Level Hashing: A High-performance and Flexible-resizing Persistent Hashing Index Structure

PENGFEI ZUO, YU HUA, and JIE WU, Wuhan National Laboratory for Optoelectronics, School of Computer, Huazhong University of Science and Technology, China

Non-volatile memory (NVM) technologies as persistent memory are promising candidates to complement or replace DRAM for building future memory systems, due to having the advantages of high density, low power, and non-volatility. In main memory systems, hashing index structures are fundamental building blocks to provide fast query responses. However, hashing index structures originally designed for dynamic random access memory (DRAM) become inefficient for persistent memory due to new challenges including hardware limitations of NVM and the requirement of data consistency. To address these challenges, this article proposes *level hashing*, a write-optimized and high-performance hashing index scheme with low-overhead consistency guarantee and cost-efficient resizing. Level hashing provides a sharing-based two-level hash table, which achieves constant-scale worst-case time complexity for search, insertion, deletion, and update operations, and rarely incurs extra NVM writes. To guarantee the consistency with low overhead, level hashing leverages log-free consistency schemes for deletion, insertion, 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 to expand a hash table and rehash 2/3 of buckets to shrink a hash table, thus significantly improving the resizing performance and reducing the number of rehashed buckets. Extensive experimental results show that the level hashing speeds up insertions by 1.4×−3.0×, updates by 1.2×−2.1×, expanding by over 4.3×, and shrinking by over 1.4× while maintaining high search and deletion performance compared with start-of-the-art hashing schemes.

CCS Concepts: • **Hardware** → **Non-volatile memory**; • **Information systems** → **Data structures**; *Point lookups*;

Additional Key Words and Phrases: Persistent memory, hashing index structure, write optimization, crash consistency

ACM Reference format:
Pengfei Zuo, Yu Hua, and Jie Wu. 2019. Level Hashing: A High-performance and Flexible-resizing Persistent Hashing Index Structure. *ACM Trans. Storage* 15, 2, Article 13 (June 2019), 30 pages.
https://doi.org/10.1145/3322096

13

This work was supported by National Natural Science Foundation of China (NSFC) under Grant 61772212.
The preliminary version appears in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2018, pages: 461–476, as “Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory.”
Authors' addresses: P. Zuo, Y. Hua (corresponding author), and J. Wu, Wuhan National Laboratory for Optoelectronics, School of Computer, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, Hubei, China, 430074; emails: {pfzuo, csyhua, wujie}@hust.edu.cn.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
1553-3077/2019/06-ART13 $15.00
https://doi.org/10.1145/3322096

1 INTRODUCTION

As dynamic random access memory (DRAM) technology suffers from the limitations in density scaling and power leakage [41, 55], non-volatile memory (NVM) technologies, such as 3D XPoint [24], phase-change memory (PCM) [60], resistive random-access memory (ReRAM) [1], and spin-transfer torque random-access memory (STT-RAM) [2], are expected to substitute or complement DRAM as the next-generation main memory. NVM as *persistent memory* is able to persistently store data for instantaneous failure recovery. Due to the access latency close to DRAM and byte-addressable benefit, persistent memory is directly accessed through the memory bus via using central processing unit (CPU) load and store instructions, avoiding high overheads of conventional block-based interfaces [10, 37, 62, 63]. Nevertheless, NVM typically suffers from low write performance and limited write endurance [48, 66].

The significant changes of memory architectures and characteristics result in the inefficiency of indexing data in the conventional manner that overlooks new NVM device properties and the requirement of data consistency [32, 43, 57, 63, 67]. A large number of existing works have improved tree-based index structures to efficiently adapt to persistent memory, such as consistent-and-durable-data-structures (CDDS) B-tree [57], non-volatile (NV)-Tree [63], wB$^+$-Tree [9], fingerprinting persistent (FP)-Tree [43], write optimal radix tree (WORT) [32], and failure-atomic shift and failure-atomic in-place rebalance (FAST&FAIR) [23]. Tree-based index structures typically have the search time complexity of average O(log(N)) where N is the size of data structures [4, 11]. Unlike tree-based index structures, hashing-based index structures are flat data structures, which are able to achieve constant-scale search time complexity, i.e., O(1). Due to providing fast lookup responses, hashing index structures have been widely used in main memory systems. For example, they are used to index main memory databases [19, 30, 36, 64] and are fundamental components in in-memory key-value stores [17, 33, 40, 49, 65], 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 maintained in persistent memory should avoid any data inconsistency, when system failures (e.g., power loss and system crashes) occur [20, 32, 43]. However, the new architecture that NVM is directly accessed through the memory bus results in high overhead to guarantee consistency. First, memory writes are usually reordered by CPU and memory controller [10, 12]. To ensure the ordering of memory writes for consistency guarantee, we have to employ cache line flush and memory fence instructions, causing high performance overhead [9, 28, 42, 63]. 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) [9, 12, 16, 59]. 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 [23, 32, 57, 63].

(2) ***Performance Degradation for Reducing Writes.*** Memory writes in NVM consume the limited endurance and cause higher latency and energy than reads [48, 66]. 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 [14, 67] demonstrates that common hashing schemes such as chained hashing, hopscotch hashing [22], and cuckoo hashing [45, 54] usually cause many extra memory writes for dealing with hash collisions. The write-friendly hashing schemes, such as PCM-friendly hash table (PFHT) [14] and path hashing [67], 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. At this time, the hash table needs to be resized via expanding its size [18, 22, 46, 56]. Moreover, 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 a hash table needs to create a new hash table whose size is usually doubled for expanding or half for shrinking, 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 article proposes a write-optimized and high-performance hashing index scheme for persistent memory, called *level hashing* [69], with low-overhead consistency guarantee and cost-efficient resizing. Level hashing introduces a write-optimized two-level hash table, i.e, level hash table, to achieve high-access performance even in the worst case. Moreover, level hashing leverages log-free schemes to guarantee the consistency of write operations with low overhead. To cost-efficiently resize the level hash table, level hashing leverages in-place expanding and shrinking schemes to significantly improve the resizing performance and reduce the number of NVM writes. Specifically, the contributions of this article are listed as follows:

— ***Write-optimized and High-performance Hash Table Structure.*** We propose a sharing-based two-level hash table structure (Section 3.1), in which each search, deletion, and update operation only needs to probe at most four buckets to find the target key-value item, and hence, has the constant-scale worst-case time complexity 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 Expanding and Shrinking.*** We propose a cost-efficient in-place resizing scheme for level hashing (Section 3.2), which rehashes only 1/3 of buckets in the hash table instead of the entire hash table to expand a hash table, and rehashes 2/3 buckets instead of the entire hash table to shrink a hash table, thus significantly improving the resizing performance and reducing NVM writes. 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.
— ***Low-overhead Consistency Guarantee.*** We propose log-free consistency guarantee schemes for insertion, deletion, and resizing operations in level hashing (Section 3.3). 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 and CoW mechanisms. Moreover, for update operation, we propose an opportunistic log-free scheme to update an item without the need of logging and 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 and CoW.
— ***Efficient Concurrency.*** We propose an efficient concurrent scheme for level hashing by leveraging fine-grained locking (Section 3.4). The level hash table has no cascading writes, which enables level hashing to efficiently support multi-reader and multi-writer concurrency via simply using fine-grained locking. An insertion locks only one slot in most cases, and hence, the concurrent level hashing delivers high performance.

— ***Real Implementation and Evaluation.*** We have implemented level hashing[1] and evaluated it in both real-world DRAM and simulated NVM platforms. Experimental results demonstrate that level hashing achieves 1.4×–3.0× speedup for insertions, 1.2×–2.1× speedup for updates, 4.3× speedup for expanding, and 1.4× speedup for shrinking a hash table, while maintaining high search and deletion performance, compared with start-of-the-art hashing schemes including BCH [17], PFHT [14], and path hashing [67]. 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 [34].

The rest of this article is organized as follows. The background and motivation are described in Section 2 . Section 3 presents the design of level hashing, and Section 4 presents the implementation details. The performance evaluation is shown in Section 5. We discuss the related work in Section 6 and conclude this article in Section 7.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce the data consistency guarantee in persistent memory. We then present the background of hashing index structures.

2.1 Data Consistency in NVM

In order to efficiently handle possible system failures and achieve instantaneous failure recovery, NVM has been well explored and exploited to build persistent memory systems. However, since the persistent systems typically include volatile storage components, e.g., CPU caches and possible DRAM, the potential problem of data crash consistency has to be addressed that is interpreted as preventing data from being partially updated or loss on system failures. To guarantee data consistency in persistent memory, it is essential to ensure the ordering of memory writes [9, 32, 63]. However, the CPU and memory controller usually 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 enforce the ordering of memory writes, like existing state-of-the-art work [9, 32, 43, 57, 63]. The `CLFULSH` and `MFENCE` instructions are provided by the Intel x86 architecture [26]. Specifically, `CLFULSH` evicts a dirty cache line from CPU 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 executing `CLFLUSH` instructions [26]. However, `CLFLUSH` and `MFENCE` instructions result in significant system performance decrease [9, 12, 57]. 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, i.e., the memory bus width for 64-bit CPUs [9, 32, 43, 57, 63]. If the size of the updated data is larger than 8 bytes and a system failure occurs while performing the update, the data will be corrupted. Existing techniques, such as logging and CoW mechanisms, are used to guarantee consistency of the data whose sizes are larger than an atomic write. Logging techniques include undo logging and redo logging. Undo logging first stores old data in a log before being modified and then updates them. Redo logging first buffers the newly written data in a log and then updates the old data. 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 mechanisms have to write twice for each updated data. The ordering of the two-time writes also needs to be ensured using `CLFLUSH` and `MFENCE` instructions, significantly hurting the system performance.

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

2.2 Hashing Index Structures for NVM

2.2.1 Conventional Hashing Schemes. Hashing index structures have been widely used in current main memory systems, e.g., main memory databases [15, 19, 30, 36, 64], and in-memory key-value stores [17, 33, 40, 49, 50], 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* [29] is a popular scheme to deal with hash collisions, by storing the conflicting items in a linked list via pointers. Nevertheless, the chained hashing consumes extra memory space due to maintaining 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 using 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)* [5, 17, 34] is a memory-efficient open-addressing scheme, which is 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 [5, 14, 17, 34]. Hence, the BCH in our article uses two hash functions.

2.2.2 Hashing Schemes for NVM. The hashing schemes mentioned above mainly consider the properties of 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 [48, 66]. 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 [14, 67]. More importantly, the traditional hashing schemes do not consider data consistency and, hence, cannot directly work on persistent memory.

Hashing schemes [14, 67] have been improved to efficiently adapt to NVM, which mainly focus on reducing NVM writes in hash tables. Debnath et al. [14] propose a *PCM-friendly Hash Table (PFHT)* that is a variant of BCH for reducing writes to PCM. PFHT modifies the BCH to allow only one-time eviction when inserting a new item, which can reduce 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 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 [67, 68] 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 reduces NVM writes at the cost of reducing search performance due to the need of traversing two paths until finding the target key-value item for a 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

Table 1. Comparisons Among Level Hashing and State-of-the-art Memory-efficient Hashing Schemes

	BCH	PFHT	Path hashing	Level hashing
Memory Efficiency	√	√	√	√
Search	√	–	–	√
Deletion	√	–	–	√
Insertion	×	–	–	√
Write-friendly	×	√	√	√
Resizing	×	×	×	√
Consistency	×	×	×	√

In this Table, "×" Indicates a Bad Performance, "√" Indicates a Good Performance, and "–" Indicates a Moderate Performance in the Corresponding Metrics.

for insertion operations 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 efficiency of the resizing operation that often causes a large number of NVM writes, and cannot directly work on persistent memory due to no consistency guarantee. Our article proposes the level hashing that achieves good performance in terms of all these metrics as presented in Section 3, which is also verified in the performance evaluation as shown in Section 5.

2.2.3 Resizing a Hash Table. The resizing of hash table includes expanding and shrinking operations. When the load factor of a hash table reaches a high threshold or a new key-value item fails to be inserted into the hash table, the hash table needs to be expanded by growing its size [46, 56]. Moreover, when the stored items in a hash table are far fewer than its storage units, the hash table also needs to be shrunk via reducing its size. Traditional resizing schemes [38, 46, 52] perform out-of-place resizing, which expand or shrink a hash table by creating a new hash table whose size is larger than the old one for expanding or smaller than the old one for shrinking, and then iteratively rehashing all key-value items from the old hash table to the new one.

To expand a hash table, the size of the new hash table is usually double size of the old one [38, 52, 53, 56] 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 to expand 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 [56]. If expanding the hash table to a too-small size, the new hash table may cause high hash collision rate and poor insertion performance, which will quickly incur another expanding operation. If expanding 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 expanding a hash table has been widely recognized [52, 53, 56]. For example, in the real-world applications, such as Java HashMap [27] and Memcached [40], doubling the size is the default setting for expanding a hash table. For the similar reasons, to shrink a hash table, the size of the new hash table is usually half size of the old one.

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

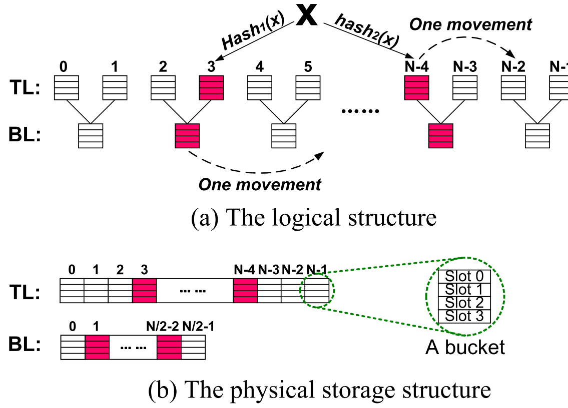


Fig. 1. The hash table structure of level hashing with four slots per bucket. *(In these tables, "TL" indicates the top level, and "BL" indicates the bottom level.)*

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 for persistent memory with cost-efficient resizing and low-overhead consistency guarantee. In this section, we first present the basic data structure of level hashing (Section 3.1), i.e., level hash table, which is able to achieve write optimization, high load factor, and high performance. We then present cost-efficient in-place expanding and shrinking schemes (Section 3.2) for the level hash table to improve the resizing performance and reduce NVM writes during resizing. We then present the (opportunistic) log-free schemes (Section 3.3) to guarantee the consistency of write operations in level hashing with low overhead. We finally present the concurrent level hashing leveraging fine-grained locking (Section 3.4).

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 write-optimized, memory-efficient, and high performance, while avoiding their weaknesses by performing the following major design decisions.

- D_1: ***Multiple Slots in Each Bucket.*** Motivated by the real-world key-value workload characteristics that small key-value items dominate in current key-value stores reported by Baidu [31] and Facebook [3], we set multiple slots in each bucket in the level hash table, e.g., four slots in each bucket as shown in Figure 1. Thus, one bucket can store multiple items each in one slot, which enables the level hash table to be cache-efficient. When one bucket in the level hash table is accessed, multiple items stored in the same bucket can be prefetched into CPU caches in one memory access, thus improving the cache efficiency and reducing the number of memory accesses. Nevertheless, the current hash table has a very low maximum load factor since hash collisions are not dealt with, as shown in Figure 2. The following three decisions are used to improve the maximum load factor by efficiently dealing with hash collisions.
- D_2: ***Using Two Different Hash Functions.*** Since each bucket contains k slots, the hash table can only deal with at most $k - 1$ hash collisions occurring in a single bucket, causing

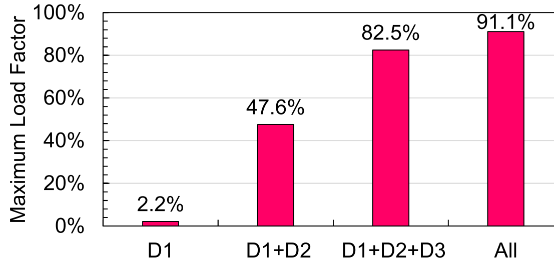


Fig. 2. The maximum load factors when adding different design decisions. *(The maximum load factor is defined as the current load factor of the hash table when an insertion failure occurs. D_1: a one-level hash table with four slots per bucket; $D_1 + D_2$: a hash table with design decisions D_1 and 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 $D_1 + D_2 + D_3 + D_4$.)*

a low maximum load factor. To improve the maximum load factor, we use two different hash functions for the level hash table, i.e., $\texttt{hash}_1()$ and $\texttt{hash}_2()$, like BCH, PFHT, and path hashing, which enables each key to have two hash locations. A new item is inserted into the less-loaded bucket between the two hash locations [6]. Due to the randomization of two independent hash functions, the maximum load factor of hash table is improved significantly as shown in Figure 2.

— ***D_3: Sharing-based Two-level Structure.*** We divide the buckets in the level hash table into two levels, i.e., a top level (TL) and a bottom level (BL), as shown in Figure 1(a). To insert a new key-value item, the item is hashed into the buckets in the top level. The bottom level is used to provide standby positions for the top level to deal with hash collisions. Each bucket in the bottom level is shared by two buckets in the top level; thus, the size of the bottom level is half of the top level. If there is no empty slot in the two top-level buckets for a new key-value item, the item can be stored in its two corresponding standby buckets in the bottom level. By using the two-level table structure, the maximum load factor of hash table is significantly improved as shown in Figure 2. Moreover, since each top-level bucket has one standby bucket, a search operation only needs to probe at most four buckets, thus having the constant-scale worst-case time complexity.

— ***D_4: At Most One Movement for Each Successful Insertion.*** We allow the movement of at most one item for each insertion in the level hash table, which enables items to be evenly distributed among buckets while avoiding the problem of the cascading writes in cuckoo hashing. Specifically, during inserting a new key-value item (I_{new}), if all the four buckets in the two levels are full, we try to move away one key-value item in the four buckets. We first check whether it is possible to move any existing key-value item in the two top-level buckets from one location to its alternative top-level location. If no movement is possible, we further check whether it is possible to move any existing key-value item in the two bottom-level buckets from one location to its alternative bottom-level location. If the movement still fails, the insertion is considered as a failure and the hash table needs to be expanded. 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 structure of a level hash table is shown in Figure 1. Figure 1(a) shows the logical structure of a level hash table containing two-level buckets. The links between two levels indicate the sharing relationships among buckets, instead of pointers. Figure 1(b) shows the physical storage structure of a level hash table, which stores each level in a one-dimensional array.

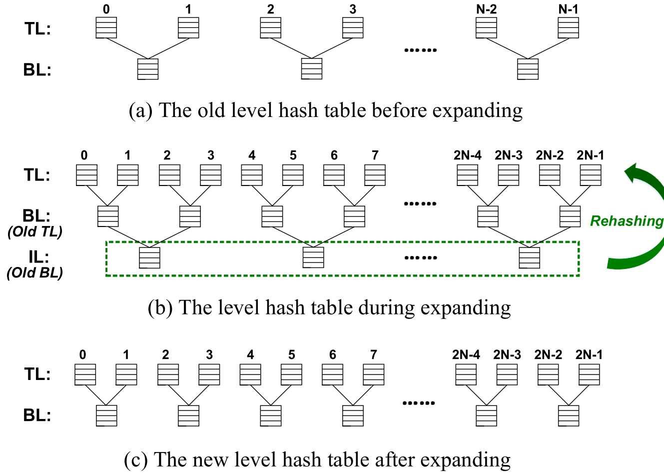


Fig. 3. The in-place expanding in the level hashing. *("IL" indicates the interim level.)*

For an item with the key K, the locations of its two corresponding top-level buckets (i.e., the No.L_{t1} and No.L_{t2} buckets) and its two corresponding bottom-level buckets (i.e., the No.L_{b1} and No.L_{b2} buckets) can be obtained by the following equations:

$$\mathsf{L}_{t1} = \mathsf{hash}_1(\mathsf{K})\%\mathsf{N},\ \mathsf{L}_{t2} = \mathsf{hash}_2(\mathsf{K})\%\mathsf{N} \tag{1}$$

$$\mathsf{L}_{b1} = \mathsf{hash}_1(\mathsf{K})\%(\mathsf{N}/2),\ \mathsf{L}_{b2} = \mathsf{hash}_2(\mathsf{K})\%(\mathsf{N}/2) \tag{2}$$

The computations of Equations (1) and (2) only require the simple bit shift operation since N is a power of 2.

3.2 Cost-efficient In-place Resizing

To improve the resizing performance and reduce NVM writes, we propose *cost-efficient in-place resizing* for level hashing, which includes an in-place expanding scheme to grow the size of a hash table and an in-place shrinking scheme to reduce the size of a hash table.

3.2.1 In-place Expanding. The basic idea of the in-place expanding scheme is to put a new level on 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.

A high-level overview of the in-place expanding process is shown in Figure 3. Before the expanding, the level hash table includes two levels, i.e., a top level (TL) with N buckets and a bottom level (BL) with $N/2$ buckets, as shown in Figure 3(a). During the expanding, we first allocate the memory space with $2N$ buckets as the new top level and put it on top of the old hash table. The level hash table becomes a three-level structure during the expanding, as shown in Figure 3(b). The third level is called the interim level (IL). The in-place expanding scheme rehashes the key-value items stored in the IL into the top-two levels. Each rehashing operation includes reading a key-value 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 expanding, the new hash table becomes a two-level structure again, as shown in Figure 3(c). The rehashing failure, which indicates a rehashed item fails to be inserted into the top-two levels, does not occur when the expanding 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

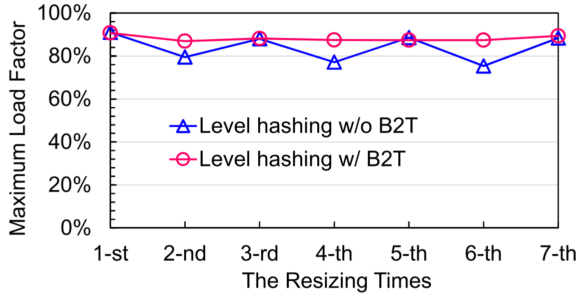


Fig. 4. The load factors when the expandings occur.

the level hash table is able to achieve the load factor of higher than 0.9 (>0.5) as evaluated in Section 5.2.1.

We observe that the new hash table with $3N$ buckets is exactly double size of the old one with $1.5N$ buckets, which meets the demand of real-world applications as discussed in Section 2.2.3. Unlike the traditional out-of-place resizing scheme [46] in which the expanding occurs between the old and new tables, the in-place expanding enables the whole expanding process to occur in a single hash table. Thus, during expanding, search and deletion operations only need to probe one table and compute the hash functions once, thus improving the access performance. More importantly, the in-place expanding 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 expanding, as well as improving the expanding performance. Moreover, during the resizing, our proposed in-place resizing scheme requires less memory space than the traditional out-of-place resizing scheme. The out-of-place resizing scheme requires $4.5N$-bucket memory space including the old hash table with $1.5N$ buckets and the new hash table with $3N$ buckets. Our proposed in-place resizing scheme only requires $3.5N$-bucket memory space including the old hash table with $1.5N$ buckets and a new top level with $2N$ buckets.

Improving the Maximum Load Factor after Expanding. 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 expanding, the top-level buckets are mostly full and the bottom-level buckets are mostly non-full. After expanding, 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 expandings occur. We observe that the maximum load factors in the 2nd, 4th, and 6th expandings are reduced, compared with those in the 1st, 3rd, and 5th expandings. The reason is that the bottom-level buckets are mostly full in the 2nd, 4th, and 6th expandings.

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 expanding. 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 expandings occur via using the B2T scheme. We observe that the maximum load factors in the 2nd, 4th, and 6th expandings are improved.

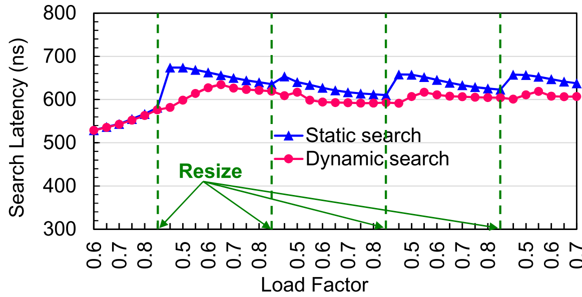


Fig. 5. Average search latency of the static and dynamic search schemes before and after expanding.

Improving the Search Performance after Expanding. After expanding the level hash table, the search performance possibly decreases. The reason is that in the original search scheme (called *static search*) as shown in Section 3.1, we always first probe the top level; if we do not find the target item, we then probe the bottom level. Before expanding, about 2/3 items are in the top level. However, the 2/3 items are in the bottom level after expanding, as shown in Figure 3. Hence, a single search needs to probe two levels in most cases (i.e., about 2/3 probability) after expanding, thus degrading the search performance.

To evaluate the static search performance in level hashing, we insert unique keys into a level hash table and expand it when its load factor reaches 0.85, until the level hash table is expanded four times. When the level hash table is in different load factors, we perform 1-million uniform random search operations. The blue line in Figure 5 shows the average latency of the static search. We observe the search latency of the static search sharply increases after each expanding since most items are in the bottom level at this point.

To address this problem, we propose *a dynamic search scheme* for level hashing. Specifically, for a search, we first compare the numbers of items stored in the top and bottom levels. If the key-value items in the top level are less than those in the bottom level, we first search the bottom level (based on Equation (2)); if we do not find the target item, we then search the top level (based on Equation (1)). If the key-value items in the top level are more than those in the bottom level, we first search the top level and then search the bottom level. Thus, after expanding, the key-value items in the top level are less than those in the bottom level and, hence, we first search the bottom level, thus improving the search performance. The red line in Figure 5 shows the average latency of the dynamic search. We observe that the dynamic search scheme efficiently reduces the average search latency in the level hash table after the first expanding.

3.2.2 In-place Shrinking. The basic idea of the in-place shrinking scheme is to put a new level at the bottom of the old hash table and only rehash the items in the top level of the old hash table when shrinking a level hash table.

A high-level overview of the in-place shrinking process for the level hash table is shown in Figure 6. Before the shrinking, the level hash table includes two levels, i.e., a top level (TL) with $2N$ buckets and a bottom level (BL) with N buckets, as shown in Figure 6(a). During the shrinking, we first allocate the memory space with $N/2$ buckets as the new bottom level and put it at the bottom of the old hash table. The level hash table becomes a three-level structure during the shrinking, as shown in Figure 6(b). The first level is called the interim level (IL). The in-place shrinking scheme rehashes the items in the IL into the bottom-two levels. Each rehashing operation includes reading an item in the IL, inserting the item into the bottom-two levels, and deleting the item from the IL. After all items in the IL are rehashed into the bottom-two levels, the memory space of the IL is

(a) The old level hash table before shrinking

(b) The level hash table during shrinking

(c) The new level hash table after shrinking

Fig. 6. The in-place shrinking in the level hashing. *(“IL” indicates the interim level.)*

Fig. 7. The storage structure of each bucket. *(The header area stores the tokens of all slots and the remaining area stores the slots each with a key-value item.)*

reclaimed. After the shrinking, the new hash table becomes a two-level structure again, as shown in Figure 6(c).

We observe that the new hash table with $1.5N$ buckets is exactly half the size of the old hash table with $3N$ buckets, which meets the demand of real-world applications as presented in Section 2.2.3. Unlike the traditional out-of-place resizing scheme [46] in which the shrinking occurs between the old and new tables, the in-place shrinking enables the whole shrinking process to occur in a single hash table. Moreover, the in-place shrinking rehashes only the top level of the old hash table instead of the entire table. The top level only contains $2/3(=2N/3N)$ of all buckets in the old hash table, thus significantly reducing data movements and NVM writes during the shrinking, as well as improving the shrinking performance.

Moreover, a hash table needs to be shrunk only when it has very few key-value items. Therefore, it does not occur that an item in the interim level fails to be rehashed into the bottom-two levels during the shrinking.

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 [17, 67]. As shown in Figure 7, 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 a 1-bit flag that indicates whether the corresponding slot is empty, e.g., “0” for the empty slot and “1” for the non-empty slot. Since the size of the header area is always smaller than 8 bytes, modifying the tokens only needs to perform an atomic write. But the size of a key-value item is usually larger than 8 bytes. A straightforward method is to guarantee the crash consistency of writing key-value items by using logging or CoW mechanisms, which, however, result in 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 (shrinking and expanding) operations, and *an opportunistic log-free guarantee scheme* for update operation, by leveraging the tokens that can be modified in the atomic-write manner.

3.3.1 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.

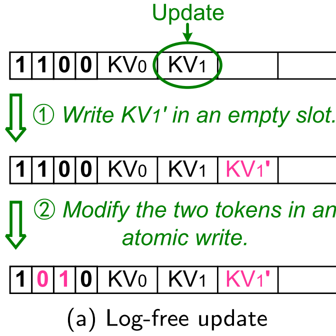
3.3.2 Log-free Insertion. In the level hash table, the insertion operation includes two cases.

(1) *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 need 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 level hash table is in a consistent state when system failures occur.

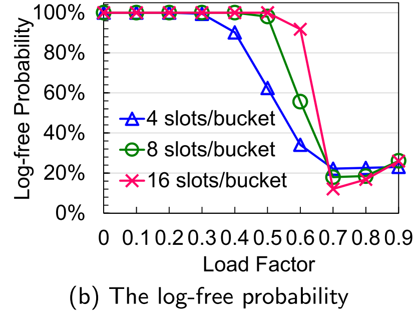
(2) *Moving one item:* The insertion incurs the movement of one existing item. In this case, we need to take two steps to insert an item, and the ordering of executing the two steps is ensured via an `MFENCE`. The first step is to move an existing item into its alternative bucket. We use $\texttt{slot}_{\texttt{cur}}$ 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 "*(1) No item movement.*"

3.3.3 Log-free Expanding and Shrinking. During expanding a level hash table, we need to rehash all key-value items in the interim level (i.e., the old bottom level) into the top-two levels, as shown in Figure 3. During shrinking a level hash table, we need to rehash all key-value items in the interim level (i.e., the old top level) into the bottom-two levels, as shown in Figure 6. For expanding and shrinking, the steps of rehashing each key-value item are the same, as presented in the following.

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 (bottom-two) levels for expanding (shrinking). 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 modify the token of $\texttt{slot}_{\texttt{new}}$ from "0" to "1" and finally modify 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



(a) Log-free update



(b) The log-free probability

Fig. 8. The opportunistic log-free update scheme. *((a) The log-free update scheme; (b) The probability of performing log-free updates with the increase of load factor and the change of the number of slots/bucket.)*

$\mathtt{slot}_{\mathtt{old}}$ is not deleted. There are two duplicate items in the hash table, which, however, does not impact the data consistency, since we can easily remove one of the two duplicates via the following recovery scheme.

Recovery: Since only the first item (I_{first}) to be rehashed in the interim level may be inconsistent in case of a system failure, we only need to check whether there are two duplicates of I_{first} in the hash table. During the recovery, we query the key of I_{first} in the top-two (bottom-two) levels if the hash table is being expanded (shrunk) when the system failure occurs. If two duplicates exist, we directly delete I_{first}. Otherwise, we rehash it. Therefore, the hash table can be recovered in a consistent state. We distinguish whether the hash table is being expanded or shrunk when the system failure occurs via a variable, i.e., `resize_state`, recorded in the persistent metadata of the hash table. That the value of `resize_state` is "0" means the level hash table is not during resizing; a value of "1" means the level hash table is being expanded, and a value of "2" means the level hash table is being shrunk.

3.3.4 Opportunistic Log-free Update. When updating an existing key-value item, if the updated item has two copies in the level hash table, we first delete one and then update the other. If we directly update the 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 by first writing the new or old item into a log and then updating the old item in place, which, however, causes 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 $\text{KV}_1{}'$), we first check whether there is an empty slot in the bucket storing the old item (KV_1).

- **Yes.** If there is an empty slot in the bucket storing the old item (KV_1) as shown in Figure 8(a), we directly write the new item ($\text{KV}_1{}'$) into the empty slot, and then modify the tokens of the old item (KV_1) and new item ($\text{KV}_1{}'$) 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 there is no empty bucket in the bucket storing the old item (KV_1), we first write the old item in a log 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 an empty slot exists in the bucket storing the item to be updated, we update the item without logging. We evaluate the probability that the bucket storing the item to be updated

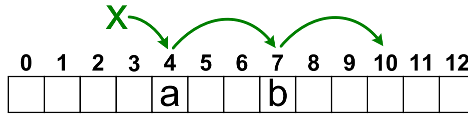


Fig. 9. An insertion operation in the cuckoo hashing. *(The sequence of $x \to a \to b \to \varnothing$ is a cuckoo path. To perform log-free insertion in cuckoo hashing, we first move b from $L7$ to $L10$, and then move a from $L4$ to $L7$, and finally insert x into $L4$.).*

includes at least one empty slot, as shown in Figure 8(b). The probability is related with the number of slots per bucket and the load factor of the hash table. We observe when the load factor of the 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 per 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 bottom-level buckets have the higher probability to include an empty slot than the top-level buckets.

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 key-value 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 scheme can also be used in the hashing schemes with iterative eviction operations during insertions, e.g., cuckoo hashing. Specifically, an insertion operation in cuckoo hashing may iteratively evict key-value items until finding an empty location. The sequence of evicted items is called a cuckoo path [34]. 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 9, the new item x is inserted into the location $L4$, and the sequence of $x \to a \to b \to \varnothing$ 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 a larger number of cores and threads, concurrent data structures become increasingly important [7, 17, 34, 39]. Since having no cascading writes and not using pointers, level hashing is able 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 or write the same slot. Hence, we allocate a fine-grained locking for each slot. When reading or 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; hence, the concurrent level hashing delivers high performance as evaluated in Section 5.2.8.

4 IMPLEMENTATION DETAILS

In this section, we present the implementation details of search, insertion, deletion, update, and resizing operations in the level hashing on persistent memory.

ALGORITHM 1: Search (key)

```
//Compare the number of items in the top and bottom levels
if  |TL| > |BL|
    //Compute two hash locations in the top level
    t1 = hash1(key)%N    // N is the size of the TL
    t2 = hash2(key)%N
    //Lookup the key in the top level
    for each slot in TL[t1] and TL[t2]
        if slot.key == key & slot.token == 1
             return slot.value
    //Compute hash locations in the bottom level
    b1 = t1%(N/2)     // N/2 is the size of the BL
    b2 = t2%(N/2)
    //Lookup the key in the bottom level
    for each slot in BL[b1] and BL[b2]
        if slot.key == key & slot.token == 1
             return slot.value
else
    //Compute hash locations in the bottom level
    b1 = t1%(N/2)     // N/2 is the size of the BL
    b2 = t2%(N/2)
    //Lookup the key in the bottom level
    for each slot in BL[b1] and BL[b2]
        if slot.key == key & slot.token == 1
             return slot.value
    //Compute two hash locations in the top level
    t1 = hash1(key)%N    // N is the size of the TL
    t2 = hash2(key)%N
    //Lookup the key in the top level
    for each slot in TL[t1] and TL[t2]
        if slot.key == key & slot.token == 1
             return slot.value
return NULL
```

4.1 Search

Algorithm 1 presents the pseudo-code of the dynamic search operation in the level hashing. To search a key, we first compare the number of items stored in the top and bottom levels.

If the items in the top level are more than those in the bottom level, we first probe the top level, as presented in Section 3.2.1. The two hash locations of the key in the top level are computed by using two different hash functions, i.e., $hash_1()$ and $hash_2()$. This algorithm lookups the key in the two buckets in the top level. If finding the item with the key, the algorithm returns its value. Otherwise, the two hash locations in the bottom level are computed according to Equation (2). The key is further searched in the two buckets in the bottom level. If finding the item with the key, the algorithm returns its value. If not being found in the bottom level, the key does not exist in the hash table.

If the items in the top level are not more than those in the bottom level, the algorithm first searches the bottom level, and if it does not find the target item, the algorithm then searches the top level.

4.2 Deletion

Algorithm 2 presents the pseudo-code of the deletion operation in the level hashing. To delete an item, we first find the slot storing the target key via the function **Search'**(). The implementation

of **Search'**() is similar to **Search**() in Algorithm 1 while it returns a slot pointer instead of a value. If **Search'**() returns NULL, the target key does not exist in the hash table. Otherwise, the returned slot contains the item with the target key. The algorithm deletes the item by modifying its token from "1" to "0" via an atomic write, as described in Section 3.3.1.

ALGORITHM 2: Delete (key)

```
//Search the slot containing the target key
slot = Search'(key)
//If not find the target key, return FALSE
if slot == NULL
    return FALSE
else
     //Otherwise, delete the target item via an atomic write
     slot.token = 0
     clflush(slot.token)
     mfence()
     return TRUE
```

4.3 Insertion

Algorithm 3 presents the pseudo-code of the insertion operation in the level hashing. To insert a new item with <key, value>, the two hash locations of the key in the top level are computed by using two different hash functions, i.e., $\texttt{hash}_1$() and $\texttt{hash}_2$(). If finding any empty slots in the two top-level buckets, this algorithm inserts the new key-value item into the less-loaded bucket. To insert the new item, the <key, value> pair is first copied into an empty slot and then the token of the empty slot is set to "1" for guaranteeing crash consistency as presented in Section 3.3.2. The order of the two-step writes is ensured by an MFENCE instruction.

If there is no empty slot in the two top-level buckets, the two hash locations of the key in the bottom level are computed. The algorithm further probes the empty slot in the two bottom-level buckets. If an empty slot is found, the new <key, value> pair is inserted into it.

If there is still no empty slot in the two bottom-level buckets, the algorithm checks whether it is possible to move any key-value item in the four buckets (including two top-level and two bottom-level buckets) from its current bucket to its alternative buckets, by using the **TryMovement**() function, as presented in Section 3.1. If the movement fails, the hash table needs to be expanded.

4.4 Update

Algorithm 4 presents the pseudo-code of the update operation in the level hashing. The algorithm first searches the slot storing the target key via the function **Search'**(). If **Search'**() returns NULL, the target key does not exist in the hash table. Otherwise, the returned slot contains the item with the target key-value item. To update this item, the algorithm checks if an empty slot exists in this bucket including the target key-value item. If yes, we can perform log-free updates as presented in Section 3.3.4. The algorithm directly writes the new item into the empty slot, and then modifies the tokens of the old and new items 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.

If there is no empty slot in this bucket containing the target key-value item, the algorithm first writes the old key-value item into the log and then updates the key-value item in place.

ALGORITHM 3: Insert (key, value)

```
//Compute two hash locations in the top level
t1 = hash1(key)%N // N is the size of the TL
t2 = hash2(key)%N
//If there are any empty slots in the two top-level buckets
if  !(|TL[t1]| == |TL[t2]| == ASSOC_NUM) //ASSOC_NUM: the number of slots in each bucket
    //Compare the number of items in the two buckets, and insert the new item into
    //the less-loaded bucket
    if  |TL[t1]| > |TL[t2]|
        for each slot in TL[t2]
            if slot.token == 0
                slot.<key, value> = <key. value>
                clflush(slot.<key, value>)
                mfence()
                slot.token = 1
                clflush(slot.token)
                mfence()
                return TRUE
    else
        for each slot in TL[t1]
            if slot.token == 0
                slot.<key, value> = <key. value>
                clflush(slot.<key, value>)
                mfence()
                slot.token = 1
                clflush(slot.token)
                mfence()
                return TRUE
//Compute two hash locations in the bottom level
b1 = hash1(key)%(N/2)
b2 = hash2(key)%(N/2)
if  !(|BL[t1]| == |BL[t2]| == ASSOC_NUM)
    if  |BL[b1]| > |BL[t2]|
        for each slot in BL[b2]
            if slot.token == 0
                slot.<key, value> = <key. value>
                clflush(slot.<key, value>)
                mfence()
                slot.token = 1
                clflush(slot.token)
                mfence()
                return TRUE
    else
        for each slot in BL[b1]
            if slot.token == 0
                slot.<key, value> = <key. value>
                clflush(slot.<key, value>)
                mfence()
                slot.token = 1
                clflush(slot.token)
                mfence()
                return TRUE
//If there are no empty slots in the four buckets, move away an existing item
return TryMovement(key, value)
```

ALGORITHM 4: Update (key, value)

```
//Search the slot containing the target key
bucket->slot = Search'(key)
//If not find, the update fails
if bucket->slot == NULL
    return FALSE
//Check if an empty slot exists in this bucket
for each slot in this bucket
    //If true, perform log-free update
    if slot.token == 0
        slot.<key, value> = <key. value>
        clflush(slot.<key, value>)
        mfence()
        slot.token = 1
        bucket->slot.token = 0
        clflush(slot.token)
        mfence()
        return TRUE
//No empty slot in this bucket
//Log the old item
Log(bucket->slot)
mfence()
//Update the target item in place
bucket->slot.value = value
clflush(bucket->slot.value)
mfence()
return TRUE
```

4.5 Resizing

Algorithm 5 presents the pseudo-code of the expanding operation in the level hashing. When a level hash table is being expanded, the old bottom level is called the interim level (IL). To rehash each key-value item in the non-empty slot ($\texttt{slot}_{old}$) in the interim level, the algorithm first searches the `key` in the top-two levels for finding an empty slot ($\texttt{slot}_{new}$) via the function **SearchT2Levels**(). The implementation of the **SearchT2Levels**() function is similar to Algorithm 1, which returns a slot pointer that points to an empty slot. Algorithm 5 copies the item from $\texttt{slot}_{old}$ into $\texttt{slot}_{new}$ and then atomically modifies the token of $\texttt{slot}_{new}$ from "0" to "1".

ALGORITHM 5: Expand ()

```
//Rehash each item in the interim level, i.e., the old bottom level
for each slot_old with <key, value> in IL
    //Search an empty slot in the top-two levels
    slot_new = SearchT2Levels(key)
    //Copy the key-value
    slot_new.<key, value> = slot_old.<key, value>
    clflush(slot_new.<key, value>)
    mfence()
    slot_new.token = 1
    clflush(slot_new.token)
    mfence()
    //Delete the item from the old slot
    slot_old.token = 0
    clflush(slot_old.token)
    mfence()
```

Finally, it atomically modifies the token of $\texttt{slot}_{\texttt{old}}$ from "1" to "0". The MFENCE instruction is used to ensure the ordering of the three steps.

Algorithm 6 presents the pseudo-code of the shrinking operation in the level hashing. When a level hash table is being shrunk, the old top level is called the interim level (IL). To rehash each key-value item in the non-empty slot ($\texttt{slot}_{\texttt{old}}$) in the interim level, the algorithm first searches the key in the bottom-two levels for finding an empty slot ($\texttt{slot}_{\texttt{new}}$) via the function **SearchB2Levels**(). The following steps of copying each item are the same as those in Algorithm 5.

ALGORITHM 6: Shrink ()

```
//Rehash each item in the interim level, i.e., the old top level
for each slot_old with <key, value> in IL
    //Search an empty slot in the bottom-two levels
    slot_new = SearchB2Levels(key)
    //Copy the key-value
    slot_new.<key, value> = slot_old.<key, value>
    clflush(slot_new.<key, value>)
    mfence()
    slot_new.token = 1
    clflush(slot_new.token)
    mfence()
    //Delete the item from the old slot
    slot_old.token = 0
    clflush(slot_old.token)
    mfence()
```

5 PERFORMANCE EVALUATION

5.1 Experimental Setup

All our experiments are performed on a Linux server with the 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.

To evaluate the performance of level hashing on persistent memory, we conduct our experiments using Hewlett Packard's Quartz [44, 58], which is a DRAM-based performance emulator for persistent memory. Quartz emulates the latency of persistent memory by injecting software created delays per epoch and limiting the DRAM bandwidth by using DRAM thermal control registers. However, the current implementation of Quartz [44] does not yet support the emulation of write latency in persistent memory. We, hence, emulate the write latency by adding an extra delay after each CLFLUSH, like existing work [28, 32, 37, 51, 59].

The experimental results in PFHT [14] and path hashing [67] demonstrated that PFHT and path hashing significantly outperform other existing hashing schemes, including chained hashing, linear probing [47], hopscotch hashing [22], and cuckoo hashing [45, 54], 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. Due to the lack of the source code of PFHT, we faithfully implement the proposed idea and main components of PFHT based on the descriptions in their paper [14], and use the open-source code [21] of path hashing for comparisons. Since these hashing schemes do not consider the data consistency issue on persistent memory, we implement the persistent BCH, PFHT, and path hashing using our proposed consistency guarantee schemes as presented in Section 3.3 for fair comparisons. Moreover, we also compare the performance of these hashing schemes in DRAM without the need of crash consistency guarantee.

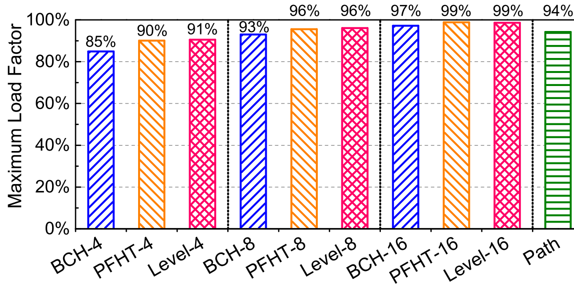


Fig. 10. Maximum load factors of hash tables. *(# in the NAME-# indicates the number of slots per bucket.)*

Since 16-byte key has been widely used in current key-value stores [3, 31, 61], we use the key with the size of 16 bytes and the value whose size is no longer than 15 bytes. Two slots align a cache line (64 bytes) 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 [13], a benchmark for key-value stores, to evaluate the performance of level hashing in multiple mixed workloads and the concurrent performance of the concurrent level hashing. In the following experimental results, each data value is the average of 10-run results.

5.2 Experimental Results

5.2.1 Maximum Load Factor. The maximum load factor is an important metric for hash tables due to directly affecting the number of items that a hash table can store and the hardware cost [17, 34]. To evaluate the maximum load factor, we insert unique string keys into empty BCH, PFHT, path, and level 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 [17, 34]. For PFHT, the 3% space of the total hash table size is used as a stash, following the configuration in the original paper [14]. 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 10 shows that all the four hash tables can achieve over 90% of maximum load factor. Figure 10 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.

In the following experiments, we set four slots per bucket for BCH, PFHT and level hashing, like the existing work [5, 14, 17].

5.2.2 Insertion Latency. To evaluate the insertion latency of different hashing schemes, we insert unique key-value items to empty BCH, PFHT, path, and level 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 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.

Figure 11(a) shows the average latency of each insertion operation in different hash tables in DRAM. Figure 11(b)–(d) show the average insertion latency of different hash tables in persistent

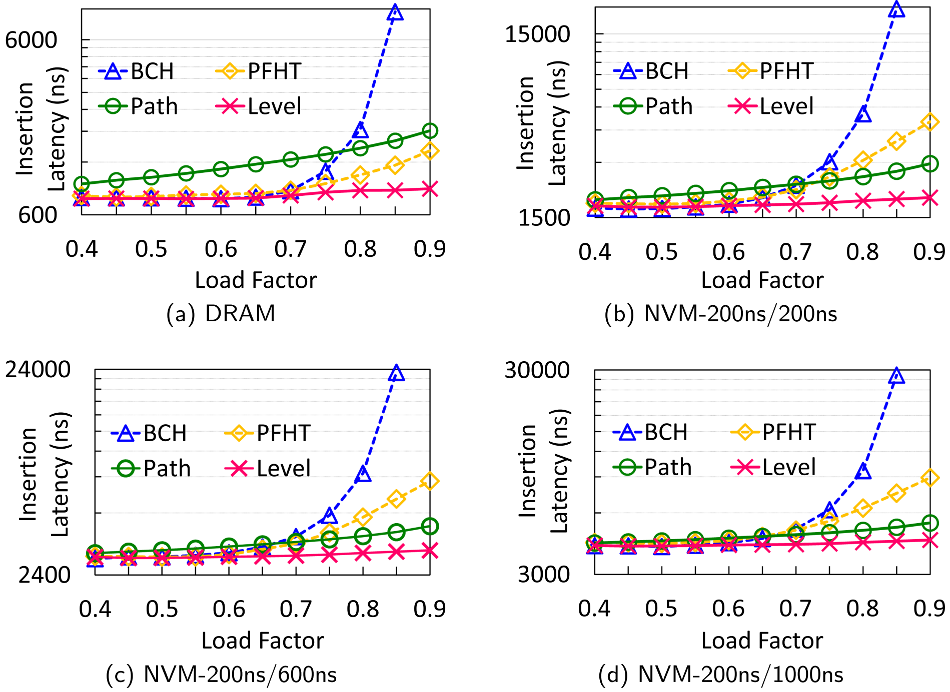


Fig. 11. Insertion latency of different hashing schemes in DRAM and NVM with different read/write latencies.

memory. Compared with the experimental results in Figure 11(a) and (b), 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 11, we observe that when the load factor is smaller than 0.6, the hash collision ratio is low. BCH, PFHT, and level hashing have the similar insertion latency, which is lower than path hashing. The reason is that path hashing needs to probe multiple-level buckets to find an empty location. When the load factor is higher than 0.6, the hash collision ratio significantly increases. Level hashing achieves the lowest insertion latency due to only probing at most two buckets in the bottom level when hash collisions occur in the top level. The insertion latency of BCH increases sharply, due to incurring many evictions to deal with hash collisions. The insertion performance of BCH becomes worse in persistent memory, since the eviction operations in BCH also incur many cache line flushes and memory fences. When the load factor is high, the insertion latency of PFHT increases since many items are inserted in the stash. PFHT uses the chained hash table to manage the key-value items in the stash. An insertion in the stash needs to allocate the node space and revise pointers, resulting in extra writes. The insertion latency of path hashing is higher than that of PFHT in DRAM as shown in Figure 11(a), while becoming lower than that of PFHT in persistent memory as shown in Figure 11(b) for a high load factor (e.g., ≥0.7). This is because 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. From Figure 11(b), we observe when the load factor is larger than 0.8, level hashing reduces the

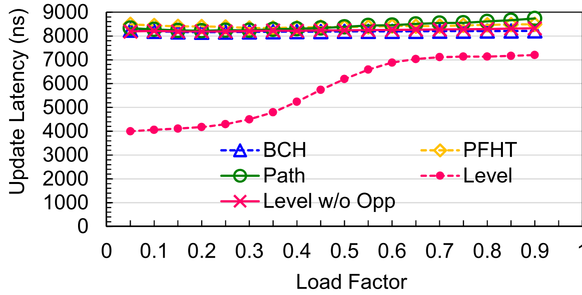


Fig. 12. Average update latency of different hashing schemes in NVM.

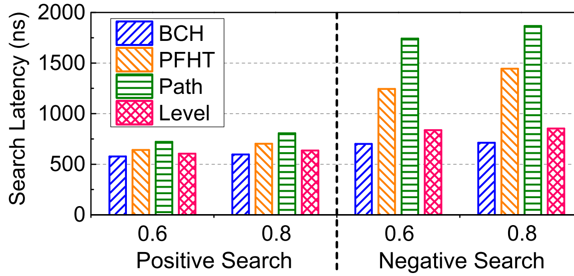


Fig. 13. Average latency of positive and negative searches in level hashing.

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.

5.2.3 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 12, 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 is significantly higher than their insertion latency as shown in Figure 11(c), 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%–52%, i.e., speeding up the updates by 1.2 × –2.1×.

5.2.4 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, it is negative. When hash tables are in two typical load factors, i.e., 0.6 and 0.8 [67], we perform 1 million positive and negative searches, respectively, and measure their average latency, as shown in Figure 13.

We observe that a higher load factor results in a 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 a 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

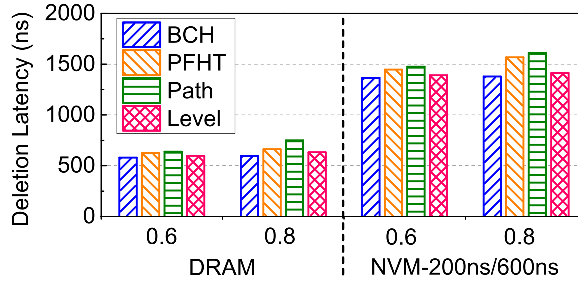


Fig. 14. Average deletion latency of different hashing schemes in DRAM and NVM.

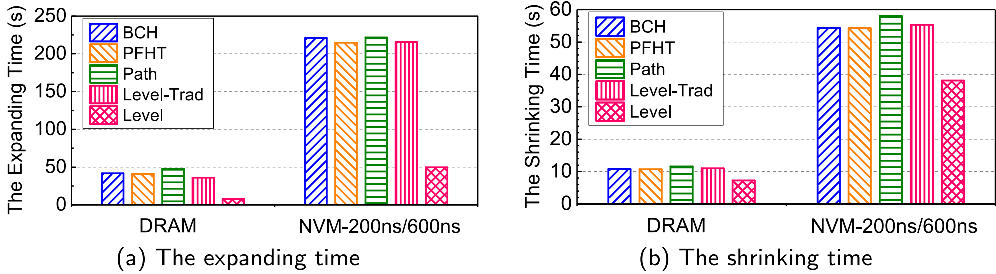


Fig. 15. The expanding and shrinking time of different hashing schemes in DRAM and NVM.

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 where 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 [29], 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.

5.2.5 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.

5.2.6 Expanding and Shrinking Time. We investigate the expanding and shrinking performance of different hashing schemes. To evaluate the expanding performance, we expand 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 10). We measure the total time that different hashing schemes complete the expanding. In order to show the benefit of our proposed in-place expanding scheme, we also evaluate the expanding performance of Level-Trad, which indicates the level hashing using the traditional resizing scheme [46], as shown in Figure 15(a).

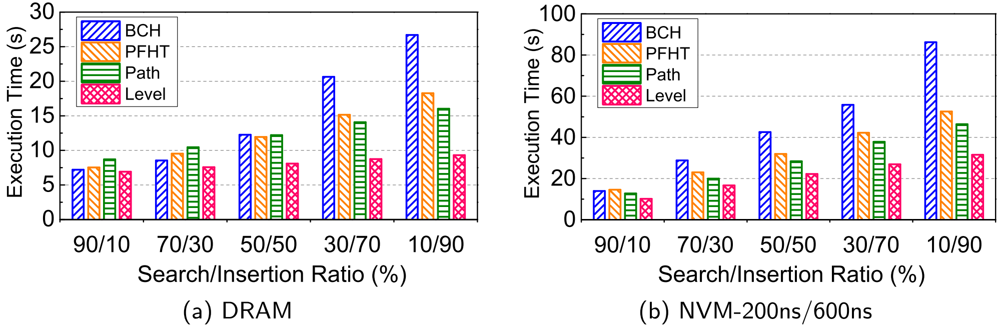


Fig. 16. Execution time of YCSB workloads using different hashing schemes in DRAM and NVM.

We observe that the level hashing reduces the total expanding time by about 76%, i.e., speeding up the expanding by 4.3×, compared with Level-Trad. The reason is that the level hashing by using the in-place expanding 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 expanding time, since they need to rehash all items from the old hash table to the new one.

To evaluate the shrinking performance, we first insert key-value items into different hash tables until their load factors reach 0.85, and then randomly delete items until their load factors reach 0.3. At this time, we shrink these hash tables and measure the total time that different hashing schemes complete the shrinking. The shrinking time of different hash tables is shown in Figure 15(b). In order to show the benefit of our proposed in-place shrinking scheme, we also evaluate the shrinking performance of Level-Trad that indicates the level hashing using the traditional resizing scheme [46].

We observe that the level hashing reduces the total shrinking time by over 31%, i.e., speeding up the shrinking by 1.4×, compared with BCH, PFHT, path hashing, and Level-Trad. The reason is that the level hashing by using the in-place shrinking scheme does not need to rehash the key-value items in the bottom level, thus reducing the number of rehashed items.

5.2.7 Index Performance in the Mixed Workloads. We use YCSB [13] to generate five mixed workloads. Each workload includes a total of 10 million search/insertion requests following the default Zipfian distribution of YCSB. Figure 16 shows the total execution time of different hash tables when executing each of the YCSB mixed workloads in the DRAM and persistent memory.

The total execution time of each hashing scheme increases with the growth of the percentage of the insertion operation due to its increase in the load factor of hash tables and producing higher latency than the search operation, especially in the persistent memory. The execution time of BCH sharply increases with the increase of the insertion ratio, since its insertion latency is much higher than other hashing schemes. PFHT and path hashing have less execution time than BCH in persistent memory, due to causing fewer writes for insertions. Among the four hashing schemes, level hashing has the least execution time under all experimental conditions due to its advantages of efficient insertion and search performance. In most write-intensive workloads (search/insertion: 10%/90%), the level hashing achieves over 2.7×, 1.7×, and 1.5× speedups, respectively, compared with BCH, PFHT, and path hashing.

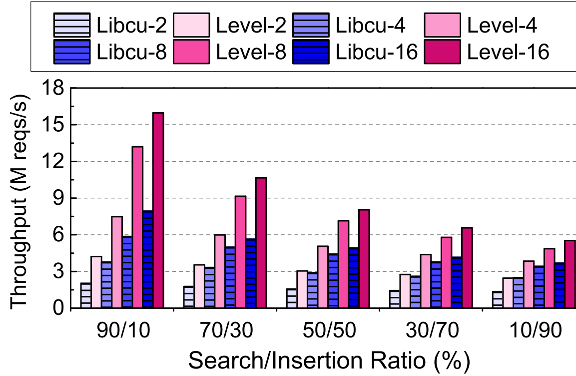


Fig. 17. The concurrent throughputs of level hashing and libcuckoo with 2/4/8/16 threads in DRAM.

5.2.8 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 [34, 35]. We focus on general hashing schemes without special hardware support, and, 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. The experimental results are shown in Figure 17.

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 [34], 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.

6 RELATED WORK

In this section, we discuss the related studies on index structures for NVM and concurrent hashing index structures.

Tree-based Index Structures on NVM. For tree-based index structures, most existing works focus on B-tree [23]. Chen et al. [8] 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. [57] propose the CDDS B-tree that leverages versioning and `CLFLUSH` and `MFENCE` instructions to guarantee data consistency in B-tree. Yang et al. [63] propose the NV-Tree to guarantee the data consistency of only leaf nodes in B^+-tree while relaxing that of internal nodes. The internal nodes can be rebuilt based on leaf nodes on system failures. NV-Tree reduces the number of cache line flushes by only persisting the leaf nodes. Chen et al. [9] propose a write-atomic B-tree (wB^+-Tree) that adds a bitmap in each node of B^+-tree and guarantees consistency via the atomic update of the bitmap. However, wB^+-Tree requires expensive redo logging for node split operations. Oukid et al. [43] 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. [23] propose the log-free failure-atomic shift (FAST) and in-place rebalance (FAIR) algorithms for B^+-tree in persistent memory by tolerating transient

inconsistency. Except B-tree, Lee et al. [32] focus on the radix tree on persistent memory and propose Write Optimal Radix Trees (WORT) that guarantee data consistency by 8-byte atomic writes. Unlike them, our article 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 [14] and path hashing [67, 68], 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 and 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 article 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 [17] 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) [25] provides a chaining-based concurrent hash table using per-bucket fine-grained locking. Libcuckoo [34] 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 [17] 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.

7 CONCLUSION

To efficiently index the data on persistent memory, this article 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 speeds up insertions by 1.4×–3.0×, updates by 1.2×–2.1×, expanding by over 4.3×, and shrinking by over 1.4×, 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 × –2.1×.

REFERENCES

[1] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.

[2] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM J. Emerging Technol. Comput. Syst. (JETC)* 9, 2 (2013), 13.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.

[4] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[5] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. 2016. Horton tables: Fast hash tables for in-memory data-intensive computing. In *USENIX Annual Technical Conference (USENIX ATC)*. 281–294.

[6] John Byers, Jeffrey Considine, and Michael Mitzenmacher. 2003. Simple load balancing for distributed hash tables. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*. 80–88.

[7] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box concurrent data structures for NUMA architectures. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 207–221.

[8] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking database algorithms for phase change memory. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. 21–31.

[9] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[10] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011). 105–117.

[11] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Comput. Surv. (CSUR)* 11, 2 (1979), 121–137.

[12] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. 133–146.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*. 143–154.

[14] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. 2015. 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)*. 18–26.

[15] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. 1984. Implementation techniques for main memory database systems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1–8.

[16] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. 1–15.

[17] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 371–384.

[18] Hui Gao, Jan Friso Groote, and Wim H. Hesselink. 2004. Almost wait-free resizable hashtables. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*.

[19] Hector Garcia-Molina and Kenneth Salem. 1992. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng.* 4, 6 (1992), 509–516.

[20] Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. 2012. Software persistent memory. In *USENIX Annual Technical Conference (USENIX ATC)*. 1–15.

[21] Path Hashing. 2017. Path Hashing: A Write-friendly Hashing Scheme for Non-volatile Memory Systems. Retrieved from https://github.com/Pfzuo/Path-Hashing.

[22] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *Proceedings of the International Symposium on Distributed Computing (DISC)*. 350–364.

[23] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. 187–200.

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

[25] Intel. 2017. Intel Threading Building Blocks. Retrieved from https://www.threadingbuildingblocks.org/.

[26] Intel. 2018. Intel Architecture Instruction Set Extensions Programming Reference. Retrieved from https://software.intel.com/en-us/isa-extensions.

[27] Java. 2018. Java HashMap. Retrieved from http://www.docjar.com/html/api/java/util/HashMap.java.html.

[28] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in write-ahead logging. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 385–398.

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

[30] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 468–479.

[31] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value storage system for cloud data. In *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST)*.

[32] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write optimal radix tree for persistent memory storage systems. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)*. 257–270.

[33] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. 2015. 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)*. 476–488.

[34] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. 1–14

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

[36] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*. 21–35.

[37] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DUDETM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 329–343.

[38] Yujie Liu, Kunlong Zhang, and Michael Spear. 2014. Dynamic-sized nonblocking hash tables. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC)*. 242–251.

[39] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 235–245.

[40] Memcached. 2018. Memcached. Retrieved from https://memcached.org/.

[41] W. Mueller, G. Aichmayr, W. Bergner, E. Erben, T. Hecht, C. Kapteyn, A. Kersch, S. Kudelka, F. Lau, J. Luetzen, A. Orth, J. Nuetzel, T. Schloesser, A. Scholz, U. Schroeder, A. Sieck, A. Spitzer, M. Strasser, P-F. Wang, S. Wege, and R. Weis. 2005. Challenges for the DRAM cell scaling to 40nm. In *IEEE International Electron Devices Meeting (IEDM)*.

[42] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 401–410.

[43] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. 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)*. 371–386.

[44] Hewlett Packard. 2015. Quartz: A DRAM-based performance emulator for NVM. Retrieved from https://github.com/HewlettPackard/quartz.

[45] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *Proceedings of the European Symposium on Algorithms (ESA)*. 1–26.

[46] Nick Piggin. 2008. ddds: "dynamic dynamic data structure" algorithm, for adaptive dcache hash table sizing. Linux kernel mailing list. Retrieved from https://lwn.net/Articles/302132/.

[47] Boris Pittel. 1987. Linear probing: The probable largest search time grows logarithmically with the number of records. *J. Algorithms* 8, 2 (1987), 236–249.

[48] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 24–33.

[49] Redis. 2018. Redis. Retrieved from https://redis.io/.

[50] Stephen M. Rumble, Ankita Kejriwal, and John K. Ousterhout. 2014. Log-structured memory for DRAM-based storage. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)*. 1–16.

[51] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. 2017. Failure-Atomic slotted paging for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 91–104.

[52] Ori Shalev and Nir Shavit. 2006. Split-ordered lists: Lock-free extensible hash tables. *J. ACM* 53, 3 (2006), 379–405.

[53] Julian Shun and Guy E. Blelloch. 2014. Phase-concurrent hash tables for determinism. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 96–107.

[54] Yuanyuan Sun, Yu Hua, Song Jiang, Qiuyu Li, Shunde Cao, and Pengfei Zuo. 2017. SmartCuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC)*. 553–565.

[55] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. 2008. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *International Symposium on Computer Architecture (ISCA)*. 51–62.

[56] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. 2011. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX Annual Technical Conference (USENIX ATC)*. 1–14.

[57] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)*. 5.

[58] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. 2015. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference (Middleware)*. 37–49.

[59] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 91–103.

[60] H-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E. Goodson. 2010. Phase change memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.

[61] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 349–362

[62] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 323–338.

[63] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceeding of the USENIX Conference on File and Storage Technologies (FAST)*. 167–181.

[64] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 209–220.

[65] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. 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.

[66] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 14–23.

[67] Pengfei Zuo and Yu Hua. 2017. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST)*. 1–10.

[68] Pengfei Zuo and Yu Hua. 2018. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Trans. Parallel Distrib. Syst.* 29, 5 (2018), 985–998.

[69] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 461–476.

Received January 2019; accepted March 2019