Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory

Pengfei Zuo, Yu Hua, Jie Wu
Wuhan National Laboratory for Optoelectronics
School of Computer, Huazhong University of Science and Technology, China
Corresponding author: Yu Hua (csyhua@hust.edu.cn)

Abstract

Non-volatile memory (NVM) as persistent memory is expected to substitute or complement DRAM in memory hierarchy, due to the strengths of non-volatility, high density, and near-zero standby power. However, due to the requirement of data consistency and hardware limitations of NVM, traditional indexing techniques originally designed for DRAM become inefficient in persistent memory. To efficiently index the data in persistent memory, this paper proposes a write-optimized and high-performance hashing index scheme, called *level hashing*, with low-overhead consistency guarantee and cost-efficient resizing. Level hashing provides a sharing-based two-level hash table, which achieves a constant-scale search/insertion/deletion/update time complexity in the worst case and rarely incurs extra NVM writes. To guarantee the consistency with low overhead, level hashing leverages log-free consistency schemes for insertion, deletion, and resizing operations, and an opportunistic log-free scheme for update operation. To cost-efficiently resize this hash table, level hashing leverages an in-place resizing scheme that only needs to rehash 1/3 of buckets instead of the entire table, thus significantly reducing the number of rehashed buckets and improving the resizing performance. Experimental results demonstrate that level hashing achieves 1.4×−3.0× speedup for insertions, 1.2×−2.1× speedup for updates, and over 4.3× speedup for resizing, while maintaining high search and deletion performance, compared with state-of-the-art hashing schemes.

1 Introduction

As DRAM technology is facing significant challenges in density scaling and power leakage [44, 56], non-volatile memory (NVM) technologies, such as ReRAM [9], PCM [61], STT-RAM [10] and 3D XPoint [1], are promising candidates for building future memory systems. The non-volatility enables data to be persistently stored into NVM as *persistent memory* for instantaneous failure recovery. Due to byte-addressable benefit and the access latency close to DRAM, persistent memory can be directly accessed through the memory bus by using CPU load and store instructions, thus avoiding high overheads of conventional block-based interfaces [18, 39, 63, 64]. However, NVM typically suffers from the limited endurance and low write performance [50, 67].

The significant changes of memory architectures and characteristics result in the inefficiency of indexing data in the conventional manner that overlooks the requirement of data consistency and new NVM device properties [35, 46, 58, 64, 68]. A large amount of existing work has improved tree-based index structures for efficiently adapting to persistent memory, such as CDDS B-tree [58], NV-Tree [64], wB$^+$-Tree [17], FP-Tree [46], WORT [35], and FAST&FAIR [30]. Tree-based index structures are typically with the lookup time complexity of average O(log(N)) where N is the size of data structures [12, 19]. Unlike tree-based index structures, hashing-based index structures are flat data structures, which are able to achieve constant lookup time complexity, i.e., O(1), which is independent of N [42]. Due to providing fast lookup responses, hashing index structures are widely used in main memory systems. For example, they are fundamental components in main memory databases [27, 33, 38, 65], and used to index in-memory key-value stores [7, 8, 25, 36, 66], e.g., Redis and Memcached. However, when hashing index structures are maintained in persistent memory, multiple non-trivial challenges exist which are rarely touched by existing work.

1) High Overhead for Consistency Guarantee. Data structures in persistent memory should avoid any data inconsistency (i.e., data loss or partial updates) when system failures occur [28, 35, 46]. However, the new architecture that NVM is directly accessed through the memory bus causes high overhead to guarantee consistency. First, memory writes are usually reordered by CPU and memory controller [18, 20]. To ensure the

ordering of memory writes for consistency guarantee, we have to employ cache line flush and memory fence, introducing high performance overhead [17, 31, 45, 64]. Second, the atomic write unit for modern processors is generally no larger than the memory bus width (e.g., 8 bytes for 64-bit processors) [17, 20, 24, 60]. If the written data is larger than an atomic write unit, we need to employ expensive logging or copy-on-write (CoW) mechanisms to guarantee consistency [30, 35, 58, 64].

2) Performance Degradation for Reducing Writes. Memory writes in NVM consume the limited endurance and cause higher latency and energy than reads [50, 67]. Moreover, more writes in persistent memory also cause more cache line flushes and memory fences as well as possible logging or CoW operations, significantly decreasing the system performance. Hence, write reduction matters in NVM. Previous work [22, 68] demonstrates that common hashing schemes such as chained hashing, hopscotch hashing [29] and cuckoo hashing [47, 55] usually cause many extra memory writes for dealing with hash collisions. The write-friendly hashing schemes, such as PFHT [22] and path hashing [68], are proposed to reduce NVM writes in hashing index structures but at the cost of decreasing access performance (i.e., the throughput of search, insertion and deletion operations).

3) Cost Inefficiency for Resizing Hash Table. With the increase of the load factor (i.e., the ratio of the number of stored items to that of total storage units) of a hash table, the number of hash collisions increases, resulting in the decrease of access performance as well as insertion failure. Resizing is essential for a hash table to increase the size when its load factor reaches a threshold or an insertion failure occurs [26, 29, 48, 57]. Resizing a hash table needs to create a new hash table whose size is usually doubled, and then iteratively rehash all the items in the old hash table into the new one. Resizing is an expensive operation due to requiring O(N) time complexity to complete where N is the number of items in the hash table. Resizing also incurs N insertion operations, resulting in a large number of NVM writes with cache line flushes and memory fences in persistent memory.

To address these challenges, this paper proposes *level hashing*, a write-optimized and high-performance hashing index scheme with low-overhead consistency guarantee and cost-efficient resizing for persistent memory. Specifically, this paper makes the following contributions:

• ***Low-overhead Consistency Guarantee.*** We propose log-free consistency guarantee schemes for insertion, deletion, and resizing operations in level hashing. The three operations can be atomically executed for consistency guarantee by leveraging the token in each bucket whose size is no larger than an atomic write unit, without the need of expensive logging/CoW. Furthermore, for update operation, we propose an opportunistic log-free scheme to update an item without the need of logging/CoW in most cases. If the bucket storing the item to be updated has an empty slot, an item can be atomically updated without using logging/CoW.

• ***Write-optimized Hash Table Structure.*** We propose a sharing-based two-level hash table structure, in which a search/deletion/update operation only needs to probe at most four buckets to find the target key-value item, and hence has the constant-scale time complexity in the worst case with high performance. An insertion probes at most four buckets to find an empty location in most cases, and in rare cases only moves at most one item, with the constant-scale worst-case time complexity.

• ***Cost-efficient Resizing.*** To improve the resizing performance, we propose a cost-efficient in-place resizing scheme for level hashing, which rehashes only 1/3 of buckets in the hash table instead of the entire hash table, thus significantly reducing NVM writes and improving the resizing performance. Moreover, the in-place resizing scheme enables the resizing process to take place in a single hash table. Hence, search and deletion operations only need to probe one table during the resizing, improving the access performance.

• ***Real Implementation and Evaluation.*** We have implemented level hashing[1] and evaluated it in both real-world DRAM and simulated NVM platforms. Extensive experimental results show that the level hashing speeds up insertions by 1.4×−3.0×, updates by 1.2×−2.1×, and resizing by over 4.3× while maintaining high search and deletion performance, compared with start-of-the-art hashing schemes including BCH [25], PFHT [22] and path hashing [68]. The concurrent level hashing improves the request throughput by 1.6×−2.1×, compared with the start-of-the-art concurrent hashing scheme, i.e., libcuckoo [37].

The rest of this paper is organized as follows. Section 2 describes the background and motivation. Section 3 presents the design details. The performance evaluation is shown in Section 4. Section 5 discusses the related work and Section 6 concludes this paper.

2 Background and Motivation

In this section, we present the background of the data consistency issue in persistent memory and hashing index structures.

2.1 Data Consistency in NVM

In order to improve system reliability and efficiently handle possible system failures (e.g., power loss and

[1]The source code of level hashing is available at https://github.com/Pfzuo/Level-Hashing.

system crashes), the non-volatility property of NVM has been well explored and exploited to build persistent memory systems. However, since the persistent systems typically contain volatile storage components, e.g., CPU caches, we have to address the potential problem of data consistency that is interpreted as preventing data from being lost or partially updated in case of a system failure. To achieve data consistency in NVM, it is essential to ensure the ordering of memory writes to NVM [17, 35, 64]. However, the CPU and memory controller may reorder memory writes. We need to use the cache line flush instruction (`CLFLUSH` for short), e.g., *clflush*, *clflushopt* and *clwb*, and memory fence instruction (`MFENCE` for short), e.g., *mfence* and *sfence*, to ensure the ordering of memory writes, like existing state-of-the-art schemes [17, 35, 46, 58, 64]. The `CLFULSH` and `MFENCE` instructions are provided by the Intel x86 architecture [4]. Specifically, `CLFULSH` evicts a dirty cache line from caches and writes it back to NVM. `MFENCE` issues a memory fence, which blocks the memory access instructions after the fence, until those before the fence complete. Since only `MFENCE` can order `CLFLUSH`, `CLFLUSH` is used with `MFENCE` to ensure the ordering of `CLFLUSH` instructions [4]. However, the `CLFLUSH` and `MFEMCE` instructions cause significant system performance overhead [17, 20, 58]. Hence, it is more important to reduce writes in persistent memory.

It is well-recognized that the atomic memory write of NVM is 8 bytes, which is equal to the memory bus width for 64-bit CPUs [17, 35, 46, 58, 64]. If the size of the updated data is larger than 8 bytes and a system failure occurs before completing the update, the data will be corrupted. Existing techniques, such as logging and copy-on-write (CoW), are used to guarantee consistency of the data whose sizes are larger than an atomic-write size. The logging technique first stores the old data (undo logging) or new data (redo logging) into a log and then updates the old data in place. The CoW first creates a new copy of data and then performs updates on the copy. The pointers that point to the old data are finally modified. Nevertheless, logging and CoW have to write twice for each updated data. The ordering of the two-time writes also needs to be ensured using `CLFLUSH` and `MFENCE`, significantly hurting the system performance.

2.2 Hashing Index Structures for NVM

2.2.1 Conventional Hashing Schemes

Hashing index structures are widely used in current main memory databases [23, 27, 33, 38, 65], and key-value stores [7, 8, 25, 36, 51], to provide fast query responses. Hash collisions, i.e., two or more keys are hashed into the same bucket, are practically unavoidable in hashing index structures. *Chained hashing* [32] is a popular scheme to deal with hash collisions, which stores the conflicting items in a linked list via pointers. However, the chained hashing consumes extra memory space due to maintaining the pointers, and decreases access performance when the linked lists are too long.

Open addressing is another kind of hashing scheme to deal with hash collisions without pointers, in which each item has a fixed probe sequence. The item must be in one bucket of its probe sequence. *Bucketized cuckoo hashing (BCH)* [13, 25, 37] is a memory-efficient open-addressing scheme, which has been widely used due to the constant lookup time complexity in the worst case and memory efficiency (i.e., achieving a high load factor). BCH uses f ($f \geq 2$) hash functions to compute f bucket locations for each item. Each bucket includes multiple slots. An inserted item can be stored in any empty slot in its corresponding f buckets. If all slots in the f buckets are occupied, BCH randomly evicts an item in one slot. The evicted item further iteratively evicts other existing items until finding an empty location. For a search operation, BCH probes at most f buckets and hence has a constant search time complexity in the worst case. Due to sufficient flexibility with only two hash functions, $f = 2$ is actually used in BCH [13, 22, 25, 37]. Hence, the BCH in our paper uses two hash functions.

2.2.2 Hashing Schemes for NVM

The mentioned hashing schemes above mainly consider the properties of the traditional memory devices, such as DRAM and SRAM. Unlike them, the new persistent memory systems are tightly related with the significant changes of memory architectures and characteristics, which bring the non-trivial challenges to hashing index structures. For example, NVM typically has limited endurance and incurs higher write latency than DRAM [50, 67]. The chained hashing results in extra NVM writes due to the modifications of pointers and BCH causes *cascading NVM writes* due to frequently evicting and rewriting items for insertion operations, which exacerbate the endurance of NVM and the insertion performance of hash tables [22, 68]. More importantly, the traditional hashing schemes do not consider data consistency and hence cannot directly work on persistent memory.

Hashing schemes [22, 68] have been improved to efficiently adapt to NVM, which mainly focus on reducing NVM writes in hash tables. Debnath et al. [22] propose a *PCM-friendly Hash Table (PFHT)* which is a variant of BCH for reducing writes to PCM. PFHT modifies the BCH to only allow one-time eviction when inserting a new item, which can reduce the NVM writes from frequent evictions but results in low load factor. In order to improve the load factor, PFHT further uses a stash to store the items failing to be inserted into the

Table 1: Comparisons among level hashing and state-of-the-art memory-efficient hashing schemes. *(In this table, "×" indicates a bad performance, "√" indicates a good performance and "−" indicates a moderate performance in the corresponding metrics.)*

	BCH	PFHT	Path hashing	Level hashing
Memory Efficiency	√	√	√	√
Search	√	−	−	√
Deletion	√	−	−	√
Insertion	×	−	−	√
NVM Writes	×	√	√	√
Resizing	×	×	×	√
Consistency	×	×	×	√

hash table. However, PFHT needs to linearly search the stash when failing to find an item in the hash table, thus increasing the search latency. Our previous work [68, 69] proposes the *path hashing* that supports insertion and deletion operations without any extra NVM writes. Path hashing logically organizes the buckets in the hash table as an inverted complete binary tree. Each bucket stores one item. Only the leaf nodes are addressable by hash functions. When hash collisions occur in the leaf node of a path, all non-leaf nodes in the same path are used to store the conflicting key-value items. Thus insertions and deletions in the path hashing only need to probe the nodes within two paths for finding an empty bucket or the target item, without extra writes. However, path hashing offers a low search performance due to the need of traversing two paths until finding the target item for each search operation.

Table 1 shows a high-level comprehensive comparison among these state-of-the-art memory-efficient hashing schemes including BCH, PFHT and path hashing. In summary, BCH is inefficient for insertion due to frequent data evictions. PFHT and path hashing reduce NVM writes in the insertion and deletion operations but at the cost of decreasing access performance. More importantly, these hashing schemes overlook the data consistency issue of hash tables in NVM as well as the efficiency of the resizing operation that often causes a large number of NVM writes. Our paper proposes the level hashing that achieves good performance in terms of all these metrics as shown in Section 3, which is also verified in the performance evaluation as shown in Section 4.

2.2.3 Resizing a Hash Table

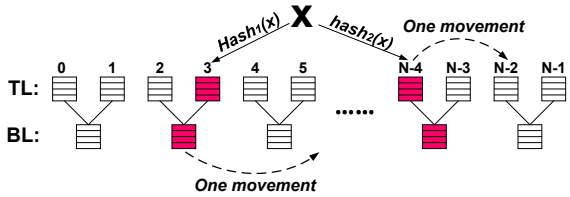
With the increase of the load factor of a hash table, the number of hash collisions increases, resulting in the decrease of the access performance as well as insertion failure [48, 57]. Once a new item fails to be inserted into a hash table, this hash table has to be resized by growing its size. Traditional resizing schemes [40, 48, 53] perform out-of-place resizing, in which expanding a hash table needs to create a new hash table whose size is larger than that of the old one, and then iteratively rehash all items from the old hash table to the new one.

The size of the new hash table is usually double size of the old one [40, 53, 54, 57], due to two main reasons. First, the initial size of a hash table is usually set to be a power of 2, since it allows very cheap modulo operations. For a hash table with power-of-2 (i.e., 2^n) buckets, computing the location of a key based on its hash value, i.e., `hash(key)%2^n`, is a simple bit shift, which is much faster than computing an integral division, e.g., `hash(key)%(2^n-1)`. Thus, if doubling the size in resizing a hash table, the size of the new hash table is still a power of 2. Second, the access performance of a hash table depends on the size of the hash table [57]. If resizing the hash table to a too small size, the new hash table may result in high hash collision rate and poor insertion performance, which will quickly incur another resizing operation. If resizing the hash table to a too large size for inserting a few new items, the new hash table consumes too much memory, reducing the memory space available for other applications. In general, doubling the size when resizing a hash table has been widely recognized [53, 54, 57]. For example, in the real-world applications, such as Java HashMap [5] and Memcached [7], doubling the size is the default setting for resizing a hash table.

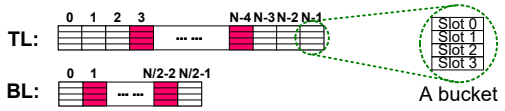
When the stored items are far fewer than the storage units in a hash table, the hash table also needs to be resized via shrinking its size. Resizing is an expensive operation that consumes O(N) time to complete, where N is the number of buckets in the old hash table. Moreover, during the resizing, each search or deletion operation needs to check both old and new hash tables, decreasing the access performance. For hashing index structures maintained in persistent memory, resizing causes a large number of NVM writes with cache line flushes and memory fences, significantly hurting the NVM endurance and decreasing the resizing performance.

3 The Level Hashing Design

We propose *level hashing*, a write-optimized and high-performance hashing index scheme with cost-efficient resizing and low-overhead consistency guarantee for persistent memory. In this section, we first present the basic data structure of level hashing (§3.1), i.e., level hash table, which aims to achieve the high performance as well as high load factor, and rarely incurs extra writes. We then present a cost-efficient in-place resizing scheme (§3.2) for level hashing to reduce NVM writes and improve the resizing performance. We then present the (opportunistic) log-free schemes (§3.3) to reduce the consistency overhead. We finally present the concurrent level hashing leveraging fine-grained locking (§3.4).



(a) The logical structure



(b) The physical storage structure

Figure 1: The hash table structure of level hashing with 4 slots per bucket. *(In these tables, "TL" indicates the top level, and "BL" indicates the bottom level.)*

3.1 Write-optimized Hash Table Structure

A level hash table is a new open-addressing structure with all the strengths of BCH, PFHT and path hashing, including memory-efficient, write-optimized, and high performance, while avoiding their weaknesses, via performing the following major design decisions.

*D_1: **Multiple Slots per Bucket.*** According to multiple key-value workload characteristics published by Facebook [11] and Baidu [34], small key-value items whose sizes are smaller than a cache-line size dominate in current key-value stores. For example, the size of most keys is smaller than 32 bytes, and 16 or 21-byte key with 2-byte value is a common request type in Facebook's key-value store [11]. Motivated by the real-world workload characteristics, we enable the level hash table to be cache-efficient by setting multiple slots in each bucket, e.g., 4 slots per bucket as shown in Figure 1. Thus a bucket can store multiple key-value items each in one slot. When accessing a bucket in the level hash table, multiple key-value items in the same bucket can be prefetched into CPU caches in one memory access, which improves the cache efficiency and thus reduces the number of memory accesses.

*D_2: **Two Hash Locations for Each Key.*** Since each bucket has k slots, the hash table can deal with at most $k-1$ hash collisions occurring in a single hash position. It is possible that more than k key-value items are hashed into the same position. In this case, insertion failure easily occurs, resulting in a low load factor. To address this problem, we enable each key to have two hash locations via using two different hash functions, i.e., $\texttt{hash}_1()$ and $\texttt{hash}_2()$, like BCH [13, 25, 37], PCHT [22] and path hashing [68, 69]. A new key-value item is inserted into the less-loaded bucket between the two hash locations [14]. Due to the randomization of two independent hash functions, the load factor of hash table is significantly improved as shown in Figure 2.

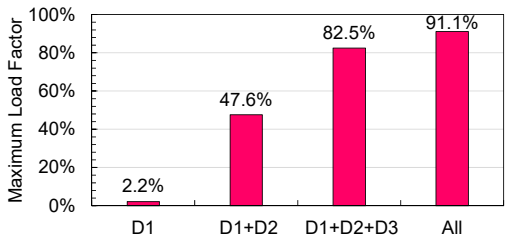


Figure 2: The maximum load factors when adding different design decisions. *(D_1: a one-level hash table with 4 slots per bucket; D_1+D_2: a hash table with design decisions D_1 and D_2; $D_1+D_2+D_3$: a hash table with D_1, D_2 and D_3; All: level hash table that uses $D_1+D_2+D_3+D_4$.)*

*D_3: **Sharing-based Two-level Structure.*** The buckets in the level hash table are divided into two levels, i,e., a top level and a bottom level, as shown in Figure 1a. Only the buckets in the top level are addressable by hash functions. The bottom level is not addressable and used to provide standby positions for the top level to store conflicting key-value items. Each bottom-level bucket is shared by two top-level buckets, and thus the size of the bottom level is half of the top level. If a hash collision occurs in a top-level bucket and all positions in the bucket are occupied, the conflicting key-value item can be stored in its corresponding standby bucket in the bottom level. By using the two-level structure, the load factor of hash table is significantly improved as shown in Figure 2. Moreover, since each addressable bucket has one standby bucket, a search operation only needs to probe at most four buckets, having the constant-scale time complexity in the worst case.

*D_4: **At Most One Movement for Each Successful Insertion.*** To enable key-value items to be evenly distributed among buckets, if both buckets are full during inserting an item in the BCH [13, 22, 25, 37], BCH iteratively evicts one of existing items and thus incurs cascading writes, which is not friendly for NVMs. To avoid the problem of the cascading writes, instead, level hashing allows the movement of at most one item for each insertion. Specifically, during inserting a new item (I_{new}), if the two top-level buckets are full, we check whether it is possible to move any key-value item from one of its two top-level buckets to its alternative top-level location. If no movement is possible, we further insert the new item I_{new} into the bottom level. If the two bottom-level buckets are full, we also check whether it is possible to move any key-value item from one of its two bottom-level buckets to its alternative bottom-level location. If the movement still fails, the insertion fails and the hash table needs to be resized. Note that the movement is saved if the alternative location of the moved item has no empty slot. Allowing one movement

redistributes the items among buckets, thus improving the maximum load factor, as shown in Figure 2.

Put them all together, the hash table structure of level hashing is shown in Figure 1. Figure 1a shows the logical structure of a level hash table that contains two-level buckets. The links between two levels indicate the bucket sharing relationships, instead of pointers. Figure 1b shows the physical storage of a level hash table, in which each level is stored in a one-dimensional array. For a key-value item with the key K, its corresponding two buckets in the top level (i.e., the No.L_{t1} and No.L_{t2} buckets) and its two standby buckets in the bottom level (i.e., the No.L_{b1} and No.L_{b2} buckets) can be obtained via the following equations:

$$L_{t1} = \texttt{hash}_1\texttt{(K)\%N}, L_{t2} = \texttt{hash}_2\texttt{(K)\%N} \quad (1)$$

$$L_{b1} = \texttt{hash}_1\texttt{(K)\%(N/2)}, L_{b2} = \texttt{hash}_2\texttt{(K)\%(N/2)} \quad (2)$$

The computations of Equations 1 and 2 only require the simple bit shift operation since N is a power of 2. The simple yet efficient hash table structure shown in Figure 1 has the following strengths:

• *Write-optimized.* Level hashing does not cause the cascading writes via allowing at most one movement for each insertion. Moreover, only a very small number of insertions incur one movement. Based on our experiments, when continuously inserting key-value items into a level hash table until reaching its maximum load factor, only 1.2% of insertions incur one movement.

• *High-performance.* For a search/deletion/update operation, level hashing probes at most four buckets to find the target item. For an insertion operation, level hashing probes at most four buckets to find an empty location in most cases, and in rare cases further moves at most one existing item. Hence, level hashing achieves the constant-scale worst-case time complexity for all operations.

• *Memory-efficient.* In the level hash table, two hash locations for each key enables the key-value items in the top level to be evenly distributed [43]. Each un-addressable bucket is shared by two addressable buckets to store the conflicting items, which enables the items in the bottom level to be evenly distributed. Allowing one movement enables items to be evenly redistributed. These design decisions enable the level hash table to be load-balanced and memory-efficient, thus achieving more than 90% load factor as shown in Figure 2.

Moreover, the level hashing has a good resizing performance via a cost-efficient in-place resizing scheme as shown in Section 3.2. We guarantees the data consistency in the level hashing with low overhead via the (opportunistic) log-free schemes as shown in Section 3.3.

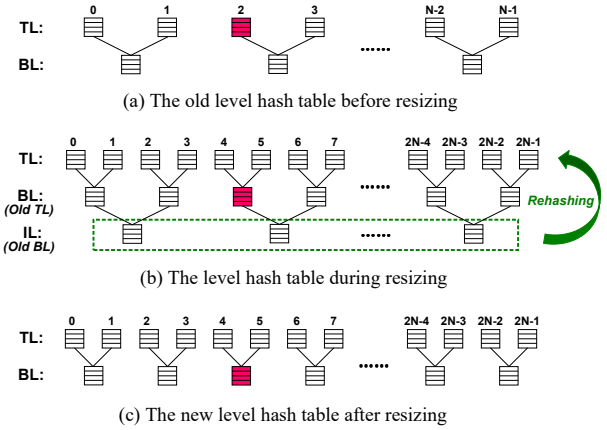


Figure 3: The cost-efficient in-place resizing in the level hashing. *("IL" indicates the interim level.)*

3.2 Cost-efficient In-place Resizing

To reduce NVM writes and improve the resizing performance, we propose *a cost-efficient in-place resizing scheme*. The basic idea of the in-place resizing scheme is to put a new level on the top of the old hash table and only rehash the items in the bottom level of the old hash table when expanding a level hash table.

1) An Overview of Resizing. A high-level overview of the in-place resizing process in the level hashing is shown in Figure 3. Before the resizing, the level hash table is a two-level structure, including a top level (TL) with N buckets and a bottom level (BL) with $N/2$ buckets, as shown in Figure 3a. During the resizing, we first allocate the memory space with $2N$ buckets as the new top level and put it on the top of the old hash table. The level hash table becomes a three-level structure during the resizing, as shown in Figure 3b. The third level is called the interim level (IL). The in-place resizing scheme rehashes the items in the IL into the top-two levels. Each rehashing operation includes reading an item in the IL, inserting the item into the top-two levels and deleting the item from the IL. After all items in the IL are rehashed into the top-two levels, the memory space of the IL is reclaimed. After the resizing, the new hash table becomes a two-level structure again, as shown in Figure 3c. The rehashing failure (which indicates a rehashed item fails to be inserted into the top-two levels) does not occur when the resizing is underway, since currently the total number of stored items is smaller than half of the total size of the new level hash table, and level hashing is able to achieve the load factor of higher than 0.9 (> 0.5) as evaluated in Section 4.2.1.

We observe that the new hash table with $3N$ buckets is exactly double size of the old hash table with $1.5N$ buckets, which meets the demand of real-world applications as discussed in Section 2.2.3. Unlike the

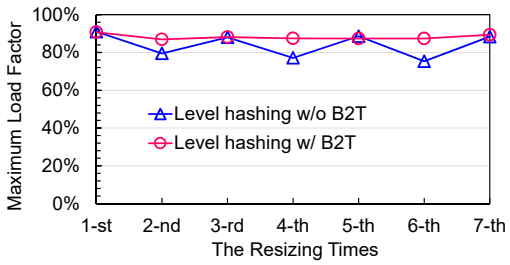


Figure 4: The load factors when the resizings occur.

traditional out-of-place resizing scheme [48] in which the resizing occurs between the old and new tables, the in-place resizing enables the whole resizing process to occur in a single hash table. Thus during resizing, search and deletion operations only need to probe one table and compute the hash functions once, thus improving the access performance. Moreover, the in-place resizing rehashes only the bottom level of the old hash table instead of the entire table. The bottom level only contains $1/3(=0.5N/1.5N)$ of all buckets in the old hash table, thus significantly reducing data movements and NVM writes during the resizing, as well as improving the resizing performance.

We can also shrink the level hash table in place which is an inverse process of expanding the level hash table. Specifically, to shrink the level hash table, we first allocate the memory space with $N/4$ buckets as the new bottom level which is placed on the bottom of the old hash table. We then rehash all items in the old top level into the bottom-two levels.

2) Improving the Maximum Load Factor after Resizing. In the level hash table, each item is stored in the bottom level only when its corresponding two top-level buckets are full. Thus before resizing, the top-level buckets are mostly full and the bottom-level buckets are mostly non-full. After resizing, the top level in the old hash table becomes the bottom level in the new hash table as shown in Figure 3. Thus the bottom-level buckets in the new hash table are mostly full, which easily incur an insertion failure, reducing the maximum load factor. The blue line in Figure 4 shows the load factors of the level hash table when the multiple successive resizings occur. We observe that the maximum load factors in the 2-nd, 4-th, and 6-th resizings are reduced, compared with those in the 1-st, 3-rd and 5-th resizings. The reason is that the bottom-level buckets are mostly full in the 2-nd, 4-th and 6-th resizings.

To address this problem, we propose *a bottom-to-top movement (B2T) scheme* for level hashing. Specifically, during inserting an item, if its corresponding two top-level buckets (L_{t1} and L_{t2}) and two bottom-level buckets (L_{b1} and L_{b2}) are full, the B2T scheme tries to move one existing item (I_{ext}) in the bottom-level bucket L_{b1} or L_{b2} into the top-level alternative locations of I_{ext}. Only when the corresponding two top-level buckets of I_{ext} have no empty slot, the insertion is considered as a failure and incurs a hash table resizing. By performing the B2T scheme, the items between top and bottom levels are redistributed, thus improving the maximum load factor. The red line in Figure 4 shows the load factors when the resizings occur via using the B2T scheme. We observe that the maximum load factors in the 2-nd, 4-th and 6-th resizings are improved.

3) Improving the Search Performance after Resizing. After resizing, the search performance possibly decreases. This is because in the original search scheme (called *static search*) as shown in Section 3.1, we always first probe the top level, and if not finding the target item, we then probe the bottom level. Before resizing, about $2/3$ items are in the top level. However, the $2/3$ items are in the bottom level after resizing, since the top level in the old hash table becomes the bottom level in the new one as shown in Figure 3. Hence, a single search needs to probe two levels in most cases (i.e., about $2/3$ probability) after resizing, thus degrading the search performance.

To address this problem, we propose *a dynamic search scheme* for level hashing. Specifically, for a search, we study two cases based on the numbers of items in the top and bottom levels. First, if the items in the bottom level are more than those in the top level, we first probe the bottom level (based on Equation 2), and if not finding the target item, we then probe the top level (based on Equation 1). Second, if the items in the bottom level are less than those in the top level, we first probe the top level and then the bottom level. Thus after resizing, the items in the bottom level are more than those in the top level and hence we first probe the bottom level, thus improving the search performance. We also demonstrate the performance improvement in Section 4.2.4.

3.3 Low-overhead Consistency Guarantee

In the open-addressing hash tables, a token associated with each slot is used to indicate whether the slot is empty [25, 68]. As shown in Figure 5, in a bucket, the header area stores the tokens of all slots and the remaining area stores the slots each with a key-value item. A token is defined as a 1-bit flag that indicates whether the corresponding slot is empty. For example, the token '0' indicates the corresponding slot is empty and the token '1' indicates the slot is non-empty. The header area is 1 byte when the number of slots is not larger than 8, and 2 bytes for the buckets with 16 slots. Since the header area is always smaller than 8 bytes, modifying the tokens only needs to perform an atomic write. But the key-value items are usually larger than 8 bytes. A straightforward approach is to guarantee the consistency of writing key-value items via logging

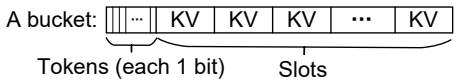


Figure 5: The storage structure of each bucket.

or CoW, which however incurs significant performance overhead as discussed in Section 2.1.

To reduce the overhead of guaranteeing consistency in level hashing, we propose *log-free consistency guarantee schemes* for deletion, insertion, and resizing operations, and *an opportunistic log-free guarantee scheme* for update operation, by leveraging the tokens to be performed in the atomic-write manner.

1) Log-free Deletion. When deleting a key-value item from a slot, we change the token of the slot from '1' to '0', which invalidates the deleted key-value item. The deletion operation only needs to perform an atomic write to change the token. After the token of the slot is changed to '0', the slot becomes available and can be used to insert a new item.

2) Log-free Insertion. There are two cases when inserting a new item into the level hash table.

a) No item movement: The insertion incurs no movement, i.e., inserting a new item to an empty slot. In this case, we first write the new item into the slot and then change its token from '0' to '1'. The ordering of writing the item and changing the token is ensured via an `MFENCE`. Although the new item is larger than 8 bytes, writing the item does not require logging or CoW, since the item becomes valid until the token is set to '1'. If a system failure occurs during writing the item, this item may be partially written but invalid since the current token is '0' and this slot is still available. Hence, the hash table is in a consistent state when system failures occur.

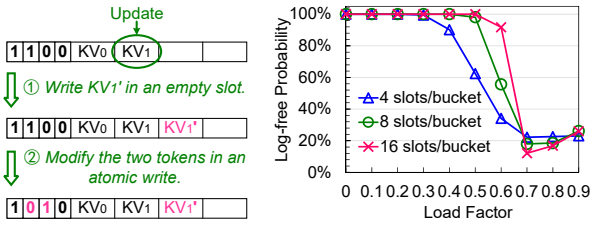
b) Moving one item: The insertion incurs the movement of one item. In this case, we need to take two steps to insert an item, and the ordering of executing the two steps is ensured via an `MFENCE`. The first step is to move an existing item into its alternative bucket. We use $\texttt{slot}_{\texttt{cur}}$ to indicate the current slot of the existing item and use $\texttt{slot}_{\texttt{alt}}$ to indicate its new slot in the alternative bucket. Moving this item first copies the item into $\texttt{slot}_{\texttt{alt}}$, then modifies the token of $\texttt{slot}_{\texttt{alt}}$ from '0' to '1' and finally modifies the token of $\texttt{slot}_{\texttt{cur}}$ from '1' to '0'. If a system failure occurs after changing the token of $\texttt{slot}_{\texttt{alt}}$ before changing the token of $\texttt{slot}_{\texttt{cur}}$, the hash table contains two duplicate key-value items, which however does not impact on the data consistency. It is because when searching this key-value item, the returned value is always correct whichever one of the two items is queried. When updating this item, one of the two items is first deleted and the other one is then updated, as presented in Section 3.3(4). After moving this existing item, the second step inserts the new item into the empty slot using the method of "*a) no item movement*".

3) Log-free Resizing. During resizing, we need to rehash all key-value items in the interim level. For a rehashed item, we use $\texttt{slot}_{\texttt{old}}$ to indicate its old slot in the interim level and use $\texttt{slot}_{\texttt{new}}$ to indicate its new slot in the top-two levels. Rehashing an item in the interim level can be decomposed into two steps, i.e., inserting the item into $\texttt{slot}_{\texttt{new}}$ (*Log-free Insertion*) and then deleting the item from $\texttt{slot}_{\texttt{old}}$ (*Log-free Deletion*). To guarantee the data consistency during a rehashing operation, we first copy the key-value item of $\texttt{slot}_{\texttt{old}}$ into $\texttt{slot}_{\texttt{new}}$, and then modifies the token of $\texttt{slot}_{\texttt{new}}$ from '0' to '1' and finally modifies the token of $\texttt{slot}_{\texttt{old}}$ from '1' to '0'. The ordering of the three steps is ensured via `MFENCE`s. If a system failure occurs when copying the item, the hash table is in a consistent state since the $\texttt{slot}_{\texttt{new}}$ is still available and the item in $\texttt{slot}_{\texttt{old}}$ is not deleted. If a system failure occurs after changing the token of $\texttt{slot}_{\texttt{new}}$ before changing the token of $\texttt{slot}_{\texttt{old}}$, $\texttt{slot}_{\texttt{new}}$ is inserted successfully but the item in $\texttt{slot}_{\texttt{old}}$ is not deleted. There are two duplicate items in the hash table, which however has no impact on the data consistency, since we can easily remove one of the two duplicates after the system is recovered without scanning the whole hash table. In case of a system failure, only the first item (I_{first}) to be rehashed in the interim level may be inconsistent. To check whether there are two duplicates of I_{first} in the hash table, we only need to query the key of I_{first} in the top-two levels. If two duplicates exist, we directly delete I_{first}. Otherwise, we rehash it. Therefore, the hash table can be recovered in a consistent state.

4) Opportunistic Log-free Update. When updating an existing key-value item, if the updated item has two copies in the hash table, we first delete one and then update the other. If we directly update the key-value item in place, the hash table may be left in the corrupted state when a system failure occurs, since the old item is overwritten and lost, and the new item is not written completely. Intuitively, we address this problem via first writing the new or old item into a log and then updating the old item in place, which however incur high performance overhead.

To reduce the overhead, we leverage *an opportunistic log-free update scheme* to guarantee consistency. Specifically, for an update operation (e.g., updating KV_1 to KV_1'), we first check whether there is an empty slot in the bucket storing the old item (KV_1).

- **Yes.** If an empty slot exists in the bucket as shown in Figure 6a, we directly write the new item (KV_1') into the empty slot, and then modify the tokens of the old item (KV_1) and new item (KV_1')



(a) Log-free update (b) The log-free probability

Figure 6: The opportunistic log-free update scheme. *((a) The log-free update scheme; (b) The probability of performing log-free update with the increase of load factor and the change of the number of slots/bucket.)*

simultaneously. The two tokens are stored together and hence can be simultaneously modified in an atomic write. The ordering of writing the new item and modifying the tokens is ensured by an `MFENCE`.

- **No.** If no empty bucket exists in the bucket storing the old item (KV_1), we first log the old item and then update the old item in place. If a system failure occurs during overwriting the old item, the old item can be recovered based on the log.

In summary, if there is an empty slot in the bucket storing the item to be updated, we update the item without logging. We evaluate the opportunity to perform log-free update, i.e., the probability that the bucket storing the updated item contains at least one empty slot, as shown in Figure 6b. The probability is related with the number of slots in each bucket and the load factor of hash table. We observe that when the load factor of hash table is smaller than about $2/3$, the probability of log-free update is very high and decreases with the increase of the load factor and the decrease of the number of slots in each bucket. However, when the load factor is larger than $2/3$, the probability increases with the increase of the load factor. This is because the number of storage units in the top level is $2/3$ of the total storage units. When the load factor is beyond $2/3$, more items are inserted into the bottom level, and the buckets in the bottom level have the higher probability to contain an empty slot than those in the top level.

We further discuss whether the proposed consistency-guarantee schemes work on other hashing schemes. 1) The proposed log-free deletion scheme can be used in other open-addressing hashing schemes, since deletion only operates on a single item. 2) The opportunistic log-free update scheme can be used in other multiple-slot hashing schemes, e.g, BCH, and PFHT. 3) Obviously, the log-free insertion scheme can be used in the hashing schemes without data evictions during insertions, e.g., path hashing, and the hashing schemes with at most one eviction, e.g., PFHT. In fact, the log-free insertion

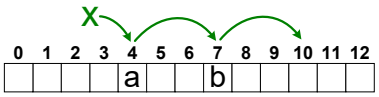


Figure 7: An insertion in the cuckoo hashing.

scheme can also be used in the hashing schemes with iterative eviction operations during insertions, e.g., cuckoo hashing. Specifically, an insertion in cuckoo hashing may iteratively evict key-value items until finding an empty location. The sequence of evicted items is called a cuckoo path [37]. To perform log-free insertion, we first search for a cuckoo path with an empty location but do not execute evictions during search. We then perform evictions starting with the last item in the cuckoo path and working backward toward the first item. For example, as shown in Figure 7, the new item x is inserted into the location $L4$, and the sequence of $x \rightarrow a \rightarrow b \rightarrow \varnothing$ is a cuckoo path. To perform log-free insertion, we first move b from $L7$ to $L10$, and then move a from $L4$ to $L7$, and finally insert x into $L4$.

3.4 Concurrent Level Hashing

As current systems are being scaled to larger number of cores and threads, concurrent data structures become increasingly important [15, 25, 37, 41]. The level hash table does not use pointers and has no cascading writes, which enables level hashing to efficiently support multi-reader and multi-writer concurrency via simply using fine-grained locking.

In the concurrent level hashing, the conflicts occur when different threads concurrently read/write the same slot. Hence, we allocate a fine-grained locking for each slot. When reading/writing a slot, the thread first locks it. Since level hashing allows each insertion to move at most one existing item, an insertion operation locks at most two slots, i.e., the current slot and the target slot that the item will be moved into. Nevertheless, the probability that an insertion incurs a movement is very low as presented in Section 3.1. An insertion locks only one slot in the most cases, and hence the concurrent level hashing delivers high performance as evaluated in Section 4.2.7.

4 Performance Evaluation

4.1 Experimental Setup

All our experiments are performed on a Linux server (kernel version 3.10.0) that has four 6-core Intel Xeon E5-2620 2.0GHz CPUs (each core with 32KB L1 instruction cache, 32KB L1 data cache, and 256KB L2 cache), 15MB last level cache and 32GB DRAM.

Since the real NVM device is not available for us yet, we conduct our experiments using Hewlett Packard's Quartz [2, 59], which is a DRAM-based performance

emulator for persistent memory and has been widely used [31, 35, 39, 52, 60]. Quartz emulates the latency of persistent memory by injecting software created delays per epoch and limiting the DRAM bandwidth by leveraging DRAM thermal control registers. However, the current implementation of Quartz [2] does not yet support the emulation of write latency in the persistent memory. We hence emulate the write latency by adding an extra delay after each `CLFLUSH` instruction, following the methods in existing work [31, 35, 39, 52, 60].

The evaluation results in PFHT [22] and path hashing [68] demonstrated that PFHT and path hashing significantly outperform other existing hashing schemes, including chained hashing, linear probing [49], hopscotch hashing [29] and cuckoo hashing [47, 55], in NVM. Therefore, we compare our proposed level hashing with the state-of-the-art NVM-friendly schemes, i.e., PFHT and path hashing, and the memory-efficient hashing scheme for DRAM, i.e., BCH, in both DRAM and NVM platforms. Since these hashing schemes do not consider the data consistency issue on persistent memory, we implement persistent BCH, PFHT, and path hashing using our proposed consistency guarantee schemes as discussed in Section 3.3 for fairly comparing their performance on persistent memory. Moreover, we also compare the performance of these hashing schemes without crash consistency guarantee in DRAM.

Since 16-byte key has been widely used in current key-value stores [11, 34, 62], we use the 16-byte key, the value that is no longer than 15 bytes, and 1-bit token for each slot. Two slots align a cache line (64B) via padding several unused bytes. Every hash table is sized for 100 million key-value items and thus needs about 3.2GB memory space. Besides examining the single-thread performance of each kind of operation, we also use YCSB [21], a benchmark for key-value stores, to evaluate the concurrent performance of the concurrent level hashing in multiple mixed workloads. In the experimental results, each data value is the average of 10-run results.

4.2 Experimental Results

4.2.1 Maximum Load Factor

The maximum load factor is an important metric for hash table due to directly affecting the number of key-value items that a hash table can store and the hardware cost [25, 37]. For evaluating the maximum load factor, we insert unique string keys into empty BCH, PFHT, level and path hash tables until an insertion failure occurs. Specifically, BCH reaches the maximum load factor when a single insertion operation fails to find an empty slot after 500 evictions [25, 37]. For PFHT, the 3% space of the total hash table size is used as a stash,

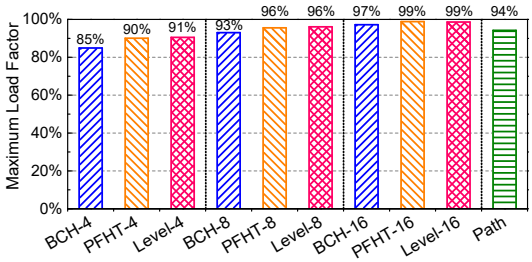


Figure 8: Maximum load factors of hash tables. *(# in the NAME-# indicates the number of slots per bucket.)*

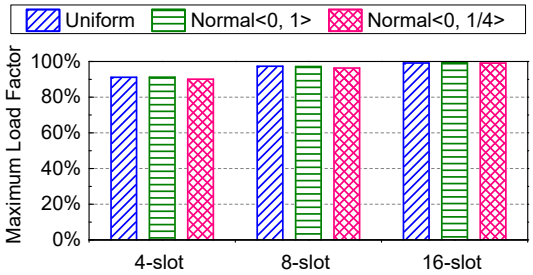
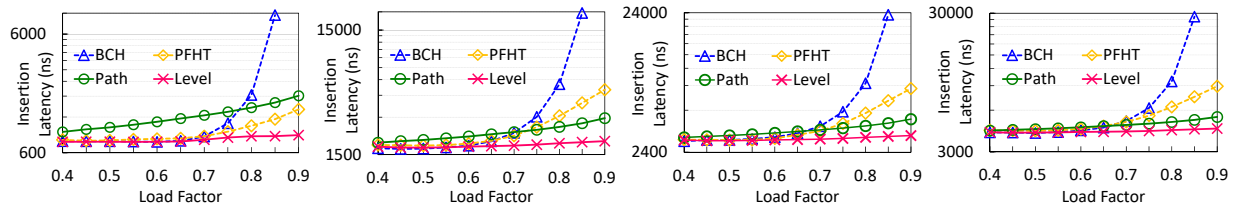


Figure 9: Maximum load factors of the level hash table with different-distribution integer keys. *(Normal$< x, y >$ indicates the logarithmic normal distribution with the parameters $\mu = x$ and $\sigma = y$.)*

following the configuration in the original paper [22]. PFHT reaches the maximum load factor when the stash is full. Level and path hash tables reach the maximum load factors when a single insertion fails to find an empty slot or bucket.

Figure 8 shows that all the four hash tables can achieve over 90% of maximum load factor. Figure 8 also compares different hash tables with the different numbers of slots in each bucket. More slots in each bucket incur higher maximum load factor for BCH, PFHT and level hash table. For the same number of slots in each bucket, PFHT and level hash table have approximately the same maximum load factor, which are higher than BCH. Path hash table is a one-item-per-bucket table and achieves up to 94.2% maximum load factor.

We also evaluate the maximum load factors of the level hash table with different-distribution integer keys including uniform and skewed normal key distributions, as shown in Figure 9. We observe that the level hash table achieves the approximate maximum load factors for the different key distributions. The reason is that hash functions map keys to random hash values, and hence whatever the key distribution is, the generated hash value distribution is still randomized. Keys are then randomly distributed among buckets of hash table based on their hash values. Therefore, the skewed key distribution doesn't result in the skewed hash value



(a) DRAM (b) NVM-200ns/200ns (c) NVM-200ns/600ns (d) NVM-200ns/1000ns

Figure 10: Insertion latency of different hashing schemes in DRAM and NVM with different read/write latencies.

distribution without significantly affecting the maximum load factor of hash table.

In the following experiments, we set 4 slots per buckets for BCH, PFHT and level hashing, like existing work [13, 22, 25].

4.2.2 Insertion Latency

To evaluate the insertion latency of different hashing schemes, we insert unique key-value items to empty BCH, PFHT, level and path hash tables until reaching their maximum load factors. In the meantime, we measure the average latency of each insertion operation when hash tables are in the different load factors. We evaluate these hashing schemes on both DRAM and the persistent memory with different read/write latencies, i.e., 200ns/200ns, 200ns/600ns, and 200ns/1000ns. On persistent memory, these hash tables are implemented with data consistency guarantee as described in Section 4.1.

Figure 10a shows the average latency of each insertion operation in different hash tables in DRAM. Figures 10b, 10c and 10d show the average insertion latency of different hash tables in persistent memory. Compared with the experimental results in Figures 10a and 10b, we observe that the insertion latency in persistent memory is much higher than that in DRAM, while the read/write latency of persistent memory (200ns) is close to that of DRAM (136ns). The main reason is that each inserted item must be flushed into persistent memory via `CLFLUSH`, and the ordering of writes is ensured via `MFENCE` for consistency guarantee, significantly increasing the latency.

As shown in Figure 10, with the increase of the load factors, the insertion latency of BCH sharply increases, due to causing many eviction operations to deal with hash collisions. The insertion performance of BCH becomes worse in persistent memory, since the eviction operations in BCH cause many cache line flushes and memory fences. The insertion latency of PFHT increases since many items need to be inserted in the stash when the load factor is high. PFHT uses the chained hash table to manage the items in the stash. An insertion in the stash needs to allocate the node space and revise pointers, causing extra writes. The insertion latency of path hashing is higher than that of PFHT in DRAM as shown in Figure 10a, while becoming lower than that of PFHT in persistent memory as shown in Figure 10b, for a high load factor (e.g., ≥ 0.7). The reason is that path hashing performs only multiple read operations to find an empty bucket for inserting an item without extra write operations. Reads are much cheaper than writes in persistent memory. In both DRAM and persistent memory, level hashing has the best insertion performance due to probing fewer buckets than path hashing and rarely causes extra writes. From Figure 10b, we observe when the load factor is larger than 0.8, level hashing reduces the insertion latency by over 67%, 43%, and 30%, i.e., speeding up the insertions by over 3.0×, 1.8×, and 1.4×, compared with BCH, PFHT and path hashing.

4.2.3 Update Latency

We investigate the update latency of different hash tables with different load factors in persistent memory. The read/write latency of NVM is 200ns/600ns. As shown in Figure 11, we observe that the update latencies of BCH, PFHT, and path hashing are similar since the update only operates on a single key-value item. In a low load factor (e.g., < 0.5), their update latency are significantly higher than their insertion latency as shown in Figure 10c, since each update operation needs to use the expensive logging to guarantee consistency.

To show the efficiency of our proposed opportunistic log-free update scheme as presented in Section 3.3(4), we also evaluate the update latency of *Level w/o Opp* which indicates the level hashing without this opportunistic scheme. Compared with BCH, PFHT, path hashing, and Level w/o Opp, we observe that level hashing efficiently reduces the update latency by 15% $\sim$ 52%, i.e., speeding up the updates by 1.2× $\sim$ 2.1×.

4.2.4 Search Latency

We evaluate the performance of both positive and negative searches in different hash tables on the persistent memory. For a search operation, if the target item is found in the hash table, the query is positive. Otherwise,

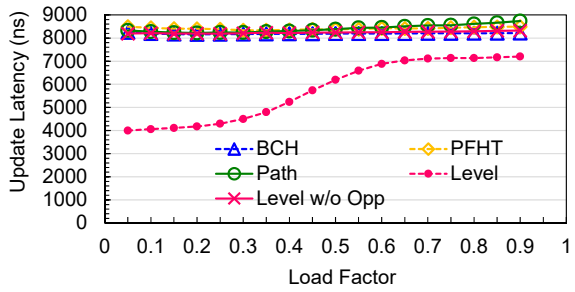


Figure 11: Average update latency of different hashing schemes in NVM.

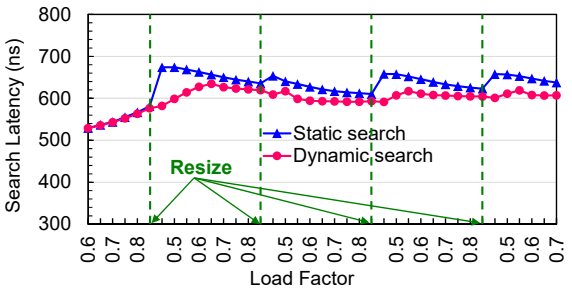


Figure 13: Average search latency of level hashing before and after resizing.

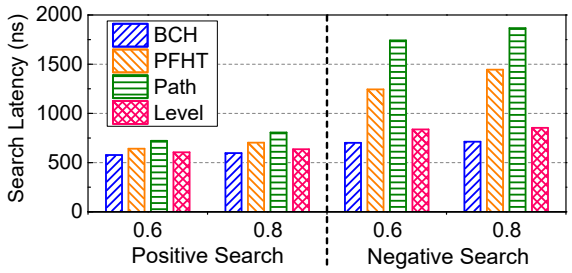


Figure 12: Average latency of positive and negative searches in level hashing.

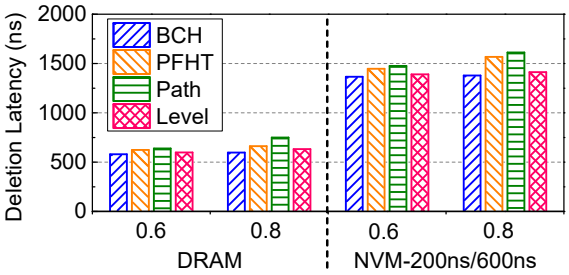


Figure 14: Average deletion latency of different hashing schemes in DRAM and NVM.

it is negative. When hash tables are in two typical load factors, i.e., 0.6 and 0.8 [68], we perform 1 million positive and negative searches respectively and measure their average latency, as shown in Figure 12.

We observe that higher load factor results in higher search latency for each hash table. Among these hash tables, BCH has the lowest positive search latency due to probing the fewest positions to find a target item. The positive search latency of level hashing is very close to that of BCH since level hashing probes at most two buckets in the bottom level when failing to find the target item in the top level. PFHT has higher positive search latency than BCH and level hashing, due to linearly searching the stash when failing to find the target item in the main hash table. The chains in the stash become long when the load factor is high, e.g., 0.8. Path hashing has the highest search latency due to probing multi-level buckets. Moreover, the negative search has higher search latency than the positive search for each hash table, since the negative search must traverse all positions that the target item may be stored. Level hashing probes at most four buckets for each search operation, which has the constant worst-case search time complexity like BCH. Nevertheless, PFHT uses chained hashing to manage the items in the stash with the $O(N_1)$ worst-case search time complexity [32], where N_1 is the number of items in the stash. The path hash table has about $\log(N_2)/2$ levels, thus producing the $O(\log(N_2))$ worst-case search time complexity, where N_2 is the total number of buckets.

To show the effectiveness of the proposed dynamic search scheme in Section 3.2(3), we evaluate the average latency of positive searches before and after resizing in level hashing. We insert unique keys into the level hash table and resize the hash table when its load factor reaches 0.85, until the level hash table is resized four times. When the level hash table is in different load factors, we perform 1-million uniform random searches. The average search latency is shown in Figure 13. We observe the search latency using the static search sharply increases after each resizing since most items are in the bottom level at this point. By performing the dynamic search, we efficiently reduce the search latency of the hash table after the first resizing.

4.2.5 Deletion Latency

We investigate the deletion latency of different hash tables in DRAM and persistent memory, as shown in Figure 14. In DRAM, we observe that the deletion latency of each hash table is approximate to its search latency since the deletion operation first searches the position storing the target item and then sets the position to null. The set-null operation has very low latency in DRAM due to being completed in CPU caches. But in persistent memory, the set-null operation causes high latency since the modified data have to be flushed into NVM for consistency guarantee. Like the positive search performance, BCH and level hashing have better deletion performance than PFHT and path hashing.

4.2.6 Resizing Time

To evaluate the resizing performance of different hashing schemes, we resize the hash tables when their load factors reach the same threshold, i.e., 0.85 (the maximum load factor that the 4-slot BCH can achieve as shown in Figure 8). We measure the total time that different hashing schemes complete the resizing. In order to show the benefit of our proposed in-place resizing scheme, we also evaluate the resizing performance of Level-Trad, which indicates the level hashing using the traditional resizing scheme [48], as shown in Figure 15.

We observe that the level hashing reduces the resizing total time by about 76%, i.e., speeding up the resizing by 4.3×, compared with Level-Trad. The reason is that the level hashing by using the in-place resizing scheme only needs to rehash the key-value items in the bottom level, significantly reducing the number of rehashed items. The number of buckets in the bottom level is 1/3 of all buckets. An item is stored in the bottom level only when both buckets in the top level are full. Hence, the items in the bottom level to be rehashed are always less than 1/3 of all items in the level hash table. Moreover, BCH, PFHT, path hashing and Level-Trad have the similar resizing time, since they need to rehash all items from the old hash table to the new one.

4.2.7 Concurrent Throughput

Since PFHT and path hashing do not support the concurrent access, we compare the concurrent level hashing with the state-of-the-art concurrent hash table in DRAM, i.e., libcuckoo [6, 37]. We focus on general hashing schemes without special hardware support. We hence use the libcuckoo with fine-grained locking instead of that with hardware transaction memory (HTM). We vary the number of concurrent threads from 2 to 16 and use the YCSB workloads with different search/insertion ratios. We use the default configuration of YCSB, i.e., zipfian request distribution with 0.99 skewness. The experimental results are shown in Figure 16. We observe that the concurrent level hashing has $1.6\times - 2.1\times$ higher throughput than libcuckoo in all workloads. This is because libcuckoo incurs iterative eviction operations during an insertion. Thus an insertion needs to lock an entire cuckoo path [37], i.e., locking all slots in the eviction sequence. As a result, all insertion and search operations in other threads that access any one slot in the locked cuckoo path have to wait until the current insertion completes, thus reducing the concurrent performance. Unlike libcuckoo, in the concurrent level hashing, most insertions lock only one slot and a few insertions lock at most two slots, reducing the concurrent conflictions and thus delivering high performance.

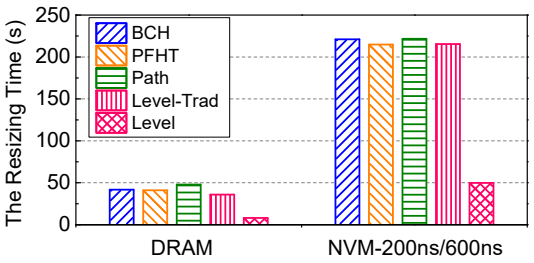


Figure 15: The resizing time of different hashing schemes in DRAM and NVM.

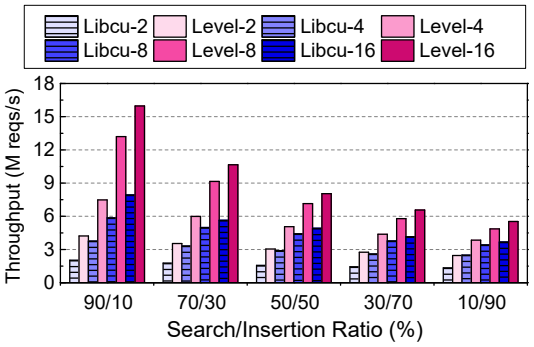


Figure 16: The concurrent request throughputs of level hashing and libcuckoo with 2/4/8/16 threads in DRAM.

5 Related Work

Tree-based Index Structures on NVM. For tree-based index structures, most work focuses on B-tree [30]. Chen et al. [16] propose a PCM-friendly B$^+$-tree that reduces PCM writes by allowing leaf nodes to be unsorted, without considering the data consistency of B$^+$-tree in PCM. Venkataraman et al. [58] propose the CDDS B-tree that leverages versioning and `CLFLUSH` and `MFENCE` instructions to achieve data consistency in B-tree. Yang et al. [64] propose the NV-Tree to guarantee the consistency of only leaf nodes in B$^+$-tree while relaxing that of internal nodes. The internal nodes can be rebuilt based on leaf nodes in case of system failures. NV-Tree reduces the number of cache line flushes due to only persisting the leaf nodes. Chen et al. [17] propose a write-atomic B-tree (wB$^+$-Tree) that adds a bitmap in each node of B$^+$-tree and achieves consistency via the atomic update of the bitmap. However, wB$^+$-Tree requires expensive redo logging for node split operations. Oukid et al. [46] propose the FP-tree, a persistent B-Tree for hybrid DRAM-NVM main memory, in which only the leaf nodes of B$^+$-tree are persisted in NVM while the internal nodes are stored in DRAM. Hwang et al. [30] propose the log-free failure-atomic shift (FAST) and in-place rebalance (FAIR) algorithms for B$^+$-tree in persistent memory via tolerating transient inconsistency. Except B-tree, Lee et al. [35] focus on the radix tree on persistent memory and propose Write Optimal Radix

Trees (WORT) that achieve data consistency via 8-byte atomic writes. Unlike them, our paper focuses on the hashing-based index structure on NVM.

Hashing-based Index Structures on NVM. Existing work on hashing-based index structures for NVM, such as PFHT [22] and path hashing [68, 69], mainly focuses on reducing NVM writes without considering the consistency issue on NVM. Unlike them, our proposed level hashing guarantees the consistency of hash table via (opportunistic) log-free schemes without expensive logging/CoW mechanisms in most cases, while delivering high performance and rarely incurring extra NVM writes. Moreover, we observe that the resizing in hash tables is expensive for the endurance and performance of NVM systems, which however is overlooked by existing work. Our paper proposes a cost-efficient in-place resizing scheme to significantly reduce the NVM writes and alleviate performance penalty during resizing.

Concurrent Hashing Index Structures. MemC3 [25] proposes an optimistic concurrent cuckoo hashing that is optimized for the multi-reader and single-writer concurrency by using a global lock and version counters. The Intel Threading Building Blocks (TBB) [3] provides a chaining-based concurrent hash table using per-bucket fine-grained locking. Libcuckoo [37] is a multi-reader and multi-writer concurrent cuckoo hashing scheme using fine-grained locking that delivers higher performance than the TBB hash table. Our proposed concurrent level hashing has higher concurrent throughput than libcuckoo due to locking fewer slots for insertions. To support variable-length keys and values, MemC3 [25] stores a short summary of the key and a pointer for each key-value item in the hash table. This pointer points to the full key-value term that is stored outside the hash table. The same method can be added into level hashing as needed to support variable-length keys and values.

6 Conclusion

In order to efficiently index the data on persistent memory, this paper proposes a write-optimized and high-performance hashing index scheme, called level hashing, along with a cost-efficient in-place resizing scheme and (opportunistic) log-free consistency guarantee schemes. Level hashing efficiently supports multi-reader and multi-writer concurrency via simply using fine-grained locking. We have evaluated level hashing in both DRAM and NVM platforms. Compared with the state-of-the-art hashing schemes, level hashing achieves $1.4\times-3.0\times$ speedup for insertions, $1.2\times-2.1\times$ speedup for updates, and over $4.3\times$ speedup for resizing while maintaining high search and deletion performance. Compared with the start-of-the-art concurrent hashing scheme, the concurrent level hashing improves the throughput by $1.6\times-2.1\times$.

Acknowledgments

This work was supported by National Key Research and Development Program of China under Grant 2016YFB1000202, and National Natural Science Foundation of China (NSFC) under Grant 61772212. We are grateful to our shepherd, Steven Swanson, and the anonymous reviewers for their constructive feedback and suggestions.

References

[1] Introducing Intel Optane Technology - Bringing 3D XPoint Memory to Storage and Memory Products. https://newsroom.intel.com/press-kits/introducing-intel-optane-technology-bringing-3d-xpoint-memory-to-storage-and-memory-products/, 2015.

[2] Quartz: A DRAM-based performance emulator for NVM. https://github.com/HewlettPackard/quartz, 2015.

[3] Intel® Threading Building Blocks. https://www.threadingbuildingblocks.org/, 2017.

[4] Intel® Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/en-us/isa-extensions, 2018.

[5] JAVA HashMap. http://www.docjar.com/html/api/java/util/HashMap.java.html, 2018.

[6] Libcuckoo: A high-performance, concurrent hash table. https://github.com/efficient/libcuckoo, 2018.

[7] Memcached. https://memcached.org/, 2018.

[8] Redis. https://redis.io/, 2018.

[9] AKINAGA, H., AND SHIMA, H. Resistive random access memory (ReRAM) based on metal oxides. *Proceedings of the IEEE 98*, 12 (2010), 2237–2251.

[10] APALKOV, D., KHVALKOVSKIY, A., WATTS, S., NIKITIN, V., TANG, X., LOTTIS, D., MOON, K., LUO, X., CHEN, E., ONG, A., DRISKILL-SMITH, A., AND KROUNBI, M. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC) 9*, 2 (2013), 13.

[11] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40, ACM, pp. 53–64.

[12] BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517.

[13] BRESLOW, A. D., ZHANG, D. P., GREATHOUSE, J. L., JAYASENA, N., AND TULLSEN, D. M. Horton tables: Fast hash tables for in-memory data-intensive computing. In *USENIX Annual Technical Conference (USENIX ATC)* (2016).

[14] BYERS, J., CONSIDINE, J., AND MITZENMACHER, M. Simple load balancing for distributed hash tables. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)* (2003).

[15] CALCIU, I., SEN, S., BALAKRISHNAN, M., AND AGUILERA, M. K. Black-box concurrent data structures for NUMA architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[16] CHEN, S., GIBBONS, P. B., AND NATH, S. Rethinking database algorithms for phase change memory. In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)* (2011).

[17] CHEN, S., AND JIN, Q. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment 8*, 7 (2015), 786–797.

[18] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[19] COMER, D. Ubiquitous b-tree. *ACM Computing Surveys (CSUR) 11*, 2 (1979), 121–137.

[20] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)* (2009).

[21] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC)* (2010).

[22] DEBNATH, B., HAGHDOOST, A., KADAV, A., KHATIB, M. G., AND UNGUREANU, C. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads (INFLOW)* (2015).

[23] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation techniques for main memory database systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)* (1984).

[24] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)* (2014).

[25] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[26] GAO, H., GROOTE, J. F., AND HESSELINK, W. H. Almost wait-free resizable hashtables. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)* (2004).

[27] GARCIA-MOLINA, H., AND SALEM, K. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering 4*, 6 (1992), 509–516.

[28] GUERRA, J., MÁRMOL, L., CAMPELLO, D., CRESPO, C., RANGASWAMI, R., AND WEI, J. Software persistent memory. In *USENIX Annual Technical Conference (USENIX ATC)* (2012).

[29] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch hashing. In *Proceedings of the International Symposium on Distributed Computing (DISC)* (2008).

[30] HWANG, D., KIM, W.-H., WON, Y., AND NAM, B. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)* (2018).

[31] KIM, W.-H., KIM, J., BAEK, W., NAM, B., AND WON, Y. NVWAL: exploiting nvram in write-ahead logging. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[32] KNUTH, D. E. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[33] KOCBERBER, O., GROT, B., PICOREL, J., FALSAFI, B., LIM, K., AND RANGANATHAN, P. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2013).

[34] LAI, C., JIANG, S., YANG, L., LIN, S., SUN, G., HOU, Z., CUI, C., AND CONG, J. Atlas: Baidu's key-value storage system for cloud data. In *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST)* (2015).

[35] LEE, S. K., LIM, K. H., SONG, H., NAM, B., AND NOH, S. H. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2017).

[36] LI, S., LIM, H., LEE, V. W., AHN, J. H., KALIA, A., KAMINSKY, M., ANDERSEN, D. G., SEONGIL, O., LEE, S., AND DUBEY, P. 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 (ISCA)* (2015).

[37] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)* (2014).

[38] LIM, H., KAMINSKY, M., AND ANDERSEN, D. G. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)* (2017).

[39] LIU, M., ZHANG, M., CHEN, K., QIAN, X., WU, Y., ZHENG, W., AND REN, J. DUDETM: building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[40] LIU, Y., ZHANG, K., AND SPEAR, M. Dynamic-sized nonblocking hash tables. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing (PODC)* (2014).

[41] LIU, Z., CALCIU, I., HERLIHY, M., AND MUTLU, O. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2017).

[42] MAURER, W. D., AND LEWIS, T. G. Hash table methods. *ACM Computing Surveys (CSUR) 7*, 1 (1975), 5–19.

[43] MITZENMACHER, M. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems 12*, 10 (2001), 1094–1104.

[44] MUELLER, W., AICHMAYR, G., BERGNER, W., ERBEN, E., HECHT, T., KAPTEYN, C., KERSCH, A., KUDELKA, S., LAU, F., LUETZEN, J., ORTH, A., NUETZEL, J., SCHLOESSER, T., SCHOLZ, A., SCHROEDER, U., SIECK, A., SPITZER, A., STRASSER, M., WANG, P.-F., WEGE, S., AND WEIS, R. Challenges for the dram cell scaling to 40nm. In *IEEE International Electron Devices Meeting (IEDM)* (2005).

[45] NARAYANAN, D., AND HODSON, O. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2012).

[46] OUKID, I., LASPERAS, J., NICA, A., WILLHALM, T., AND LEHNER, W. FPTree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2016).

[47] PAGH, R., AND RODLER, F. F. Cuckoo hashing. In *Proceedings of the European Symposium on Algorithms (ESA)* (2001).

[48] PIGGIN, N. ddds: "dynamic dynamic data structure" algorithm, for adaptive dcache hash table sizing. linux kernel mailing list. https://lwn.net/Articles/302132/, 2008.

[49] PITTEL, B. Linear probing: the probable largest search time grows logarithmically with the number of records. *Journal of algorithms 8*, 2 (1987), 236–249.

[50] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)* (2009).

[51] RUMBLE, S. M., KEJRIWAL, A., AND OUSTERHOUT, J. K. Log-structured memory for DRAM-based storage. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2014).

[52] SEO, J., KIM, W.-H., BAEK, W., NAM, B., AND NOH, S. H. Failure-atomic slotted paging for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[53] SHALEV, O., AND SHAVIT, N. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM 53*, 3 (2006), 379–405.

[54] SHUN, J., AND BLELLOCH, G. E. Phase-concurrent hash tables for determinism. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2014).

[55] SUN, Y., HUA, Y., JIANG, S., LI, Q., CAO, S., AND ZUO, P. Smartcuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)* (2017).

[56] THOZIYOOR, S., AHN, J. H., MONCHIERO, M., BROCKMAN, J. B., AND JOUPPI, N. P. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *International Symposium on Computer Architecture (ISCA)* (2008).

[57] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX Annual Technical Conference (USENIX ATC)* (2011).

[58] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2011).

[59] VOLOS, H., MAGALHAES, G., CHERKASOVA, L., AND LI, J. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference (Middleware)* (2015).

[60] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[61] WONG, H.-S. P., RAOUX, S., KIM, S., LIANG, J., REIFENBERG, J. P., RAJENDRAN, B., ASHEGHI, M., AND GOODSON, K. E. Phase change memory. *Proceedings of the IEEE 98*, 12 (2010), 2201–2227.

[62] XIA, F., JIANG, D., XIONG, J., AND SUN, N. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)* (2017).

[63] XU, J., AND SWANSON, S. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2016).

[64] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: reducing consistency cost for nvm-based single level systems. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)* (2015).

[65] YU, X., BEZERRA, G., PAVLO, A., DEVADAS, S., AND STONEBRAKER, M. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment 8*, 3 (2014), 209–220.

[66] ZHANG, K., WANG, K., YUAN, Y., GUO, L., LEE, R., AND ZHANG, X. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1226–1237.

[67] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)* (2009).

[68] ZUO, P., AND HUA, Y. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST)* (2017).

[69] ZUO, P., AND HUA, Y. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems 29*, 5 (2018), 985–998.