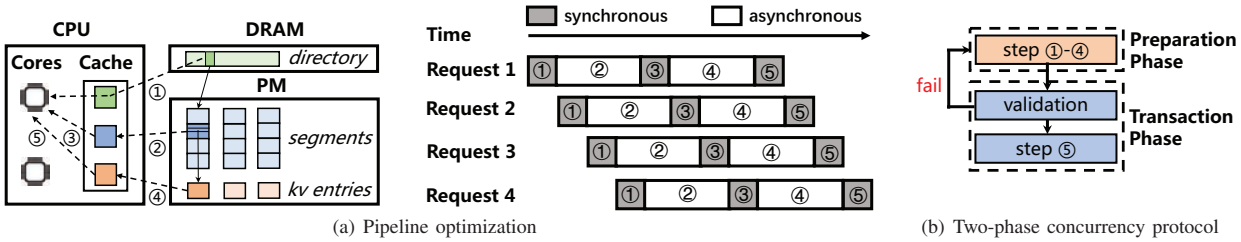
C. Compacted-flush Insertion

To support the insertion of variable-sized key-value entries in persistent indexes, a widely used approach is out-of-place insert, which installs a pointer in the index structure to indicate the actual key-value entries. However, this approach suffers from severe write amplification when inserting small-sized key-value entries ($\leq$128-byte) since the access granularity of PM physical media is 256-byte. To deal with this problem, we propose a compacted-flush mechanism for inserting variable-sized key-value entries, which leverages the persistent CPU cache to fully utilize PM write bandwidth.

The main idea of compacted-flush insertion is to compact multiple small insertions in the persistent CPU cache and then asynchronously flush them to PM in an XPLine-sized granularity. As shown in Figure 4, to serve small insertions ($\leq$128-byte), Spash allocates a thread-local XPLine-sized memory chunk and populates it in an append-only manner. Hence, the consecutive small random insertions tend to compact in the persistent CPU cache. Meanwhile, the persistent CPU cache allows each insertion to commit without initiating a synchronous flush instruction, which avoids high latency and bandwidth-wasting induced by the repetitive flushes to the same cacheline [9], [39]. Instead, Spash asynchronously issues a flush instruction when an XPLine-sized memory chunk is exhausted, preventing the random cacheline eviction from causing write amplification (***DP 2***). With the compacted-flush mechanism, Spash achieves the best utilization of PM write bandwidth for inserting small-sized key-value entries.

To manage XPLine-sized memory chunks with different size classes, Spash employs a state-of-the-art persistent allocator, DCMM [29], which maintains multiple free block lists with various size classes for each thread. In particular, the block classes with small sizes ($\leq$128-byte) are managed in XPLine-sized chunks to better support our compacted-flush insertion.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.



(a) Pipeline optimization (b) Two-phase concurrency protocol

Fig. 5. The execution framework of hash operations.(①: Get the address of segment; ②:Load the main bucket; ③: Locate the key-value slot; ④: Load the key-value entry; ⑤: Process (insert/update/delete/search) the key-value entry.)

D. Execution Flow

In this section, we first summarize the execution flow of hash operations in Spash. Then, we introduce a pipeline optimization for Spash to hide long PM read latency and thoroughly squeeze the high read bandwidth out of PM.

As shown in Figure 5(a), Spash divides the execution of hash operations into five steps. The first four steps are shared by all hash operations (Insert/Update/Delete/Search). First, Spash obtains the address of the target segment from the volatile directory according to the hash prefix of the requested key (step 1). Second, Spash loads the main bucket of the target segment from PM to CPU cache according to the hash suffix of the requested key (step 2). Third, Spash locates the key-value slot corresponding to the requested key following the procedure described in Section III-A (step 3). Fourth, for large-sized key-value entries, Spash loads the corresponding key-value pairs from PM into CPU cache according to the pointer stored in the key-value slot (step 4). For inlined key-value entries, Spash directly retrieves the key-value information from the target key-value slot.

After locating the requested key in the hash table, Spash processes the target key-value entry according to the operation type (step 5). Search operations directly return the value of the requested key. Update operations perform adaptive in-place update to the corresponding values while delete operations set the corresponding keys (pointers) to null. For insert operations, Spash inserts a key-value entry into a free slot and adopts the compacted-flush approach to best utilize PM write bandwidth.

To fully utilize high PM read bandwidth and hide high PM read latency, Spash employs a pipeline optimization on the execution of hash operations. As shown in Figure 5(a), Spash allows each CPU core to concurrently execute multiple requests in a pipelined manner. Specifically, the PM read operations in step 2 and 4 can be executed asynchronously by using memory prefetch instruction. During the asynchronous PM reads, CPU core is free to execute the critical tasks (step 1 and 3) of the subsequent few requests if the request queue is not empty. Consequently, multiple PM read operations can be executed in parallel to exploit high PM read bandwidth.

Moreover, it is necessary to choose a proper pipeline depth (the number of parallelly executed requests in a single core) to avoid increasing operation latency. Our evaluation shows that 4 parallelly executed requests are sufficient to hide PM read latency and utilize most of PM read bandwidth.

IV. Concurrency Control

In this section, we introduce our HTM-based concurrency control strategy, including a two-phase concurrency protocol (Section IV-A) for base operations and a collaborative staged doubling mechanism (Section IV-B) for the directory doubling.

A. Two-phase Concurrency Protocol

To achieve lock-free concurrency control and guarantee durable linearizability of Spash, we propose a two-phase concurrency protocol based on HTM. As shown in Figure 5(b), Spash decouples the pipelined hash operations into two phases: preparation phase and transaction phase. The preparation phase includes step 1-4 of pipelined hash operations, which locates the target key-value entry. The transactions phase only includes step 5, which processes the key-value entry.

The transaction phase employs HTM to ensure the atomicity of memory operations during the key-value processing. Although multiple words need to be modified during the key-value processing (e.g., in-place update of large-sized key-value entries, segment split initiated by inserting to a full segment), HTM transactions can atomically make the effect of those operations visible and durable as mentioned in Section II-C2, thereby ensuring the durable linearizability of concurrent hash operations. Moreover, the lock-free nature of HTM transactions also improves the multi-core scalability of Spash.

Meanwhile, we leave the preparation phase outside of the HTM transaction, which brings the following two benefits: (1) It reduces the *conflict abort* in HTM transactions. Excluding the computations and the memory load operations from HTM shortens the critical path of the transaction, thereby reducing the possibility of conflicts with concurrent operations. (2) It avoids the overlapping of HTM transactions during the pipeline executions. HTM does not support the creation of multiple overlapping transactions in a pipelined manner. Hence, we only wrap step 5 into the HTM transaction, so that multiple requests in the same pipeline batch can serially execute their HTM transactions as shown in Figure 5(a).

Moreover, the transaction phase includes an additional validation step to ensure the correctness of the preparation phase. Since the preparation phase is not protected by HTM, the concurrent operations might move the target key-value entry to a different segment or slot from the one obtained in Step 1 and Step 3. Hence, in the transaction phase, Spash checks whether the segment address and the target key-value slot have

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.

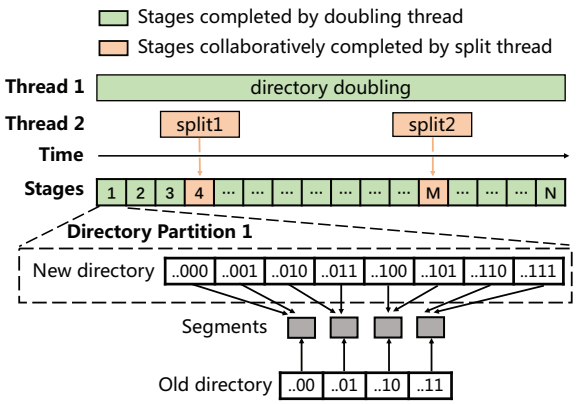


Fig. 6. Collaborative Staged Doubling

been changed. If so, Spash should actively abort the HTM transaction and retry the preparation phase.

We also provide a fallback path for the transaction phase when the time of HTM aborts in a single hash operation reaches a preset threshold. In such a rare case, the segment lock stored in the first bit of its corresponding directory entry will be taken. The concurrent operations on the locked segment do not start the HTM transaction until the lock is released.

B. Collaborative Staged Doubling

Directory doubling happens when the directory lacks sufficient space for the insertion of newly allocated segments. Directly using an HTM transaction to wrap the execution of directory doubling often lead to *capacity abort* since it involves a large memory footprint. To deal with this problem, we propose collaborative staged doubling to ensure the concurrent consistency of directory doubling. Collaborative staged doubling avoids the *capacity aborts* by accomplishing the doubling procedure in numerous small stages and improves the scalability of Spash by enabling the concurrent operations collaboratively completing the directory doubling (note that directory halving is handled similarly).

As shown in Figure 6, unlike the traditional design, collaborative staged doubling contains multiple small stages. During the directory doubling, Spash first allocates a double-sized directory, then logically divides the old directory into multiple cacheline-sized directory partitions. Each stage in the collaborative staged doubling is responsible for doubling a small directory partition. Specifically, for each stage, Spash invokes an HTM transaction to copy the segment pointers of a directory partition from the old directory to the new directory. In this way, the *capacity aborts* can be avoided since each directory partition has a small memory footprint.

Moreover, collaborative staged doubling does not block concurrent base operations. For the concurrent modifications (e.g., split) on the directory during the doubling procedure, Spash allows them to operate on the new directory to avoid the *lost update* anomaly [16]. As shown in Figure 6, if a split operation (e.g., split1 and split2 in thread2) needs to modify a directory partition that has not yet been copied to the new directory, it will initiate an HTM transaction to collaboratively assist the doubling thread in completing the corresponding pending stage. Then, the split operation can safely modify the new directory despite the directory doubling not having been completed. For concurrent reads, if the corresponding directory partition has not finished its doubling procedure, they obtain the segment pointers in the old directory. Otherwise, they will be redirected to the new directory to get the latest segment pointers. Collaborative staged doubling can significantly improve the overall throughput and reduce the tail latency by enabling lock-free concurrent accesses.

V. Discussion

Generality of the Spash's design to other persistent indexes. Most of the design of Spash can be applied to other PM-based indexes (e.g., B^+-Tree, and radix tree). Specifically, the adaptive in-place update and compacted-flush insertion techniques can be directly employed on any persistent indexes to improve write performance. The HTM-based concurrency control is also applicable for ensuring both durable linearizability and high scalability of other persistent data structures. The pipeline optimization can be used in other index structures to fully utilize memory bandwidth.

Generality of the Spash's design to other PM products. In addition to Intel Optane PM, many manufacturers (e.g., Fujitsu [56], Samsung [31], Xi'an UniIC [57], and SK Hynix [58]) are developing their PM products due to its huge application value and mature ecosystem. Although Spash is implemented atop the Optane DCPMM, our design is likely to take effect on other PM products for the following reasons. First, most of the non-volatile media exhibit similar characteristics, such as asymmetric read/write performance, higher latency and lower bandwidth than DRAM. Second, CPU cache persistence techniques (e.g., eADR [12], battery-backed cache [59], non-volatile cache [60]–[62]) is promising to be adopted in the future PM-equipped platform since it has been widely studied in the last few years.

Generality of the Spash's design to the future CXL devices. Compute express link (CXL) [63] is a recently published industry open standard for high-speed communication. CXL has attracted great attention from industry and academia since it can be used to expand both memory capacity and memory bandwidth. The design of indexes for CXL devices can also employ the techniques proposed in Spash since CXL-based memory devices have higher latency than DDR-based DRAM, which is similar to PM.

VI. Evaluation

In this section, we evaluate the performance of Spash and compare it with the state-of-the-art persistent hash tables. We first describe our experiment setup (Section VI-A). Then, we introduce the evaluation of both micro-benchmarks (Section VI-B) and macro-benchmarks (Section VI-C). Finally, we conduct a comprehensive analysis of the impact of each design component in Spash (Section VI-D).

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.

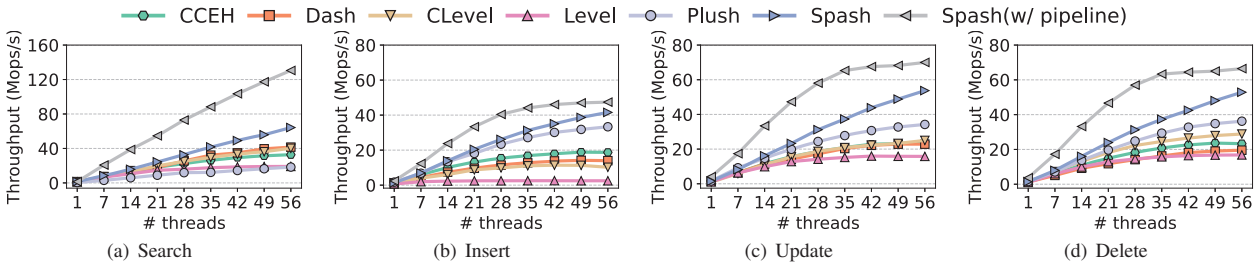


Fig. 7. The throughput of hash operations. (Uniform access distribution, inlined key-value entries)

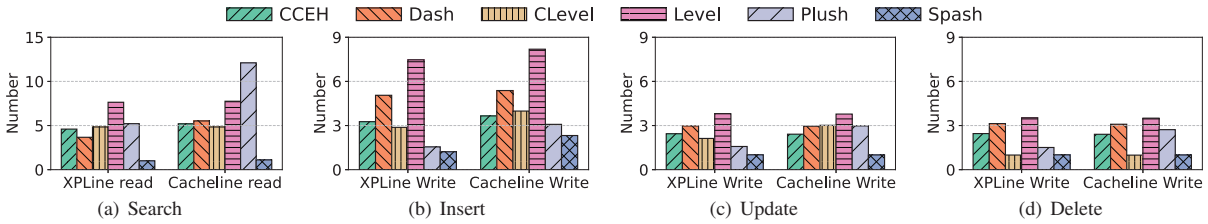


Fig. 8. The average number of XPLine and cacheline accesses to PM during the hash operations.

A. Experiment Setup

Testbed. Our testbed machine is a dual-socket server with two Intel Xeon Gold 6348 CPUs, the third-generation Xeon Scalable processors that support eADR and TSX. Each CPU has 28 cores (56 hyper-threads) and a shared 42MB L3 cache, while each CPU core has a 48KB L1D cache, 32KB L1I cache, 1280KB L2 cache. Each socket is equipped with 8×32GB DDR4 DRAM and 8×256GB Barlow Pass (BPS) DIMM.

Compared Systems. We compare Spash against six state-of-the-art persistent hash, including CCEH [22], Dash [7], Level hashing [23], CLevel [47], Plush [64], and Halo [65]. Among them, CCEH, Dash, and Level can not support both variable-sized keys and values. Therefore, for micro-benchmarks (Section VI-B) with small-sized key-value entries that can be accommodated within 8B-8B key-value slots, we employ the original implementations of these baseline systems. Meanwhile, we implement extended versions for CCEH, Dash, and Level hashing to handle the macro-benchmarks (Section VI-C) containing large-sized key-value entries (ranging from 16B to 1024B). These extended implementations manage variable-sized key-value entries by storing a pointer to the actual key or value and adopting out-of-place update/insert, which is widely used in previous works [16], [28], [47]. For fairness and simplicity, we remove cacheline flush instructions and the memory barriers for persistence (we retain the memory barriers for thread synchronization) from the compared systems, enabling them to leverage the persistent CPU cache.

B. Micro-benchmarks

In this section, we evaluate the performance of each hash operation (Search, Insert, Update, and Delete) and the load factors for different persistent hash indexes. We initialize each persistent hash with 20M inlined key-value entries. Then, we run individual base operations with uniform distribution. Specifically, we execute 8 billion insert, 8 billion search, 8 billion update, and 8 billion delete in order. Meanwhile, we employ `ipmctl` [66] to calculate the average number of XPLine and cacheline accesses to PM during each base operation. Furthermore, we record the load factor (the number of inserted key-value entries divided by the capacity of hash tables) of each index during the execution to evaluate their memory utilization. Halo is excluded from the micro-benchmark since it crashes during the executions. The primary reason is that Halo needs to maintain a complete hash table in DRAM for performance improvement, resulting in the exhaustion of DRAM space in a large dataset.

Read(Search). As shown in Figure 7(a), for search operations, Spash achieves 3.1∼7.1× higher throughput than other persistent hash tables under 56 threads. This is primarily because Spash produces minimal PM read in search operations by using a fine-grained extendible hash structure and metadata-free segments. As shown in Figure 8(a), Spash only reads 1.1 cacheline and 1.0 XPLine of PM in each search operation on average, which is 3.7∼10.9× lower than other counterparts. Moreover, the pipeline optimization further improves the search performance of Spash by 2.0× due to the full utilization of PM read bandwidth.

In contrast, Level hashing and CLevel need to read at most four buckets for each search operation, which is costly because these buckets do not reside in a contiguous memory region. Meanwhile, Level hashing and CCEH produce PM writes to maintain read locks, which slows down the search operations. Dash incurs multiple XPLine-sized bucket-reads for each search operation, which has a significantly higher overhead than Spash. Plush employs an LSM-Tree-based structure, requiring an average traveral of $O(logN)$ (N represents the number of key-values) levels to retrieve a key-value entry.

Write(Insert/Update/Delete). As shown in Figure 7(b)-7(d), Spash outperforms other hash tables by 1.4∼19.0× for insert, 2.0∼4.4× for update and 1.8∼3.9× for delete. We also attribute this to our fine-grained extendible architecture and metadata-free segment design, which reduces PM writes. As

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.

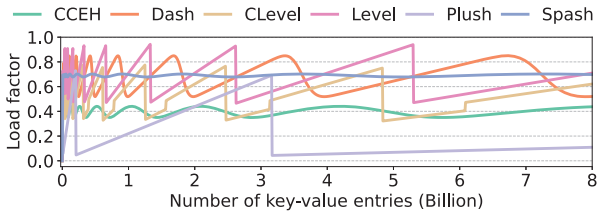


Fig. 9. The load factors of different hash indexes when varying the number of inserted key-value entries.

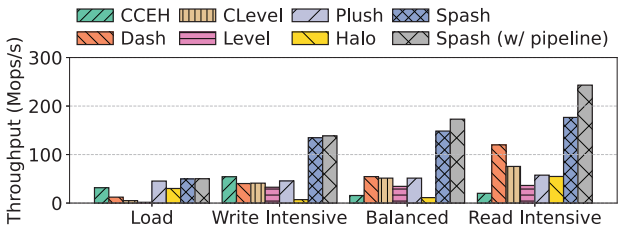


Fig. 10. YCSB throughput under inlined key-value entries. (Zipfian access distribution, inlined key-value entries, 56 threads)

shown in Figure 8, for each update or delete operation, Spash generates 1.0 cacheline write and 1.0 XPLine write on average, which is much lower than other persistent hash indexes. Moreover, Spash has minimal PM write overhead during the insert operation. Although each insert operation of Spash produces 2.0 cacheline write on average, it only writes 1.1 XPLine. This is because around 35% of cacheline write of insert operation comes from the split operations. A split operation must modify two XPLine-sized segments, each containing 4 cachelines from the same XPLine. Fortunately, the split operations are bandwidth-efficient due to the XPLine granularity of PM.

Other persistent hash tables produce more PM writes to maintain metadata for durable linearizability. Those metadata modifications are avoided in Spash by using HTM-based concurrency protocol, which ensures both concurrent consistency and crash consistency. Moreover, Level hashing and CLevel perform poorly in insertions owing to the large overhead of the full-table rehashing. In Spash, the full-table rehashing is obviated by using extendible hashing architecture. Although Plush adopts a DRAM buffer with write-ahead logs to reduce PM writes, it still generates more PM writes than Spash. The primary reason for this lies in its LSM-Tree-based structure, which leads to a large volume of PM writes when flushing DRAM buffer to PM and merging PM-based hash tables across different levels.

We also find that the pipeline design does not have a significant influence on the peak throughput for write-intensive workloads since PM write bandwidth is the performance bottleneck in this case.

Load Factor. We evaluate the memory utilization of hash tables by calculating their load factors with increasing key-value entries. As shown in Figure 9, Spash has a much higher load factor than CCEH due to the design of fine-grained extendible architecture and circular probing in the segments. Meanwhile, Spash achieves a similar average load factor with Dash and Level hashing. Although Dash and Level hashing have a higher maximum load factor than Spash, their load factors fluctuate greatly with the number of inserted key-value entries. The reason is that Spash dynamically allocates the fine-grained segments on demand while other hash tables perform level-based resizing or large-grained segment splitting. The load factor of Plush is relatively low and exhibits significant fluctuations as Plush allocates a 16× larger level when the current levels cannot accommodate new insertions.

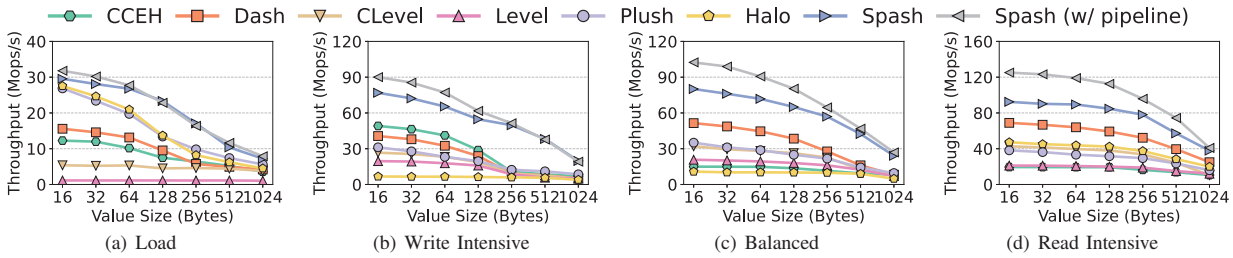
C. Macro-benchmarks

In this section, we conduct the evaluation on the YCSB workload, a real-world workload with skewed access patterns. The experiment with YCSB workloads consists of the load phase and run phase, both of which run under 56 threads. In the load phase, each persistent hash is initialized with 100M key-value entries. In the run phase, each hash table handles 100M queries, which are a mixture of read (Search) and write (Update). The key access pattern follows the zipfian distribution with the default zipfian parameter (0.99). We evaluate the YCSB performance under both inlined and variable-sized (16-1024 bytes) key-value entries.

Inlined Key-value. Figure 10 shows the performance of each persistent hash under load phase and three YCSB workloads: read-intensive (search:update = 90:10), write-intensive (search:update = 10:90), balanced (search:update = 50:50). Spash outperforms other hash tables by 1.1∼31.2× in the load phase and 2.0∼19.6× in three YCSB workloads. Our proposed HTM-based concurrency controls significantly improve the performance of Spash under skewed workloads by eliding locks for both read and write operations. Moreover, Spash performs the in-place update for inlined key-value entries. The updates for frequently accessed key-value entries are likely to be served in CPU cache, which saves PM write bandwidth.

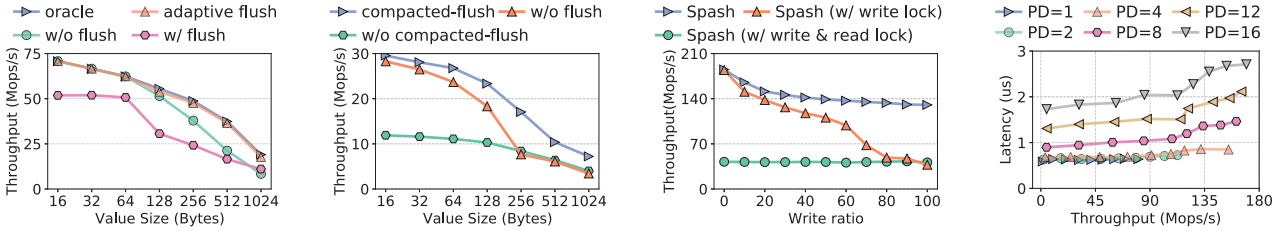
Other persistent hash tables can not perform as well as Spash due to their lock-based design or the under-utilization of CPU cache. Level hashing performs poorly across all three YCSB workloads because it uses locks for both read and write operations. Although CLevel is a lock-free hash table, it employs out-of-place update for any key-value entries, which can not utilize the CPU cache to absorb the updates of hot keys. As for Dash and Halo, their performance on the write-intensive workload is inferior to the read-intensive workload since they implement lock-free reads and lock-based writes. Halo performs worse than Dash, especially for write-intensive workloads, since its log-structured architecture produces more PM writes during the invalidation and reclamation of log entries. CCEH performs poorly in read-intensive and balanced workloads as it employs the read-write locks. The YCSB performance of Plush is also restrained by both lock-based out-of-place write and shared write-ahead logs. Additionally, Plush achieves comparable performance with Spash in the load phase since the merging overhead is relatively low when the number of key-values is small.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.



(a) Load (b) Write Intensive (c) Balanced (d) Read Intensive

Fig. 11. YCSB throughput under 16-byte key and variable-sized values. (Zipfian access distribution, 56 threads)



(a) The effect of adaptive in-place update. (Update-only workloads, zipfian access distribution, 56 threads)

(b) The effect of compacted-flush insertion. (Insert-only workloads, uniform access distribution, 56 threads)

(c) The effect of HTM-based concurrency control. (YCSB workloads, zipfian access distribution, inlined key-value entries, 56 threads)

(d) The pipelined search performance under different pipeline depth (PD) and thread numbers. (1,7,...,to 56 with a step of 7)

Fig. 12. The effect of each design component in Spash.

Variable-sized Key-value. Figure 11 shows the YCSB performance under 16-byte key and various value sizes. In the load phase, Spash delivers up to 28.0× higher throughput than the state-of-the-art hash tables. Besides the few PM writes mentioned previously, the compacted-flush insertion contributes the most to the high performance of Spash. The compacted-flush approach is capable of fully utilizing PM write bandwidth by compacting multiple small insertions ($\leq$ 128-byte) into an XPLine-sized memory chunk. In the meantime, Spash asynchronously flushes those insertions to PM in the XPLine-granularity, which further avoids write amplification induced by the random cacheline evictions.

In the run phase, Spash achieves up to 13.4× higher throughput than other counterparts in the write-intensive workloads. We attribute this to our proposed adaptive in-place update technique. Spash effectively identifies the hotspots of the workloads and removes the flush instructions for the updates of frequently accessed keys. In this way, the updates of the hot key-value entries are likely to hit the CPU cache, which drastically reduces the consumption of PM write bandwidth. Meanwhile, Spash actively flushes the update of cold key-value entries with large sizes ($>$ 64-byte), which effectively alleviates write amplification, saving scarce PM write bandwidth. As for other persistent hash tables, they use the out-of-place update for all circumstances, which prevents them from utilizing CPU cache to improve scalability. Moreover, Spash performs the best in the read-intensive workload and balanced workload due to both adaptive update strategies and pipeline framework as we mentioned above.

D. In-depth Analysis

We conduct four optimizations for Spash, including adaptive in-place update, compacted-flush insertion, HTM-based concurrency control, and pipeline executions, which efficiently support Spash to obtain high scalability. We evaluate the effects of those optimizations in this section.

Adaptive In-place Update. Our proposed adaptive in-place update mechanism significantly conserves PM write bandwidth in platforms with the persistent CPU cache by adopting different update strategies for the key-value entries with various hotnesses and sizes. To show the impact of the adaptive in-place update, we evaluate the YCSB throughput of Spash using the adaptive update strategy and other update strategies. As shown in Figure 12(a), the adaptive update strategy delivers 1.2∼2.2× higher throughput than in-place update (w/ flush) under various key-value sizes. The reason is that using flush instructions for the updates of hot key-value entries produces unnecessary PM writes. Meanwhile, adaptive update outperforms in-place update (w/o flush) by up to 2.1× when the value size is larger than 64-byte. This is because removing the flush instructions for the cold key-value entries with large sizes causes write amplification. In a nutshell, using adaptive update strategies for the key-value entries with various hotnesses and sizes can make the best use of PM write bandwidth.

To support the adaptive updates, we heuristically maintain a small hot-key list with 8K entries (each partition has two hot-keys), which works well for the workloads with different key-value sizes as shown in Figure 12(a). A small hotspot detector is enough to achieve significant performance improvement since most of the accesses are often concentrated on a few hotspots in real-world workloads [67]. To further evaluate the overhead of our hotspot detector, we compare its performance with an oracle hotspot detector under YCSB workloads. The oracle hotspot detector identifies the hotness of the key-value entry with near-zero overhead by getting its access probability from our workload generator. As shown in Figure 12(a),

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.

our hotspot detector achieves comparable performance to the oracle hotspot detector under YCSB workloads. The results demonstrate that our proposed hotspot detector is sufficient to support the adaptive update with low overhead.

Compacted-flush Insertion. Figure 12(b) shows that the compacted-flush approach accelerates the performance of Spash by up to 2.5× due to the full utilization of PM write bandwidth. Moreover, we actively flush the memory chunks in an XPLine-sized granularity, which achieves up to 2.2× performance improvement since it avoids write amplification induced by random cacheline eviction.

HTM-based Concurrency Protocol. Through two-phase concurrency protocol and collaborative staged doubling, Spash implements a lock-free concurrency control mechanism based on HTM, which significantly improves the scalability in multi-core systems. To evaluate the effect of HTM-based concurrency control, we implement two variants of Spash that employ the concurrency protocols of Dash and Level Hashing, respectively. Spash (w/ write lock) serializes concurrent write operations by a per-segment lock and retains the lock-free read design, which is similar to the concurrency protocol used in Dash. Spash (w/ write & read lock) adopts a per-segment lock to serialize both read and write operations, which is the concurrency protocol used in Level Hashing. As shown in Figure 12(c), Spash achieves up to 3.5× higher throughput than Spash (w/ write lock) and Spash (w/ write & read lock) in write-intensive workloads due to our lock-free write design. Moreover, Spash outperforms Spash (w/ write & read lock) by up to 4.4× in read-intensive workloads because of the lock-free read offered by HTM.

Pipeline Optimization. The pipeline executions improve the performance of Spash by fully utilizing high PM read bandwidth. As depicted in Figure 7, 10, and 11, pipeline optimization substantially improves the performance of Spash, particularly for read-intensive workloads. Meanwhile, we must choose a proper pipeline depth (PD) to avoid increasing operation latency. As shown in Figure 12(d), PD=4 is sufficient to utilize most PM read bandwidth and almost do not increase the operation latency.

VII. Related Work

Persistent Hash. In the past few years, many works have studied PM-based hash indexes. Level hashing [23] proposes a write-optimized persistent hash with a low overhead crash consistency guarantee. Level hashing suffers low scalability since it requires full-table rehashing and employs locks for both read and write operations. CLevel [47] further proposes lock-free concurrency control based on Level hashing. However, the performance of CLevel is still impeded by excessive PM reads and writes. CCEH [22] is a variant of extendible persistent hash, which introduces an intermediate segment level to reduce the directory size. CCEH exhibits low memory utilization due to the low association in the segment. Dash [7] proposes several load balancing techniques for CCEH, including balanced insert, displacement, and stashing, which improves the load factor of the hash table but sacrifices the performance. Plush [64], a write-optimized persistent hash, employs the LSM-Tree-based structure to enable sequential write, albeit at the cost of search performance. Nevertheless, it still produces a substantial volume of PM writes when merging different levels of persistent hash tables. Halo [65] stores the entire hash table in DRAM and employs a log-structured approach to manage key-value entries in PM, enhancing efficiency when traversing index structure. However, it produces notable PM writes for snapshot creations, as well as the creation, invalidation, and reclamation of log entries. The concurrent performance of Halo is also constrained by its lock-based protocol for both resizing and regular write operations. Although these works are designed for platforms with volatile CPU cache, they provide valuable inspiration for us.

The Exploration of Persistent CPU cache. With the emergence of eADR, researchers are looking into ways to optimize the existing PM systems with persistent CPU cache. Gugnani [43] proposes lock-free algorithms for the linked list and ring buffer based on atomic CPU hardware primitives. NBTree [16] is a lock-free persistent B^+-Tree designed for eADR-enabled platforms. HTMFS [19] employs HTM to simplify the concurrency control in persistent memory file systems. CacheKV [68] redesigns the *memtable* of LSM-Tree to optimize its write performance with the persistent CPU cache. Different from them, Spash fully exploits persistent CPU cache to optimize both write and concurrent performance of persistent hash indexes.

VIII. Conclusion

Persistent CPU cache technologies such as eADR offer new opportunities to optimize the write performance and concurrency control of PM-based storage systems. In this paper, we propose Spash, a highly scalable persistent hash index that fully leverages the benefits of persistent CPU cache. Spash employs a fine-grained extendible hash architecture, metadata-free segment, adaptive in-place update, and compacted-flush insertion to reduce PM access overhead. Moreover, Spash applies a two-phase concurrency protocol and collaborative staged doubling mechanism, which leverage HTM and persistent CPU cache to achieve high concurrency and durable linearizability. For YCSB workloads, Spash outperforms other indexes by up to 19.6×.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback and insightful suggestions. This work is supported by National Key Research and Development Program of China (Grant No. 2022YFB4500303), National Natural Science Foundation of China (NSFC) (Grant No. 62227809), the Fundamental Research Funds for the Central Universities, Shanghai Municipal Science and Technology Major Project (Grant No. 2021SHZDZX0102), Natural Science Foundation of Shanghai (Grant No. 22ZR1435400), and Huawei Innovation Research Plan.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] J. Handy, "Understanding the intel/micron 3d xpoint memory," *Proc. SDC*, 2015.

[2] R. M. Krishnan, W.-H. Kim, X. Fu, S. K. Monga, H. W. Lee, M. Jang, A. Mathew, and C. Min, "{TIPS}: Making volatile index structures persistent with {DRAM-NVMM} tiering," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 773–787.

[3] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu, "Scalable persistent memory file system with {Kernel-Userspace} collaboration," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 81–95.

[4] T. Ma, M. Zhang, K. Chen, Z. Song, Y. Wu, and X. Qian, "Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 757–773.

[5] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "{MatrixKV}: Reducing write stalls and write amplification in {LSM-tree} based {KV} stores with matrix container in {NVM}," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 17–31.

[6] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: Reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 494–508.

[7] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 8.

[8] J. Liu, S. Chen, and L. Wang, "Lb+ trees: Optimizing persistent index performance on 3dxpoint memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, 2020.

[9] Z. Dang, S. He, P. Hong, Z. Li, X. Zhang, X.-H. Sun, and G. Chen, "Nvalloc: rethinking heap metadata management in persistent memory allocators," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 115–127.

[10] Intel, "Deprecating the pcommit instruction," https://software.intel.com/content/www/us/en/develop/blogs/deprecate-pcommit-instruction.html, 2016.

[11] ——, "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.

[12] ——, "eadr: New opportunities for persistent memory applications," https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html, 2021.

[13] A. Rudoff, "Persistent memory programming," *Login: The Usenix Magazine*, vol. 42, no. 2, pp. 34–40, 2017.

[14] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *International Symposium on Distributed Computing*. Springer, 2016, pp. 313–327.

[15] X. Fu, D. Lee, and C. Min, "{DURINN}: Adversarial memory and thread interleaving for detecting durable linearizability bugs," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 195–211.

[16] B. Zhang, S. Zheng, Z. Qi, and L. Huang, "Nbtree: a lock-free pm-friendly persistent b-tree for eadr-enabled pm systems," *Proceedings of the VLDB Endowment*, vol. 15, no. 6, pp. 1187–1200, 2022.

[17] Z. Chen, Y. Hua, Y. Zhang, and L. Ding, "Efficiently detecting concurrency bugs in persistent memory programs," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 873–887.

[18] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on Computer architecture*, 1993, pp. 289–300.

[19] J. Yi, M. Dong, F. Wu, and H. Chen, "{HTMFS}: Strong consistency comes for free with hardware transactional memory in persistent memory file systems," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022, pp. 17–34.

[20] Z. Wang, H. Qian, J. Li, and H. Chen, "Using restricted transactional memory to build a scalable in-memory database," in *Proceedings of the Ninth European Conference on Computer Systems*, 2014, pp. 1–15.

[21] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner, "Improving in-memory database index performance with intel® transactional synchronization extensions," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 476–487.

[22] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "{Write-Optimized} dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 31–44.

[23] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 461–476.

[24] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, 2018, pp. 187–200.

[25] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, 2015, pp. 167–181.

[26] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, "utree: a persistent b+-tree with low tail latency," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2634–2648, 2020.

[27] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[28] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "Bztree: A high-performance latch-free range index for non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[29] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu, "{ROART}: Range-query optimized persistent {ART}," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 1–16.

[30] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, "Pactree: A high performance persistent range index using pac guidelines," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, 2021, pp. 424–439.

[31] S. Jung, H. Lee, S. Myung, H. Kim, S. K. Yoon, S.-W. Kwon, Y. Ju, M. Kim, W. Yi, S. Han *et al.*, "A crossbar array of magnetoresistive memory devices for in-memory computing," *Nature*, vol. 601, no. 7892, pp. 211–216, 2022.

[32] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal–oxide rram," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[33] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung *et al.*, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[34] G. Liu, L. Chen, and S. Chen, "Zen: a high-throughput log-free oltp engine for non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 835–848, 2021.

[35] C. Chen, J. Yang, M. Lu, T. Wang, Z. Zheng, Y. Chen, W. Dai, B. He, W.-F. Wong, G. Wu *et al.*, "Optimizing in-memory database engine for ai-powered on-line decision augmentation using persistent memory," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 799–812, 2021.

[36] B. Yan, X. Cheng, B. Jiang, S. Chen, C. Shang, J. Wang, G. Huang, X. Yang, W. Cao, and F. Li, "Revisiting the design of lsm-tree based oltp storage engine with persistent memory," *Proceedings of the VLDB Endowment*, vol. 14, no. 10, pp. 1872–1885, 2021.

[37] L. Benson, H. Makait, and T. Rabl, "Viper: an efficient hybrid pmem-dram key-value store," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1544–1556, 2021.

[38] J. Wang, Y. Lu, Q. Wang, M. Xie, K. Huang, and J. Shu, "Pacman: An efficient compaction approach for {Log-Structured}{Key-Value} store on persistent memory," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 773–788.

[39] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. 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*, 2020, pp. 1077–1091.

[40] S. Zheng, M. Hoseinzadeh, and S. Swanson, "Ziggurat: A tiered file system for {Non-Volatile} main memories and disks," in *17th USENIX*

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.

Conference on File and Storage Technologies (FAST 19), 2019, pp. 207–219.

[41] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and protection in the zofs user-space nvm file system," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 478–493.

[42] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 169–182.

[43] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the idiosyncrasies of real persistent memory," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 626–639, 2020.

[44] M. Friedman, N. Ben-David, Y. Wei, G. E. Blelloch, and E. Petrank, "Nvtraverse: In nvram data structures, the destination is more important than the journey," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 377–392.

[45] Intel, "Restricted transactional memory overview," https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2/intrinsics-for-tsx/intrinsics-for-restrict-transactional-mem-ops/restricted-transactional-memory-overview.html, 2021.

[46] ——, "Intel® transactional synchronization extensions (intel® tsx) programming considerations," https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2/intrinsics-for-tsx/tsx-programming-considerations.html, 2021.

[47] Z. Chen, Y. Huang, B. Ding, and P. Zuo, "Lock-free concurrent level hashing for persistent memory," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 799–812.

[48] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[49] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: Dependably fast multi-core in-memory transactions," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 21–35.

[50] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "Tictoc: Time traveling optimistic concurrency control," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1629–1642.

[51] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. 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*, 2012, pp. 53–64.

[52] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: Converting concurrent dram indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 462–477.

[53] J. Xu and S. Swanson, "{NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, 2016, pp. 323–338.

[54] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 133–146.

[55] T. Wang, J. Levandoski, and P.-A. Larson, "Easy lock-free indexing in non-volatile memory," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 461–472.

[56] Z. Liu, "Fujitsu targets 2019 for nram mass production." https://www.tomshardware.com/news/fujitsu-nram-nantero-carbon-nanotube,37437.html, 2018.

[57] X. U. Semiconductors, "Nvdimm products." http://www.unisemicon.com, 2022.

[58] M. Technology and H. SK, "Non volatile dual in line memory module nvdimm market research report." https://dataintelo.com/report/global-non-volatile-dual-in-line-memory-module-nvdimm-market/, 2021.

[59] M. Alshboul, P. Ramrakhyani, W. Wang, J. Tuck, and Y. Solihin, "Bbb: Simplifying persistent programming using battery-backed buffers," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 111–124.

[60] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[61] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE, 2014, pp. 216–223.

[62] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 421–432.

[63] C. Consortium, "Compute express link™: The breakthrough cpu-to-device interconnect." https://www.computeexpresslink.org, 2022.

[64] L. Vogel, A. Van Renen, S. Imamura, J. Giceva, T. Neumann, and A. Kemper, "Plush: A write-optimized persistent log-structured hash-table," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2895–2907, 2022.

[65] D. Hu, Z. Chen, W. Che, J. Sun, and H. Chen, "Halo: A hybrid pmem-dram persistent hash index with fast recovery," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1049–1063.

[66] Intel, "Ipmctl user guide," https://docs.pmem.io/ipmctl-user-guide/, 2022.

[67] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, "{HotRing}: A {Hotspot-Aware}{In-Memory}{Key-Value} store," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 239–252.

[68] Y. Zhong, Z. Shen, Z. Yu, and J. Shu, "Redesigning high-performance lsm-based key-value stores with persistent cpu caches," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 1098–1111.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on August 28,2024 at 05:57:24 UTC from IEEE Xplore. Restrictions apply.