Journal of Systems Architecture 125 (2022) 102462

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

A write-optimal and concurrent persistent dynamic hashing with radix tree assistance

Xiaomin Zou [a], Fang Wang [a,b,*], Dan Feng [a], Junhao Zhu [c], Renzhi Xiao [a], Nan Su [d]

[a] Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Engineering Research Center of Data Storage Systems and Technology (School of Computer Science & Technology, Huazhong University of Science & Technology), Ministry of Education of China, China
[b] Shenzhen Huazhong University of Science and Technology Research Institute, China
[c] National University of Defense Technology, China
[d] Shandong Massive Information Technology Research Institute, State Key Laboratory of High-end Server & Storage Technology, China

ARTICLE INFO

Keywords:
Hybrid DRAM-NVM memory
Dynamic hashing
Data consistency
Optimistic concurrency control

ABSTRACT

Non-volatile memory (NVM) is expected to coexist with DRAM as a hybrid memory to fully exploit DRAM's low read–write latency and NVM's high density, persistence, and low standby power. However, existing persistent hashing schemes cannot efficiently reap the hybrid-memory benefits, and suffer from a significant performance penalty due to consistency guarantee. In this paper, we present an optimized extendible hashing variant for hybrid DRAM-NVM memory, named OP-HMEH. In our design, hash buckets are persisted in NVM while the directory is placed in DRAM for faster access. We also keep a radix-tree-structured directory in NVM to instantaneously recover the directory in DRAM after system crashes. To reduce consistency guarantee overhead, OP-HMEH designs a cross-KV mechanism to reorganize items which can avoid the use of expensive persist barriers in most cases. Meanwhile, we employ low-overhead structural modification operations schemes to further improve system performance. For concurrency control, the original version uses two lock-based techniques. We then propose an optimistic concurrency strategy that exploits lightweight locking to protect segments and enables lock-free search/operations of directories by leveraging atomic instructions. On real Intel Optane DCPMM, experimental results with YCSB workloads show that our OP-HMEH outperforms the state-of-the-art NVM-based hashing structures by up to 2.18×. The optimized HMEH obtains higher insert performance than the original HMEH.

1. Introduction

The past few years have witnessed the rapid development of emerging non-volatile memory (NVM) devices, such as 3D XPoint [1], phase-change memory (PCM) [2], and spin-transfer torque memory (STTRAM) [3]. Recently, the release of the first commercial NVM product, Intel Optane DC persistent memory, brings NVM to reality [4]. The byte-addressable NVMs are promising candidates for replacing DRAM with disk-like durability and near-DRAM access performance. However, current NVM technologies still suffer asymmetric read–write latencies and limited write endurance [5]. It is a commonly held belief that NVM will not replace DRAM overnight and will instead coexist with DRAM for a foreseeable future [5–8].

The changes in memory features and architectures have rendered legacy indexing structures inefficient because they ignore data consistency and do not fully exploit the byte-addressable properties of NVM. Therefore, building high-performance indexing structures that are efficiently adapted to NVM is critical for future storage systems. A large body of prior research has been done to improve tree-based indexing structures for the systems equipped with NVM [6,7,9–12]. Hashing indexing structures are also widely used in applications due to their constant time complexity, i.e., O (1), for point accesses, which are superior to tree-based structures. Recently, several hashing variants for NVM-oriented memory have been proposed, such as PFHT [13], path hashing [14], level hashing [15,16], CCEH [17], Dash [18], and HDNH [19].

However, despite various optimization on the properties of NVM, we demonstrate that existing hashing schemes for NVM all have two shortcomings. First, their designs of hashing structures are imperfect for NVM. According to the way of resizing, hash-based structures are divided into static hashing structures and dynamic hashing structures. Each of them has its own merits. Most previous persistent hashing schemes focus on static hashing structures since they can provide fast lookup responses [13–15]. However, their resizing operations are more

* Corresponding author.
E-mail address: wangfang@hust.edu.cn (F. Wang).

https://doi.org/10.1016/j.sysarc.2022.102462
Received 29 August 2021; Received in revised form 23 January 2022; Accepted 8 March 2022
Available online 19 March 2022

time-consuming since they need to create a bigger or smaller hash table, typically twice or half as large, and rehash all items in the old table into the new table. Meanwhile, during the resizing, all foreground user query requests will be blocked. Considering that NVM has higher latency and lower endurance, this resizing way drastically degrades application performance and even exacerbates the wear-out problems.

Unlike static hashing structures, dynamic hashing structures [20,21] can grow and shrink according to the data size which are widely applied in file systems and database systems [22–24]. Extendible hashing is a typical dynamic hashing that induces a directory to organize buckets, thus it can dynamically add or delete buckets rather than resizing the whole hash table. Dash [18] and CCEH [17] both employ extendible hashing structures to implement cost-efficient resizing. However, the use of the directory leads to a disadvantage that every request requires at least two accesses to index the data. This extra directory access exposes the high read latency of NVM in the critical path, which prolongs the execution time of each operation.

Second, recent consistency guarantee mechanisms are costly that greatly affect the system performance. Since NVM is directly accessed through a memory bus, partial or reordering write to NVM might lead to inconsistent issues after the system failures. To guarantee crash consistency, most existing work employs persist barriers (memory fence and cache line flush instructions) to flush data from volatile CPU caches to NVM in the desired order. However, these persist barriers are proved to cause performance degradation [7,12]. Our evaluation shows that the throughput of several NVM-optimized hashing schemes increases by 20.3%–29.1% without persistent barriers.

In this paper, we present HMEH, a write-optimal extendible hashing that fully leverages the complementary advantages of hybrid DRAM-NVM memory and significantly mitigates the overhead of data consistency. HMEH stores key–value items in NVM for directly persisting and places the flat-structured directory in DRAM to cover the shortage of NVM-based extendible hashing. However, the flat-structured directory will be lost after system failures or normal shutdowns. To resolve this problem, HMEH maintains a radix-tree-structured directory in NVM. Since updates of the radix tree do not incur extra NVM writes and are easy to guarantee consistency, it only incurs negligible overhead but realizes a fast recovery.

To reduce the overhead of consistency guarantee, HMEH leverages a cross-KV mechanism to avoid using expensive persist barriers. Specifically, it rearranges the key and value of the item into multiple failure-atomic 8-byte slices and writes these slices back to NVM by natural evictions. After system crashes, we can check whether the key–value item is correctly persisted to NVM by the hash value of the key. Furthermore, HMEH exploits low-overhead structure modification operations to improve performance. First, when doubling the flat-structured directory, we employ a lazy-migrating doubling scheme that amortizes the migration of directory entries to future queries, thus reducing the blocking time caused by directory doubling. Second, to reduce the persistence overhead of segment splits, we leverage the delayed flush method which only persists the split segment when a new node of the radix-tree-structured directory is created.

Since modern processors support an increasing number of threads, we must make our HMEH scalable to fully utilize the hardware resources. To control concurrent queries in a thread-safe way, the original HMEH exploits two lock-based schemes. However, the lock maintenance overhead grows with the increase of threads and hurts the scalability. To alleviate this problem, we leverage an optimistic concurrency control mechanism to optimize concurrent HMEH. Specifically, insert operations proceed with a lightweight locking that is implemented by a version number and atomic primitives. Search operations can be performed without holding any locks by using the version number to detect conflicts. Furthermore, we exploit atomic primitives and a lazy-migrating mechanism to implement lock-free FS-directory doubling.

In summary, the main contributions of this paper are:

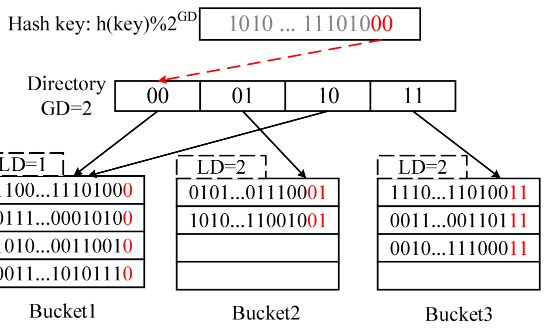


Fig. 1. Extendible hashing structure.

- We design our HMEH consisting of a flat-structured directory in DRAM and hash table in NVM to fully exploit the advantages of hybrid memory. To rebuild the flat-structured directory upon recovery, we also keep a radix-tree-structured directory in NVM to record its updates.
- We propose a cross-KV mechanism that reorganizes the storage form of key–value items as multiple 8-byte atomic cross-KV blocks. So that we can use the hash key to verify whether the item is completely written to NVM, ensuring crash consistency without persist barriers.
- We propose two low-overhead structure modification operations, i.e., we employ a lazy-migrating scheme to decrease the execution time of directory doubling, and propose delayed flush to mitigate the persistence overhead when splitting segments.
- We implement HMEH and optimize the concurrent HMEH (i.e., OP-HMEH). We use atomic primitives and lightweight synchronization techniques to coordinate accesses of concurrent threads, whose correctness is ensured with low overhead. Using Intel Optane DCPMM, our experimental results show that OP-HMEH obtains up to 1.46×/1.73×/2.18× speedup than the state-of-the-art work, HDNH/Dash/CCEH under YCSB workloads.

The rest of this paper is organized as follows. Section 2 describes the background and related work. Section 3 presents the detailed designs of HMEH. In Section 4, we introduce the optimistic concurrency of optimized HMEH and describe the implementation details. The performance evaluation is shown in Section 5. Finally, Section 6 concludes this paper.

2. Background and motivation

2.1. Extendible hashing

Extendible hashing is a dynamic hashing technique optimized for time-sensitive applications, which can dynamically allocate and deallocate hash buckets on demand [20]. Fig. 1 shows the structure of extendible hashing, it uses a resizable array (called the directory) to index multiple buckets, and each bucket stores a fixed number of items.

The directory has 2^{GD} entries, where GD is the global depth of the directory and determines the maximum number of buckets. A suffix corresponding to the trailing GD bits of the hash key is used to index the directory. To improve memory efficiency, the number of buckets does not need to be the same as the number of directory entries. Therefore, each bucket also keeps a local depth (LD) to indicate the number of the common bits in the bucket. When a bucket is only pointed by one directory entry, LD equals GD. If n directory entries point to a bucket, LD = GD-$\log_2$n. Taking an example in Fig. 1, bucket1 is pointed by two directory entries and GD is 2, thus the LD of bucket1 is 1.

When a bucket has no empty slots, extendible hashing will split it into two new buckets and copy the items from the old bucket to the new ones. As Fig. 1 illustrates, to insert a new item into bucket1, two new buckets should be created to replace bucket1. Since the LD of bucket1 is smaller than the GD, it directly modifies directory entry00/10 to point to the new split buckets. After finishing the split, we increment the LD of the new buckets to 2. However, if bucket3, whose LD is equal to GD is full, we first need to double the directory to store the pointer of the new buckets, including creating a double-sized directory and migrating all entries in the old directory to the new one. Then, we update the directory pointer to point to the new one and increase the GD by one. Finally, we split bucket3 as what we do to bucket1.

Compared to static hashing schemes, extendible hashing is more flexible since it reuses most buckets and only splits one bucket for each expansion. Therefore, when running multiple threads, its resizing will not block other concurrent accesses for a long time, which significantly improves the scalability. However, it has shortness that searching an item requires at least two accesses due to the introduction of an indirect directory level.

2.2. Hashing index structure in NVM

The hash-based structures support constant-complexity point query operations, widely used in key–value stores [25–27] and main memory databases [28–30]. Traditional hashing schemes are designed for DRAM or disk that mainly focus on dealing with hash collision and improving indexing performance. They do not consider the properties of NVM, thus becoming inefficient in NVM. Recently, several hashing-based structures have been proposed to efficiently adapt to NVM [13–15,17,18].

PFHT [13] is a PCM-friendly cuckoo hashing variant that only allows one cuckoo displacement to avoid cascading NVM writes. To improve the load factor, PFHT designs a stash to store conflicting key–value items. However, the stash is a linked-list structure which increases the search latency. Path hashing [14] is a write-friendly hashing for NVM that logically organizes storage cells as an inverted binary tree. The leaf nodes are addressable by hash functions, and other nodes in the same path are used to store the conflicting items. PFHT and path hashing are devoted to reducing NVM writes but at cost of decreasing search performance, thus they cannot provide a constant-level lookup. More importantly, neither of them takes data consistency issues into account.

Unlike PFHT and path hashing, level hashing achieves constant lookup time and provides a log-free consistency guarantee [15]. It is a sharing two-level hash table where the top level is addressable for items and the bottom level is used to deal with hash collisions. When level hashing needs to resize, the items in the old bottom level are rehashed to a new 4× larger hash table and the old top level is reused as the new bottom level. While the in-place resizing of level hashing only needs to rehash 1/3 table instead of the entire table. However, the rehashing overhead of level hashing is similar to other hashing schemes [17].

HDNH [19] stores a two-level hashing structure in NVM whose rehashing operations are performed by background threads without blocking concurrent queries. Since Optane DCPMM exhibits 3x longer read latency than DRAM, HDNH puts many efforts to improving read performance. First, it places the index metadata (i.e., fingerprints) in DRAM to reduce unnecessary NVM accesses. Second, HDNH keeps a hot table in DRAM to speed up the search for hot items. However, these structures also cause extra metadata maintenance overhead and sacrifice write performance.

The mentioned NVM-based hashing schemes are based on static hashing schemes. CCEH [17] and Dash [18] are variants of dynamical hashing that can split and merge hash buckets as needed. They both build a segment level between the directory and buckets. The bucket of Dash is set to 256 bytes while CCEH uses 64-byte buckets. To speed up search requests, Dash maintains a fingerprint for every item to avoid bucket probing. For high space utilization, it designs a load balancing strategy to postpone segment splits.

The main disadvantage of dynamic hashing is that every insertion or search needs extra access to the directory. Due to the randomization of hash functions, the accesses to the directory may incur many cache misses, leading to a performance penalty. To measure its overhead, we use the PAPI library [31] to test the LLC cache miss rate of the directory in CCEH. As we increase the number of inserted items from 16 million to 256 million, the directory size of CCEH grows from 1MB to 20MB. In the meanwhile, the cache miss ratio of the directory increases, even up to 98%. Considering that real NVM hardware has higher read latency, the extra directory access may drastically diminish the system performance.

2.3. Data consistency for hashing schemes in NVM

NVM Systems today must support Asynchronous DRAM Refresh (ADR) that ensures the data is persistent on memory controller's write pending queue and NVM [32]. By default, the CPU caches are not protected by ADR and are still volatile. To guarantee data consistency between volatile CPU caches and NVM, it is necessary to ensure the ordering of memory writes [6,9,11]. However, memory writes may be reordered by the CPU and memory controller for better performance. Hence, we have to use persist barriers to form ordered memory writes as existing schemes [10,11,15], including memory fence and cache line flush instructions (short for MFENCE and CLFLUSH). Since the atomic write of the CPU is only 8 bytes, the updates with larger sizes may be partially written after unexpected system failures [6,7,11,12]. Existing work exploits logging or copy-on-write (CoW) to guarantee the atomicity of data larger than 8 bytes [9,33,34].

In the third-generation Intel Xeon Scalable Processors, ADR could be further extended, e.g., enhanced ADR (eADR), to provide cache-level persistence [35]. With eADR, programmers do not need to use CLFLUSH instructions since the hardware flushes the data automatically, but MFENCE instructions are still required to guarantee the correctness of write ordering. Although eADR simplifies NVM programming and reduces the persistence overhead, it comes at the cost of non-standard extensions, high costs, and environment-unfriendly batteries [36]. Therefore, we expect that NVM systems with the standard ADR support are more commonly used in the foreseeable future. Ensuring data consistency is a fundamental requirement in NVM systems.

In NVM-based hashing schemes, the consistency guarantee is an essential part without which hashing structures cannot normally work on NVM. Generally, when inserting a new key–value item into a bucket, we first store the value and next the key. However, writes to NVM may be reordered. Hence, we cannot ensure the item is completely written by checking the validity of the key since the key might have been written back while the value is not. Previous research exploits a sequence of persist barriers to ensure data consistency [15,17,18]. For example, they write the value first, call MFENCE, store the key, and then call CLFLUSH. This ordering ensures that the key is not written to NVM ahead of the value. Therefore, after a system failure, they can identify the partially written items if the keys are not valid for the hash bucket.

However, persist barriers are expensive and their overhead is proportional to the amount of NVM writes. To quantify their cost, we use the random integer workload to measure the average insertion throughputs of five hashing schemes with and without persist barriers (referred as w/ persist barriers and w/o persist barriers): (a) HDNH [19], a state-of-the-art hashing scheme for hybrid DRAM-NVM memory, (b) Dash [18], a recently proposed dynamic hashing for NVM, (c) CCEH [17], a NVM-based extendible hashing, (d) level hashing [15], a static hashing for NVM, (e) linear probing [37], and (f) cuckoo hashing [38]. The details of the experimental setup are presented in Section 5.1. As shown in Fig. 2, without persist barriers, the throughputs of these hashing schemes are improved by 20.3% to 29.1%. These persist barriers significantly deteriorate system performance. Therefore, it is important to reduce the number of persist barriers.

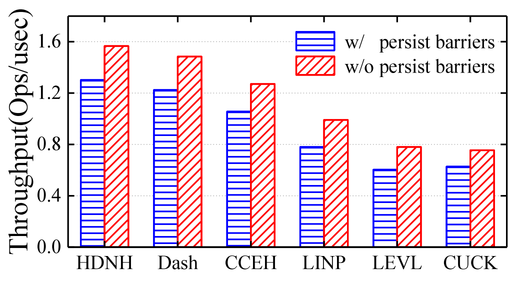


Fig. 2. The insertion throughputs with and without persist barriers.

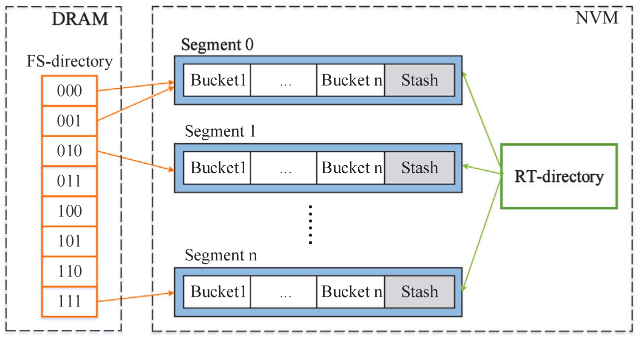


Fig. 3. Architecture of HMEH.

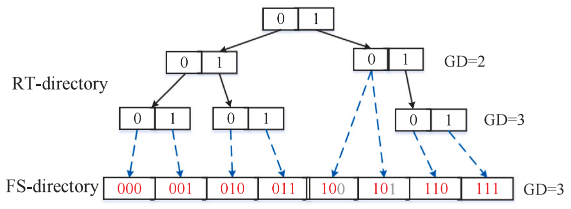


Fig. 4. The relationship of two directories.

2.4. Concurrency control

As modern processors are being scaled to lots of cores, Multi-threaded concurrency is a key issue to enhance program performance. Theoretically, the hashing structure is efficient in a concurrent environment. Because concurrent threads generally access different buckets of the hash table, hence, they can be running in parallel with low interference overhead. However, concurrency control between common operations and the resizing of hash table is difficult, since resizing incurs many extra operations and blocks all concurrent accesses until it is completed.

The simplest synchronization mechanism is probably locking, i.e., in a blocking way. The ConcurrentHashMap of Java concurrency.utils library [39] is a typical work that applies a fixed number of locks, each of which protects a subset of the hash buckets. Level hashing [15] uses fine-grained locks to guard each slot. Before accessing the slot, a thread should acquire the slot lock and release it after finishing the operation. However, during resizing, these two schemes both need to hold all locks, thus blocking all concurrent accesses of other threads and significantly diminishing system performance.

To reduce the heavy-weight locking overhead, several previous work proposes optimistic concurrency mechanisms [40–42]. For example, OLFIT [41] uses the per-node lock to serialize updates to the same node and keeps a per-node version number which is incremented once the node is updated. Before and after reading a node, OLFIT read the version number and checks if they match. If the version number is modified, OLFIT retries the lookup. Unlike conventional lock-based techniques that always update the lock variable at the start and end of operations, the read of optimistic concurrency mechanisms is lock-free with version-based retry which can improve the scalability of search.

Compare-and-swap (CAS) is an atomic instruction that is widely used in existing concurrent indexing data structures to improve scalability [43–45]. The CAS contains three operands: the memory location, the expected old data, and the new value. It first compares the expected old data with the contained data in the memory location, if their values are equal, the processor will atomically replace the contained data with the new value. Otherwise, the expected old data is modified to the contained data. However, CAS instructions cannot guarantee the atomic update of data larger than 8 bytes [46].

3. HMEH design

3.1. Overview of HMEH

We propose HMEH, a write optimal and flexible extendible hashing for hybrid DRAM-NVM memory. Fig. 3 shows the architecture of HMEH. Similar to previous work [17,18], we apply a three-level structure that uses a directory to index segments, and each of them consists of multiple buckets and a stash for colliding items. We place the flat-structured directory (FS-directory) in high-speed DRAM for fast access and store segments in NVM for efficient persistence. DRAM-based FS-directory offers two benefits: First, it moves the slow directory accesses in NVM out of the critical path which significantly speeds up the segment indexing. Second, we do not require to guarantee its crash consistency.

However, after a system crash or normal shutdown, the FS-directory in volatile DRAM will be lost. To address this problem, we design a secondary directory in NVM to rebuild the FS-directory upon recovery. The NVM directory is organized as the radix tree structure (denoted as RT-directory) for two reasons: (1) The radix tree is determined by the prefix of the inserted key which coincides with segment indexes (the MSB bits of hash keys). The height of its leaf node can effectively indicate the local depth, which enables fast recovery. (2) The characteristics of the radix tree can be efficiently utilized in NVM. The resizing of the radix tree directly adds a new node without tree rebalancing operations, which only results in an 8-byte update operation. Thus, we can easily use persist barriers to guarantee data consistency without logging or CoW.

For all foreground queries, the FS-directory is used to index segments, while the RT-directory is accessed upon recovery. Fig. 4 shows the corresponding relationship between FS-directory and RT-directory. We can leverage the depth and leaf nodes of the RT-directory to recover the FS-directory. For high CPU cache efficiency, we set the size of the RT-directory node as a cache line. There is a doubt whether the maintenance of the RT-directory leads to high overhead. Only when segment splits, we require to update the corresponding entries in its RT-directory node. To further reduce the overhead, each segment stores the pointer to the corresponding RT-directory entry so that it can quickly index and update the RT-directory entry without traversing multiple non-leaf nodes. As a result, the RT-directory only causes negligible overhead but achieves an instantaneous recovery as experimental results show in Sections Section 5.2.

3.2. Cross-KV mechanism

As described in Section 2.3, to guarantee data consistency when inserting an item, we must use persist barriers to constraint the ordering of key and value, which has been proved to be a major reason of performance degradation. Therefore, we propose a novel cross-KV strategy to bypass persist barriers in most situations. Its basic idea is to bind key

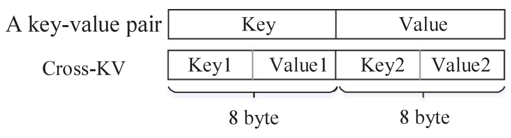


Fig. 5. The storage structure of cross-KV.

and value so that the key will not be written to NVM before the value, thereby avoiding persist barriers.

We split the key and value of an item into several pieces, and then alternately store the key and value pieces as 8-byte combined blocks since modern processors support 8-byte atomic write. Fig. 5 shows the storage structure of cross-KV. Suppose the sizes of key and value are both 8 bytes. We first divide key (value) into 4-byte key1 and key2 (value1 and value2). Next, we combine key1 and value1 (key2 and value2) as 8-byte cross-KV blocks. In this way, we bind the key and value together, thus we can judge whether the value is completely written back to NVM by the integrity of the key. Specifically, if a power loss or system crash occurs during writing an item, we fetch out the key from cross-KVs and recalculate its hash key to check whether the same segment and bucket can be indexed. If so, the key and value are both valid. Otherwise, they are partially written and will be discarded.

However, there is a special case that cross-KV fails to guarantee data consistency. That is, the partially written key are still hashed into the same segment and bucket where the complete key is located, thus the key cannot prove that the value is correctly persisted to NVM. To address this problem, before inserting an item, we form all possible partially written keys that may be produced by cross-KVs. Then we calculate the hash values of these keys to check if they can index the same position where the item will be stored. If one of them can, we insert the target item by persist barriers. In practice, the probability of this case is extremely low, so the cross-KV mechanism is still efficient.

Though HMEH leverages a unique cross-KV structure to avoid the overhead of persist barriers in most cases, it also sacrifices a little performance. First, when searching a key, we require to read the entire item, unlike other hashing indexes that only need to read the key. However, this read overhead can be ignored since we employ cacheline-sized buckets and a single access can prefetch multiple cross-KVs in the same item. Second, we require to check the above special case. Fortunately, our evaluation shows the impact of calculation overhead on performance does not exceed 1%, which is much less than the overhead of persist barriers. For larger or variable-length items, like previous research [15,47], we place key–value pairs outside the hash table to reduce the resizing overhead. We store their pointers and the short summary of the key in the hash table with the cross-KV mechanism.

3.3. Low-overhead structural modification operations

In hashing structure, dynamic insertion and deletion might cause structural modification operations (SMOs, e.g., segment split and directory doubling). However, SMOs have high overhead and degrade system performance since they often incur a large number of extra data movements. To address this problem, we optimize the two types of SMOs individually, which are described below.

lazy-migrating FS-directory doubling. Existing extendible hashing variants, such as CCEH [17], Dash [18], and the original HMEH [48], exploit traditional directory doubling that includes three steps: (1) creates a new double-sized directory, (2) iteratively migrates all the entries in the old directory to the new one, and (3) atomically updates the directory address to the new directory. Among them, step 2 is the most time-consuming since it includes a large number of data movements and blocks other concurrent operations. Different from existing schemes that perform the migration of directory entries during directory doubling, we leverage a lazy-migrating doubling scheme that amortizes the migration process to subsequent operations to reduce the blocking time.

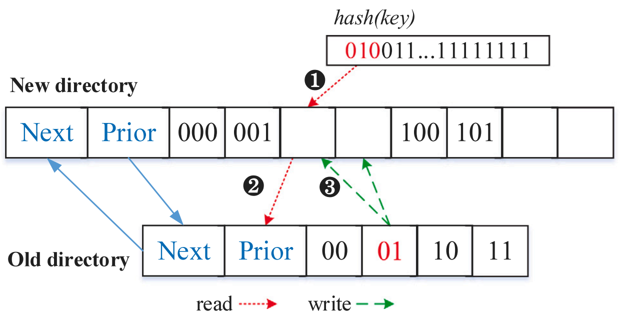


Fig. 6. Fs-directory doubeling.

We slightly modify the structure of FS-directory as shown in Fig. 6. We store two pointers: *NEXT* points to the newly created FS-directory and *Prior* points to the old-versioned one. When doubling FS-directory, we first allocate a new bigger directory. Then we modify *Next* of the old FS-directory to point to the new one and update *Prior* of the new FS-directory to point to the old one. Second, the thread atomically changes the FS-directory pointer to point to the new FS-directory and increments the global depth. Finally, the migration of old directory entries is amortized over future queries. Fig. 6 shows the amortization process: (1) if a thread accesses an empty entry, (2) it searches the corresponding entry in the old directory by the *Prior* pointer. (3) Next, it copies the old entry to the new entries.

The low-overhead Segment Split. In terms of slower NVM, segment split not only causes many NVM writes but also needs to be performed in a failure-atomic way. To reduce NVM writes and mitigate the SMOs overhead, we design a segment split scheme with low persistence overhead.

We use an example of Fig. 7 to elaborate on the process of segment split. Since the FS-directory doubling is described above, Fig. 7 only shows the updates of the RT-directory. Suppose we inset an item to S1, but S1 has no empty slot and its local depth is equal to the global depth. We first increase the size of the RT-directory. As Fig. 7 shows, we allocate a new RT-directory node and the GD of new node is 4. Then we create a new segment (S4) and rehash items of S1 into it. The migrated items in S1 are not deleted since they become invalid with the modification of LD and subsequent inserted items will overwrite them directly. In the next step shown in Fig. 7, we use persist barriers to ensure the ordering of the updates to survive system failures. That is, (1) we set the entries of the new RT-directory node to point to S1 and S4, and update LD of S4 to 3. (2) We change entry 00 of node 1 to point to node 2. (3) We update LD of S1 to 3. Moreover, we also support segment merge which is an inverse process of segment split.

Traditionally, to identify invalid items in case of system failures, we also need to write back the new split segment to NVM with flush instructions. Our cross-KV can distinguish partially written items upon recovery Thus, to alleviate the overhead of data persistence, we use a delayed flush method that leverages normal cache evictions to write back the new split segment. Meanwhile, we leverage unique RT-directory structure to flush segments regularly, which prevents certain cache lines from residing in the cache for a long time.

Specifically, the simple cache evictions may lead to inserted data loss. For example, item t1 is stored in segment s1, and it is rehashed to segment s2 because of s1's split. Then another item t2 is inserted to s1 and overwrites t1. If a system failure occurs before r1 in s2 is written back to NVM, we cannot find t1 anymore. To resolve this problem, we make some changes: when a segment split incurs the creation of a new RT-directory node, we write back this segment to NVM directly and make it non-addressable. Next, we create two new segments and

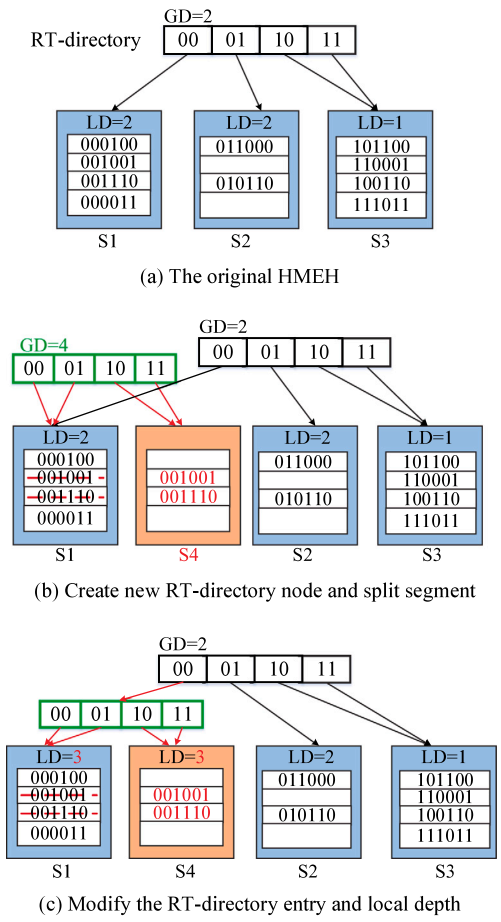


Fig. 7. Examples of failure-atomic segment split.

rehash all items of the old segment into them. By doing so, each RT-directory node has a non-addressable segment, and we can recover the data from it after a system failure. Different from previous schemes flushing each new segment, in our way, on average we split 8 segments but use instructions to flush one segment since an RT-directory node contains eight segment addresses. This method mitigates the cost of data persistence and ensures data consistency.

3.4. Improvement of load factor

The load factor is another essential parameter for hashing structures in memory and caches with limited space, i.e., a higher load factor means more items can be stored. However, the introduction of the segment layer leads to a low load factor. Because no matter which bucket in the segment overflows, we have to split the entire segment to resolve hash collisions even if there are many free slots in other buckets of the old segment.

Traditional hashing schemes improve the maximum load factor by efficiently dealing with hash collisions. For example, chained hashing [49] stores colliding items in linked lists. However, it requires frequent memory allocation and pointer access, resulting in low CPU cache efficiency. 2-choice hashing [50] applies two hash functions to compute two positions and inserts the target item into the empty one. However, it cannot improve the load factor well. Cuckoo hashing [38] exploits several hash functions and allows multiple cuckoo evictions, but it suffers from cascading writes which incur high insertion latency. Linear probing [37] scans the following buckets until it finds empty slots to store conflicting items. Thus, it is cache-friendly and can sequentially access items.

To address the low load factor issue, CCEH allows probing four buckets to store colliding items. However, our evaluation shows that its maximal load factor is lower than 45%. Though CCEH can improve the load factor by increasing the probing distance, it is not sufficient to achieve a high load factor to use this way alone, as shown in Section 5.2.3. Therefore, based on linear probing, we also exploit stash schemes. Specifically, we still set the default probing distance to 4 buckets since Intel Optane DCPMM is accessed by 256-byte block granularity [51]. The stash is array-structured secondary storage which is non-addressable and used to store colliding items. Every segment has a stash and all buckets in this segment share this stash. When a hash collision occurs, we first probe the following 4 buckets to find a proper slot, and if fail, we insert the target item to stash. Therefore, we can obtain a higher load factor in a simple but efficient way.

3.5. Recovery of two directories

In this section, we describe the recovery of two directories after a normal shutdown and system crash.

Recovery after a normal shutdown. In the case of a normal shutdown, HMEH flushes the RT-directory to NVM and then stores a flag to indicate a normal shutdown. When rebooting, HMEH only reads the RT-directory and rebuilds the FS-directory in DRAM. Since the segment addresses are stored in the leaf nodes of the RT-directory, we do a breadth-first search for RT-directory to retain the leaf nodes. Then we get the global depth (GD), the local depth (LD), and the starting position of each segment corresponding to the FS-directory. As Fig. 4 shows, with GD and LD, we can calculate the reference count which indicates the number of contiguous entries pointing to the same segment. At last, we rebuild the entries of the FS-directory according to starting positions and reference counts.

Recovery after a system crash. For a system crash, HMEH first recovers the RT-directory. Since data consistency problems only happen in leaf nodes of the RT-directory, we just need to recover leaf nodes. We exploit the global depth of the RT-directory and the local depth of segments to check the entry consistency. Specifically, we first access the leftmost entry to obtain LD and GD and calculate the reference count. Second, we compare the LD of the first entry with that of the following entries in the same reference count sequentially. If the LD of the latter entry is different, we make this entry equal to the first entry. Then we iteratively detect the inconsistencies of leaf nodes until the recovery of the RT-directory is completed. At last, we rebuild the FS-directory from the RT-directory as mentioned before.

4. Optimized concurrent HMEH

In the case of common operations (i.e., insertion, search, deletion), hashing structures can achieve high scalability since concurrent threads generally access different parts of the hash table. The main challenge is allowing common operations of the hash table to be executed in parallel with resizing (i.e., segment split and FS-directory doubling).

For segment split, CCEH and original HMEH both employ a reader-writer lock, while Dash uses bucket-level locking which will cause high lock overhead since it must acquire all bucket locks before splitting a segment. For directory doubling, they all need to hold a global lock to prevent concurrent threads from updating the directory. However, the global lock naturally increases the latency and block all concurrent queries of other threads. To mitigate lock maintenance overhead, we implement an optimized HMEH (called OP-HMEH) that leverages optimistic locking to protect segments and proposes a lock-free directory doubling.

4.1. Optimistic concurrency in OP-HMEH

We implement a lightweight and optimistic locking for segments by using a version number and the CAS instruction, similar to previous work [40–42]. The 8-byte version number is used as a lock that indicates whether the corresponding segment is occupied, i.e., "odd number" for the occupied segment and "even number" for the free segment. Before updating the segment, the write thread tries the CAS instruction to set the version number from "even number" to "odd number" until success. Then it executes the insertion operation and other threads cannot modify this segment. After finishing the insertion, the thread increments the version number to release the lock. With the version lock, we can detect conflicts by comparing the version numbers before and after reading the target item. Thus, search operations are allowed to proceed without holding the version lock. To search for an item, we first check whether the segment lock is being held (whether the version number is odd) and wait until the lock is released. Then we access the segment to find the target item and read the version number again. If the version number is changed, we require to retry the search since the item might be updated by other concurrent threads.

However, the lock-free search may incur a correctness problem. Suppose a search thread t1 accesses the segment s1 but sleeps before reading the version number of s1. Then, an insertion thread split s1 and migrate kv1 to the new segment. Meanwhile, kv1 in the old s1 is overwritten by another insertion thread. Later, although kv1 exists in the new segment, t1 will miss the target item due to the concurrent moving of other threads. To fix this missing issue, we use a verification-based retry mechanism. After reading the version number of the current segment, we need to check whether the current segment is the target one, i.e., we recompute the location of the target segment and check whether the location matches the current segment. If not, we retry the search operation to avoid the false-negative case. Since the verification overhead is very low and the above case is rare, we can guarantee the correctness of lock-free search with minor overhead.

For all operations of two directories, no lock is taken. Basic operations (i.e., updating directory entries) are inherently thread-safe without any locks since a thread must first acquire the segment lock before modifying directory entries. The update ordering of the RT-directory and FS-directory is fixed, the RT-directory first and then the FS-directory. Thus, the synchronization between two directories is serialized. When resizing the RT-directory, we employ the CAS instruction to guarantee the concurrency correctness since its expansion only results in a single 8-byte update operation. For FS-directory doubling, we create a new double-sized FS-directory and then update the next pointer of the old FS-directory to point to the new one by trying the CAS instruction. Finally, we exploit a lazy-migrating scheme to amortize entries migration over future queries, further reducing the blocking time caused by FS-directory doubling.

4.2. Implementation algorithms

In this subsection, we describe how OP-HMEH is implemented and introduce the algorithms of its basic operations, including insertion, search, and deletion.

Insertion. Algorithm 1 presents the pseudo-code of the insertion operation in OP-HMEH. To insert a new item with <key, value>, we first get the pointer to FS-directory and the global depth, then compare their version numbers to verify they are updated atomically. If their version numbers do not match, we reload them. Next, we compute the hash key and use its prefix bits to index the corresponding FS-directory entry which contains the segment address (Lines 4–6). If the entry is empty, we call **Fillup()** function to migrate this entry from the old FS-directory to this new FS-directory.

Second, we try the CAS instruction to acquire the version lock of the current segment. However, the current segment might not be our target segment and leads to a wrong insertion since it might be split by other

ALGORITHM 1: Insert(key, value)

```
1:  RETRY:
2:  //Compute the locations in directory and segment
3:  check_version (FS_dir_pointer, GD); //Continue if versions match, retry
    otherwise
4:  k = hash(key);
5:  seg_idx = k >> (k_bit - GD);
6:  seg = FS_dir_entries[seg_idx];
7:  //If the entry of FS-directory is null, then help to migrate the old entries
    to the new fs-directory
8:  if seg == null then
9:      seg→Fillup();
10:     goto RETRY;
11: end if
12: //Try CAS to acquire the lock until success
13: repeat
14:     Acqure(segment_version_lock)
15: until a cas successes
16: //If the current segmen is not the target segment, retry
17: if  key_ seg_prefix != seg_prefix then
18:     Release(segment_version_lock);
19:     goto RETRY;
20: end if
21: //When the slot is empty or invalid, insert key and value
22: buc_idx = k % BUCK_NUM; // BUCK_NUM: the number of buckets in
    each segment
23: for each slot in bucket[buck_idx]-bucket[buck_idx+3] do
24:     if  slot.key = empty or slot.key.seg_prefix!= seg_prefix then
25:         slot.<key,value> = Cross-KV(key,value);
26:         Release(segment_version_lock);
27:         return TRUE;
28:     end if
29: end for
30: for for each slot in stash do
31:     if  slot.key = empty or slot.key.seg_prefix!= seg_prefix then
32:         slot.<key,value> = Cross-KV(key,value);
33:         Release(segment_version_lock);
34:         return TRUE;
35:     end if
36: end for
37: //If there is no proper slot, split the segment
38: Split();
39: goto RETRY;
```

concurrent threads after we index it but before we acquire its lock. To guarantee the correctness of insertion, after holding the segment lock, we must verify whether the current segment we access is the target segment. If not, we release the segment lock and retry our insertion.

Third, we find a proper slot (i.e., an empty slot or a slot containing an invalid item caused by lazy deletion) in the target buckets (Lines 22–24). When a proper slot is found, the target item is inserted by cross-KV mechanism. Finally, we release the segment by incrementing the version number. If there is no proper slot in the target buckets, we probe the stash buckets. When the stash is also full, the segment should be split with **Split()** function, as described in Section 3.3.

Search. Algorithm 2 describes the process of the search operation. Similar to the insertion operation, the first step of a search is to check if the version numbers of the pointer to FS-directory and the global depth are equal. Inconsistent version numbers indicate a concurrent thread is doubling the FS-directory, we then read them again. Second, we use the hash value of the key and global depth to select a segment. Meanwhile, if needed, we help to fill up the new FS-directory (Lines 7–9).

Next, we check whether the current segment is our target segment and retry if it is not. Third, we need to verify the target segment is not locked. Once the segment is free, we continue to search the queried key in the target buckets. If neither buckets contain the queried key, we then search the stash buckets. After finishing the search operations, we

ALGORITHM 2: Search(key)

```
1: RETRY:
2: //Compute the locations in directory and segment
3: check_version (FS_dir_pointer, GD);
4: k = hash(key);
5: seg_idx = k>>(k_bit - GD);
6: seg = FS_dir_entries[seg_idx];
7: if seg == null then
8:     seg→Fillup();
9:     goto RETRY;
10: end if
11: if key_ seg_prefix != seg_prefix then
12:     goto RETRY;
13: end if
14: if is_version_lock_odd then
15:     goto RETRY;
16: end if
17: buc_idx = k % BUCK_NUM;
18: for each slot in bucket[buck_idx]-bucket[buck_idx+3] do
19:     if slot.key = key then
20:         if is_version_lock_changed then
21:             goto RETRY;
22:         end if
23:         return slot.value;
24:     end if
25: end for
26: for for each slot in stash do
27:     if slot.key = key then
28:         if is_version_lock_changed then
29:             goto RETRY;
30:         end if
31:         return slot.value;
32:     end if
33: end for
34: return NULL;
```

ALGORITHM 3: Delete(key)

```
1: RETRY:
2: //Search the position of the target item
3: Slot = Search_delete(key);
4: //If the target key cannot be found, return false
5: if slot == NULL then
6:     return FALSE;
7: else
8:     //Delete the target item and reset the flag
9:     slot.cross-kv = empty;
10:     Release(segment_version_lock);
11:     return TURE;
12: end if
```

read the version number again to check whether it is modified by other concurrent threads. If the version number is not changed, we return the result. Otherwise, we redo the search operation. In this way, the implicit conflict can be identified and will not lead to any problem of correctness.

Deletion. Algorithm 3 depicts the deletion operation. When deleting an item, we first call the function **Search_delete()** to find the slot containing the target key. The implementation of **Search_delete()** is similar to **Insert()** in Algorithm 1 while it returns a slot pointer. To guarantee the correctness of deletions, **Search_delete()** also requires to hold the segment lock and check whether it indexes the right segment before searching for the target item. This is because a concurrent thread might split the segment and copy the target item to the new segment during item migration. If we do not verify the segment, we may delete the target item in the old segment and return TURE. However, the duplication of the deleted item still exists in the new segment, resulting in an invalid deletion. Next, if **Search_delete()** returns NULL, the target item does not exist in the hash table. Otherwise, we atomically delete the item by modifying the cross-KVs to empty KV. Finally, we release the version lock of this segment.

Table 1
YCSB workloads.

Workloads	Insert ratio (%)	Search ratio (%)
Load A	100	0
B	50	50
C	0	100

5. Performance evaluation

5.1. Experimental setup

We evaluate the performance of HMEH against the state-of-the-art NVM-based hashing indexes on the Intel Optane DC Persistent Memory Module (DCPMM). All experiments are conducted on a 2-socket, 18-core Linux server (kernel version 5.4.0) equipped with 1.5 TB PMM, 192 GB DRAM, and 24MB Last Level Cache (LLC). We use the ext4-DAX file system and the APP Direct mode of Optane DCPMM [4] to perform all experiments. We use PMDK [52] to manage the NVM space and apply *clwb* for cache line flushes, which is more efficient than *clflush* and *clflushopt*. To avoid the effect of NUMA architecture, all experiments are performed on one CPU socket, like RECIPE [53].

Compared systems. We compare our work with the state-of-the-art NVM-friendly hashing schemes (i.e., HDNH, Dash, CCEH, and level hashing)[1] and typical DRAM-based hashing works (linear hashing and cuckoo hashing). To be fair, we implement persistent linear hashing and cuckoo hashing (referred to as P-LINP and P-CUCK) by using persist barriers.

In our experiments, the initial hash table of every scheme is sized for 2048 key–value items. For LEVL, we optimize the bucket size as a cache line that can leverage high cache efficiency. For other hash tables, we set them with their optimal parameters in their original papers. HDNH uses 256-byte buckets to consist with the granularity of Optane DCPMM. Dash exploits 256-byte buckets and 16 KB segments, each of them has two stash buckets. CCEH employs the default configuration with 64-byte buckets and 16 KB segments. We observe that the original CCEH does not guarantee the atomic update of the global depth and the pointer to the directory. Thus, we also apply the version number for the directory to avoid failures in experiments. Similar to prior work [17], we allow P-CUCK to try 16 evictions before rehashing, and P-LINP performs full-table rehashing when the load factor achieves 95%.

Workloads. We use 160 million random integers as micro-workload to measure the throughputs and latencies of single queries, whose key and value are both set to 8 bytes. To evaluate the concurrent performance, we generate 3 macro-workloads to test the mixed queries from the widely used macro-benchmark YCSB [54], the industry standard for evaluating key–value indexes. The workloads have different proportions of insert and search queries, described in Table 1. For each macro-workload, we first preload the hash table with 80 million items, then run 80 million operations.

5.2. Experimental results and analysis

5.2.1. Sensitivity analysis of HMEH design

To find the optimal configuration of HMEH, we devise several experiments to measure the designs of HMEH. First, we quantify the performance effects with different segment sizes which are varied from

[1] We downloaded the authors' implementations from **https://github.com/DICL/CCEH**, **https://github.com/baotonglu/dash**, and **https://github.com/Pfzuo/Level-Hashing**.

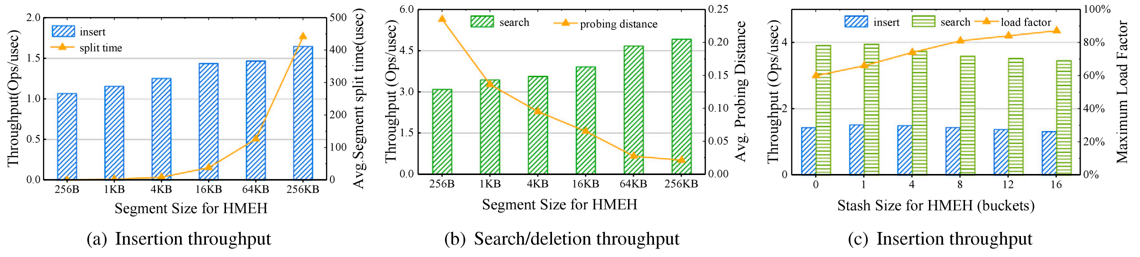


Fig. 8. Throughput of different segment sizes.

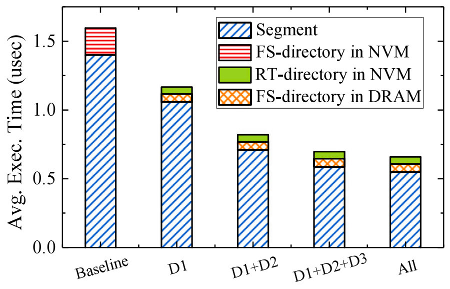


Fig. 9. Insertion Latency of HMEH when adding different designs. (Baseline: persistent extendible hashing; D1: the changes of structure; D2: cross-KV for insertion; D3: delayed flush; All: HMEH that uses D1+D2+D3+stash).

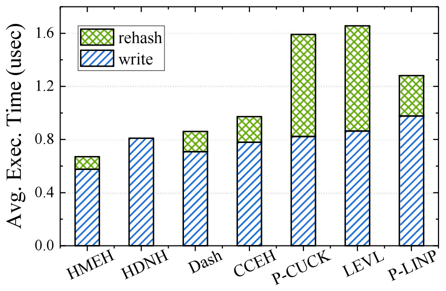


Fig. 10. Insertion Latency of different hashing schemes.

256B to 256 KB. The bucket size is fixed to a cache line, and we allow to probe 4 buckets to resolve hash collisions. To eliminate the effect of stash on the experimental results, we set the size of stash to 0.

Fig. 8(a) illustrates the average throughput of inserting 160 million random items in HMEH with different segment sizes. We see that the insertion throughput improves as the segment size grows. The main reason is small segments incur more frequent segment splits than big segments when inserting an equal number of items, increasing the entire execution time. However, big segment splits require to flush more cache lines into NVM, which increases the latency of a single segment split. As shown in Fig. 8(a), the average latency of segment splits sharply increases and even reaches 441 usec when the segment size is 256 KB. From the experimental results, to balance average insertion throughput and latency of segment split, the reasonable segment size is in the range of 4 KB to 16 KB.

Generally, a search operation only needs to access one bucket to find the target item. However, the methods to address hash collisions increase the number of bucket accesses. In Fig. 8(b), the average probing distance means the average number of extra bucket accesses when an item is searched. It decreases from 0.23 buckets to 0.021 buckets as we increase the segment sizes. This is because HMEH with larger segment size can leverage more bits to determine which bucket is accessed per query, avoiding probing more buckets to find the target items. As Fig. 8(b) shows, due to the decrease of the average probing distance, the average search throughputs increase.

Next, we investigate the performance of HMEH with different stash sizes. The segment size is set to 16 KB, and we vary stash sizes from 1 bucket to 16 buckets. As Fig. 8(c) shows, with the increase of stash size, the search throughputs degrade slowly since we require to access more slots to find target items. We then evaluate the load factors which are closely related to stash configuration. The maximum load factor linearly grows because the bigger stashes can store more colliding items. From the aforesaid observations, the optimal stash size is between 1 bucket and 8 buckets.

5.2.2. Comparative performance

According to the experimental results of HMEH design, the size of the segment is set to 16 KB and the stash has 4 buckets for the rest of the experiments. We first analyze the performance gains brought by the designs of our HMEH (i.e., hybrid structure, cross-KV mechanism, delayed flush, and the stash). We insert 160 million random items and break down the average latency of inserting an item into maintaining directory time (denoted as RT-directory and FS-directory), and other time spent in segment (denoted as segment). We apply persistent extendible hashing as our experimental baseline. As Fig. 9 shows, the design of hybrid memory yields significant performance gains, and the overhead of hybrid directories is smaller than the overhead of one FS-directory in NVM. This is because the operations on FS-directory are executed in DRAM and the resizing of RT-directory in NVM can reuse all old nodes and only allocate a new node, whose overhead can be ignored.

Then, we compare the average insertion latency of our HMEH against those of other persistent hashing schemes. We also break down the average latency of inserting an item into the write time (denoted as write), and the rehashing time (denoted as rehash). Since HDNH uses background threads to execute the rehashing operations, we only exhibit its write time. As shown in Fig. 10, HMEH exhibits the best insertion performance. Compared with HDNH, Dash, CCEH, P-CUCK, LEVL, and P-LINP, we observe that HMEH speeds up the insertions performance by 2.47×. HMEH, Dash, and CCEH show low rehashing overhead because they are dynamic hashing schemes in which rehashing is an incremental operation. LEVL presents the highest rehashing latency. The reason is that level hashing requires to delete all items of the bottom level in the old table via *clwb* during rehashing, unlike other hashing schemes which can simply deallocate old hash tables without extra deletion overhead.

5.2.3. Maximum load factor

To evaluate the maximum load factor, we insert 1 million items into empty HMEH, CCEH, LEVL, HDNH, and Dash to calculate load factors after every insertion, then we pick out the maximum one. P-LINP and P-CUCK do not have a fixed load factor, thus they are not taken into consideration in this experiment. Since HMEH performs segment split when buckets accessed by linear probing and stash have no empty slots, we measure HMEH with different probing distances and different stash sizes. CCEH also exploits linear probing and Dash designs stash buckets for every segment. For a fair comparison, we also evaluate CCEH and Dash in the same way.

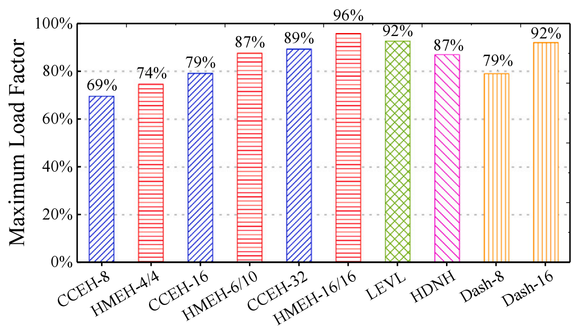


Fig. 11. Maximum load factors of hash tables. (# in the NAME-x/y# indicates the linear probing distance and the size of stash in buckets.)

Fig. 11 shows the experimental results. The maximum load factor of LEVL achieves up to 92% since it uses many effective techniques of load factor improvement, i.e., two-level structure, multi-slot bucket, two hash functions, and cuckoo replacement. The non-volatile table of HDNH is similar to LEVL, but it does not support the cuckoo movement. Thus, the load factor of HDNH only reaches 87%. Dash exhibits a high load factor that achieves up to 92% with 16 stash buckets because it exploits a bucket load balancing technique. As linear probing distance and stash size grow, the max load factors of HMEH increase stably and can reach 96%. Note that HMEH has a higher load factor than CCEH with the same number of sharing buckets. This is because HMEH employs two mechanisms working in different positions of segments and stash can be fully shared by all buckets in a segment. However, with the increase of probing distance and stash size, we require to access more buckets when inserting or searching an item. Thus, we can choose different probing distances and stash sizes to meet the requirements of different cases.

5.2.4. *Concurrent performance*

In this subsection, we evaluate the performance of multi-threaded versions of our HMEH (i.e., original HMEH and OP-HMEH), HDNH, Dash, LEVL, P-LINP, and libcuckoo [55], a state-of-the-art concurrent cuckoo hashing scheme. For libcuckoo, we use its open-source C++ implementation [56].

In the experiments shown in Fig. 12, we run 3 workloads to measure the scalability of the aforementioned hashing schemes under an increasing number of threads, including 2, 4, 8, 16, and 24 threads. For the insert-only workload shown in Fig. 12(a), the throughputs of LEVL, P-LINP, and libcuckoo do not scale with the increase of threads. The main reason is that their rehashing operations incur high latencies and block all insertions from other concurrent threads. Among them, libcuckoo suffers from cascading writes and requires to lock the entire cuckoo path to prevent other threads from accessing the slots in the same cuckoo path. Thus, it causes many locking operations and shows the worst performance.

With dynamic structure, Dash, CCEH, HMEH, and OP-HMEH scale well. HDNH also shows scalable performance by using asynchronous resizing. Though Dash and HDNH leverage fingerprint and bitmap techniques to reduce unnecessary scans of the slots, it also induces extra metadata maintenance overhead. Our OP-HMEH leverages a DRAM-based directory and cross-KV mechanism to significantly reduce the execution time of every insertion. Moreover, it leverages optimistic concurrency control to improve the multi-threaded performance. As Fig. 12(a) shows, OP-HMEH scales to 24 threads and outperforms HDNH/ Dash/ CCEH up to 1.46×/ 1.73×/ 2.18×. Getting benefit from lazy-migration directory doubling and lightweight concurrency control, OP-HMEH improves the insertion throughput by 1.18× than HMEH.

Fig. 12(b)-12(c) present the average throughputs of YCSB mixed workloads and the search-only workload. For the mixed workload, OP-HMEH outperforms HMEH/ HDNH/ Dash/ CCEH/ LEVL/ P-LINP/ libcuckoo by 1.16×/ 1.72×/ 2.1×/ 5.69×/ 4.12×/ 14.1×. For search operation, our HMEH and OP-HMEH have high performance due to the DRAM-based FS-directory. However, when the thread number exceeds 16, their scalability is limited since their bucket probing causes many unnecessary NVM accesses. Different HMEH, HDNH and Dash show near-linear search performance. The main reason is that they use the fingerprint technique to filter mismatched items. Level hashing and libcuckoo scale with the increase of threads since they are the variants of cuckoo hashing which is designed for read-intensive workloads. Note that P-LINP exhibits higher search throughput. This is because P-LINP has no indirect layer (e.g., directory) and stores items in the contiguous memory space, which efficiently reduces memory accesses.

Tail latency is also a critical design issue in storage systems. However, for concurrent static hashing, resizing operations block concurrent accesses and increase the response time because it involves intensive data movements and requires exclusive access to the entire hash table. Hence, we evaluate the tail latency of concurrent insertion. Fig. 13 illustrates its cumulative distribution function (CDF). Since static hashing locks the whole hash table during rehashing which incurs dramatic latency, the CDF graphs of LEVL, LINP, and libcuckoo have several flat regions that indicate the time they take for each rehashing. However, HMEH and CCEH are variants of extendible hashing whose resizing operations only split one segment, significantly reducing blocking time. As Fig. 13 shows, the maximum latency of LEVL is 21.7 s which is much higher than that of our OP-HMEH, 0.92 s.

5.2.5. *Impact of data size*

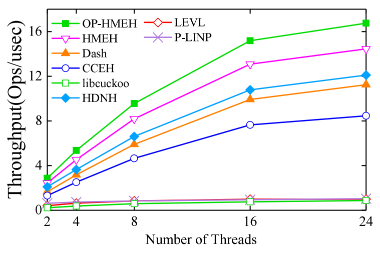
Most hashing structures achieve constant-scale time complexity and their search and insertion performance is insensitive to the dataset sizes. However, every operation of extendible hashing requires extra access to the directory, a doubt is whether the directory limits the scalability as the data size grows. In the experiments shown in Fig. 14, we measure the throughputs of different hashing schemes as the number of key–value items increases from 16 million to 256 million. We first warm up the hash table with 100 million key–value items and then use 16 threads to perform the mixed workload that consists of 50% insert and 50% search operations.

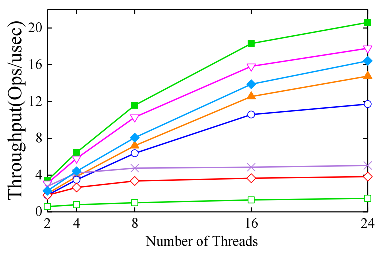
From Fig. 14, we observe that the throughput of CCEH/Dash decreases by 23%/12.1%. The performance degradation is due to the increased directory sizes with the growth of dataset size. As described in Section 2.3, the LLC cache miss rate of the directory rises as the directory size increases. The access to the directory in NVM is in the critical path and significantly increases the query latency. However, HMEH stores the directory in DRAM, thus moving the directory out of the low-speed NVM. Thus, the throughput of our HMEH only decreases slightly. Since OP-HMEH implements lock-free directory doubling, the updates of directory entries can be executed in parallel without inter-thread interference. Therefore, compared to other extendible hashing structures, the FS-directory of OP-HMEH does not limit the scalability of the hash table. Interestingly, the throughputs of static hashing schemes remain unchanged while that of HDNH decreases by 32% as the data size grows. This is because its oversimple hotspot identification becomes inefficient with the increase of data size, resulting in significant performance degradation.

5.2.6. *Negative search throughput*

This subsection evaluates the search throughputs of different hash tables with different ratios of positive/negative searches. The positive search means the target item exists in the hash table, and negative search is the opposite.

Fig. 15 shows the average search throughput under three workloads with different positive/negative search ratios. Level hashing and libcuckoo are both based on cuckoo hashing which is optimized for the read-intensive workloads. Therefore, their search throughput hardly


(a) YCSB-Load A: 100% insert


(b) YCSB-A: 50% insert - 50% search

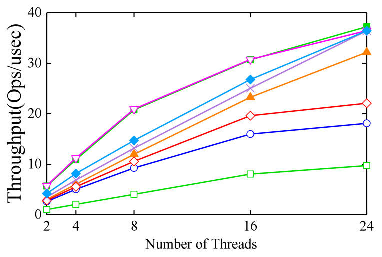

(c) YCSB-C: 100% search

Fig. 12. Scalability on YCSB workloads.

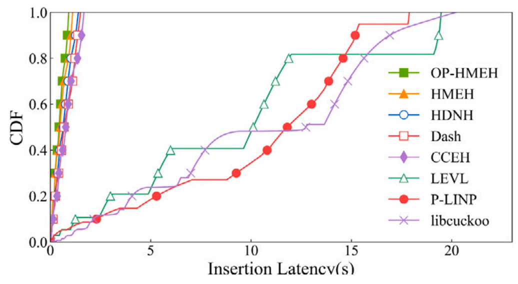


Fig. 13. The CDF of Insertion Latency.

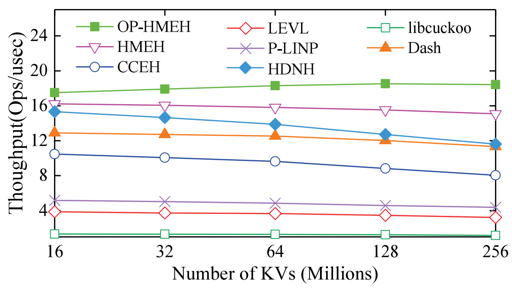


Fig. 14. Impact of dataset size.

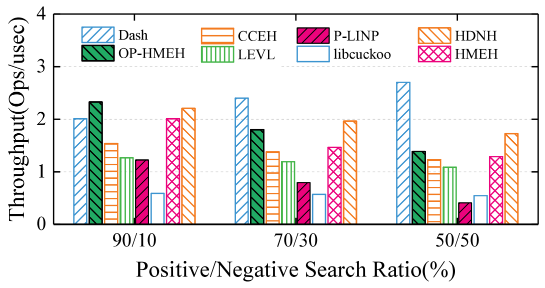


Fig. 15. Average throughput of positive and negative searches.

declines as the negative search ratio increases. However, the performance of P-LINP drops dramatically when workload has more negative searches. The main reason is that P-LINP requires to scan the successive buckets to find the target item.

Interestingly, with the growth of negative search ratio, Dash achieves better search performance instead. Because it uses fingerprints technique which avoids unnecessary NVM accesses and particularly benefits negative search. Although HDNH also keeps fingerprints for all items, it still exhibits lower negative search throughput. The main reason is its read operations first need to search the hot table and then the hash table to find target items. As Fig. 15 shows, HMEH has a higher negative search throughput than CCEH. Because the FS-directory of HMEH stored in DRAM has lower access latency than that of CCEH placed in NVM. Since HMEH requires to lookup the extra stash when failing to find the target item, the search performance of HMEH decreases as the negative search ratio grows. To obtain better lookup performance, we can set the stash size to be smaller.

Table 2
Recovery time for different workload sizes.

Number of indexed items	1.6 million	16 million	160 million
RT-directory recovery time (ms)	0.47	6.3	50.1
FS-directory rebuild time (ms)	2.5	21.8	172.2

5.2.7. Recovery time of directories

At last, we evaluate the recovery time of two directories after a system failure with one thread. The recovery consists of two steps, recovering RT-directory and rebuilding FS-directory from RT-directory. We vary the number of inserted items from 16 million to 160 million and deliberately inject faults. Table 2 shows the recovery time of different workload sizes with one thread. We see that the recovery time of RT-directory only takes 0.47 msec and 50.1 msec if there are 1.6 million and 160 million items in HMEH. The rebuild time of FS-directory spends 2.5 msec and 172.2 msec. Compared to the whole execution time, the recovery time is at the millisecond level which is negligible. Therefore, directories of HMEH can achieve an instantaneous recovery.

6. Conclusion

Existing NVM-friendly hashing schemes suffer from two weaknesses: (1) the defective designs of hashing structures and (2) high overhead for data consistency. In this paper, we propose HMEH, a variant of extendible hashing for hybrid DRAM-NVM memory, to address such problems. First, we adopt the extendible hashing structure for cost-efficient resizing but place the directory in DRAM to obtain faster access. We keep a radix-tree structure in NVM to rebuild FS-directory upon recovery. Second, we leverage cross-KV and delayed flush mechanisms to greatly reduce the overhead of data consistency. To efficiently support multi-threaded operations, we also implement an optimized HMEH that delivers high performance and scalability. Using real Intel Optane DCPMM, experimental results present that OP-HMEH achieves up to 2.18× higher throughput than state-of-the-art persistent hashing structures. The optimized HMEH provides higher insertion scalability that obtains up to 1.18× speedup than the original HMEH under YCSB workloads.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by NSFC, China (No. 61832020), National Key R&D Program of China (No. 2018YFB1003305), Project of Shenzhen Technology Scheme, China (JCYJ20210324141601005), and Natural Science Foundation of Shandong Province, China (No. ZR2019LZH012). We are grateful to our anonymous reviewers for their constructive comments and suggestions.

References

[1] Intel, Intel and micron produce breakthrough memory technology, 2015, Retrieved from https://newsroom.intel.com/news-releases/intel-andmicron-produce-breakthrough-memory-technology.

[2] S. Raoux, G.W. Burr, M.J. Breitwisch, C.T. Rettner, Y. Chen, R.M. Shelby, M. Salinga, D. Krebs, S. Chen, H. Lung, C.H. Lam, Phase-change random access memory: A scalable technology, IBM J. Res. Dev. (2008) 465–479, http://dx.doi.org/10.1147/rd.524.0465.

[3] T. Kawahara, Scalable spin-transfer torque RAM technology for normally-off computing, IEEE Des. Test Comput. 28 (1) (2011) 52–63, http://dx.doi.org/10.1109/mdt.2010.97.

[4] Intel, Intel® optane™ DC persistent memory, 2019, Retrieved from https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.

[5] M.K. Qureshi, V. Srinivasan, J.A. Rivers, Scalable high performance main memory system using phase-change memory technology, in: Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA, 2009, http://dx.doi.org/10.1145/1555754.1555760.

[6] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, W. Lehner, FPTree: A Hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory, in: Proceedings of the 2016 International Conference on Management of Data, SIGMOD, 2016, http://dx.doi.org/10.1145/2882903.2915251.

[7] J. Yang, Q. Wei, C. Chen, C. Wang, K.L. Yong, B. He, NV-tree: Reducing consistency cost for NVM-based single level systems, in: Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST, 2015, pp. 167–181, URL: http://dl.acm.org/citation.cfm?id=2750482.2750495.

[8] F. Xia, D. Jiang, J. Xiong, N. Sun, HiKV: A hybrid index key-value store for DRAM-NVM memory systems, in: 2017 USENIX Annual Technical Conference, USENIX ATC, 2017, pp. 349–362, URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia.

[9] S. Chen, Q. Jin, Persistent B+-trees in non-volatile main memory, in: Proceedings of the VLDB Endowment (PVLDB), 2015, pp. 786–797, http://dx.doi.org/10.14778/2752939.2752947.

[10] D. Hwang, W. Kim, Y. Won, B. Nam, Endurable transient inconsistency in byte-addressable persistent B+-tree, in: 16th USENIX Conference on File and Storage Technologies, FAST, 2018, pp. 187–200, URL: https://www.usenix.org/conference/fast18/presentation/hwang.

[11] S.K. Lee, K.H. Lim, H. Song, B. Nam, S.H. Noh, WORT: Write optimal radix tree for persistent memory storage systems, in: 15th USENIX Conference on File and Storage Technologies, FAST, USENIX Association, Santa Clara, CA, 2017, pp. 257–270, URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon.

[12] hivaram Venkataraman, N. Tolia, P. Ranganathan, R.H. Campbell, Consistent and durable data structures for non-volatile byte-addressable memory, in: Proceedings of the 9th USENIX Conference on File and Storage Technologies, FAST, 2011, pp. 61–75, URL: http://www.usenix.org/events/fast11/tech/techAbstracts.html#Venkataraman.

[13] B. Debnath, A. Haghdoost, A. Kadav, M.G. Khatib, C. Ungureanu, Revisiting hash table design for phase change memory, in: Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads, INFLOW, 2015, http://dx.doi.org/10.1145/2819001.2819002.

[14] P. Zuo, Y. Hua, A write-friendly hashing scheme for non-volatile memory systems, in: Proceedings of the 33rd International Conference on Massive Storage Systems and Technology, MSST, 2017.

[15] P. Zuo, Y. Hua, J. Wu, Write-optimized and high-performance hashing index scheme for persistent memory, in: 13rd USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2018, pp. 461–476, URL: https://www.usenix.org/conference/osdi18/presentation/zuo.

[16] P. Zuo, Y. Hua, J. Wu, Level hashing: A high-performance and flexible-resizing persistent hashing index structure, ACM Trans. Storage 15 (2019) 1–30, http://dx.doi.org/10.1145/3322096.

[17] M. Nam, H. Cha, Y. Choi, S.H. Noh, B. Nam, Write-optimized dynamic hashing for persistent memory, in: Proceedings of the 17th USENIX Conference on File and Storage Technologies, FAST, USENIX Association, Boston, MA, 2019, pp. 31–44, URL: https://www.usenix.org/conference/fast19/presentation/nam.

[18] B. Lu, X. Hao, T. Wang, E. Lo, Dash: Scalable hashing on persistent memory, Proc. VLDB Endow. 13 (8) (2020) 1147–1161, http://dx.doi.org/10.14778/3389133.3389134, URL: http://www.vldb.org/pvldb/vol13/p1147-lu.pdf.

[19] J. Zhu, K. Huang, X. Zou, C. Huang, N. Xu, L. Fang, HDNH: a read-efficient and write-optimized hashing scheme for hybrid DRAM-NVM memory, in: Proceedings of the 50th International Conference on Parallel Processing, ICPP, Chicago, Illinois, USA, 2021.

[20] R. Fagin, J. Nievergelt, N. Pippenger, H.R. Strong, Extendible hashing-a fast access method for dynamic files, ACM Trans. Database Syst. (1979) 315–344, http://dx.doi.org/10.1145/320083.320092.

[21] W. Litwin, Linear hashing: A new tool for file and table addressing, in: Sixth International Conference on Very Large Data Bases, October 1-3, 1980, Montreal, Quebec, Canada, Proceedings, IEEE Computer Society, Montreal, Quebec, Canada, 1980, pp. 212–223.

[22] S. Patil, G.A. Gibson, Scale and concurrency of GIGA+: file system directories with millions of files, in: Proceedings of the 9th USENIX Conference on File and Storage Technologies, FAST, 2011, pp. 177–190, URL: http://www.usenix.org/events/fast11/tech/techAbstracts.html#Patil.

[23] ORACLE, Architectural overview of the oracle ZFS storage appliance, 2018, https://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/o14-001-architecture-overviewzfsa-2099942.pdf.

[24] PostgreSQL, PostgreSQL, 2020, http://www.postgresql.org/.

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

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

[27] S. Li, H. Lim, V.W. Lee, J.H. Ahn, A. Kalia, M. Kaminsky, D.G. Andersen, S. O, S. Lee, P. Dubey, Architecting to achieve a billion requests per second throughput on a single key-value store server platform, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA, 2015, http://dx.doi.org/10.1145/2749469.2750416.

[28] H. Garcia-Molina, K. Salem, Main memory database systems: an overview, IEEE Trans. Knowl. Data Eng. (TKDE) (1992) 509–516, http://dx.doi.org/10.1109/69.180602.

[29] H. Lim, M. Kaminsky, D.G. Andersen, Cicada: dependably fast multi-core in-memory transactions, in: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD, 2017, http://dx.doi.org/10.1145/3035918.3064015.

[30] Y.O. Koçberber, B. Grot, J. Picorel, B. Falsafi, K.T. Lim, P. Ranganathan, Meet the walkers: Accelerating index traversals for in-memory databases, in: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2013, http://dx.doi.org/10.1145/2540708.2540748.

[31] P.J. Mucci, S. Browne, C. Deane, G. Ho, PAPI: A portable interface to hardware performance counters, in: Proceedings of the Department of Defense HPCMP Users Group Conference, Vol. 710, 1999.

[32] S. Scargall, Persistent memory architecture, in: Programming Persistent Memory: A Comprehensive Guide for Developers, A Press, Berkeley, CA, 2020, pp. 11–30, http://dx.doi.org/10.1007/978-1-4842-4932-1_2.

[33] D.S. Rao, S. Kumar, A.S. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, J. Jackson, System software for persistent memory, in: Proceedings of the Ninth European Conference on Computer Systems, EuroSys, http://dx.doi.org/10.1145/2592798.2592814.

[34] H. Volos, A.J. Tack, M.M. Swift, Mnemosyne: lightweight persistent memory, in: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2011, http://dx.doi.org/10.1145/1950365.1950379.

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

[36] X. Han, J. Tuck, A. Awad, Dolos: Improving the performance of persistent applications in ADR-supported secure memory, in: MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2021, http://dx.doi.org/10.1145/3466752.3480118.

[37] B. Pittel, Linear probing: The probable largest search time grows logarithmically with the number of records, J. Algorithms 8 (2) (1987) 236–249, http://dx.doi.org/10.1016/0196-6774(87)90040-x.

[38] R. Pagh, F.F. Rodler, Cuckoo hashing, J. Algorithms 51 (2) (2004) 122–144, http://dx.doi.org/10.1016/j.jalgor.2003.12.002.

[39] D. Lea, Package util.concurrent release 1.3.4, 2003, http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html.

[40] P.L. Lehman, S.B. Yao, Efficient locking for concurrent operations on B-trees, ACM Trans. Database Syst. 6 (4) (1981) 650–670, http://dx.doi.org/10.1145/319628.319663.

[41] S.K. Cha, S. Hwang, K. Kim, K. Kwon, Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems, in: Proceedings of the 27th International Conference on Very Large Data Bases, in: Proceedings of the VLDB Endowment (PVLDB), Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2001, pp. 181–190.

[42] Y. Mao, E. Kohler, R.T. Morris, Cache craftiness for fast multicore key-value storage, in: Proceedings of the 7th European Conference on Computer Systems, Eurosys, ACM, 2012, pp. 183–196, http://dx.doi.org/10.1145/2168836.2168855.

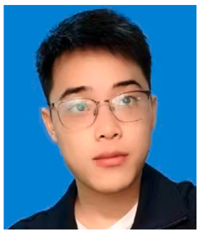
[43] P. Fatourou, N.D. Kallimanis, T. Ropars, An efficient wait-free resizable hash table, in: Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA), Vienna, Austria, July 16-18, ACM, Vienna, Austria, 2018, pp. 111–120, http://dx.doi.org/10.1145/3210377.3210408.

[44] M. Friedman, M. Herlihy, V.J. Marathe, E. Petrank, A persistent lock-free queue for non-volatile memory, in: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPoPP), Vienna, Austria, February 24-28, ACM, Vienna, Austria, 2018, pp. 28–40, http://dx.doi.org/10.1145/3178487.3178490.
[45] Y. Sun, Y. Hua, Z. Chen, Y. Guo, Mitigating asymmetric read and write costs in cuckoo hashing for storage systems, in: 2019 USENIX Annual Technical Conference (USENIX ATC) , Renton, WA, USA, July 10-12, USENIX Association, Renton, WA, USA, 2019, pp. 329–344, URL: https://www.usenix.org/conference/atc19/presentation/sun.
[46] T. Wang, J.J. Levandoski, P. Larson, Easy lock-free indexing in non-volatile memory, in: 34th IEEE International Conference on Data Engineering, (ICDE) , Paris, France, April 16-19, IEEE Computer Society, Paris, France, 2018, pp. 461–472, http://dx.doi.org/10.1109/ICDE.2018.00049.
[47] B. Fan, D.G. Andersen, M. Kaminsky, Memc3: Compact and concurrent memcache with dumber caching and smarter hashing, in: Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, NSDI, 2013, pp. 371–384.
[48] X. Zou, W. Fang, D. Fen, J. Chen, C. Liu, F. Li, N. Su, HMEH: write-optimal extendible hashing for hybrid DRAM-NVM memory, in: Proceedings of the 36rd International Conference on Massive Storage Systems and Technology, MSST, 2020.
[49] L.R. Johnson, An indirect chaining method for addressing on secondary keys, Commun. ACM (1961) 218–222, http://dx.doi.org/10.1145/366532.366540.
[50] L. Carter, M.N. Wegman, Universal classes of hash functions, J. Comput. System Sci. 18 (2) (1979) 143–154, http://dx.doi.org/10.1016/0022-0000(79)90044-8.
[51] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, S. Swanson, An empirical guide to the behavior and use of scalable persistent memory, in: 18th USENIX Conference on File and Storage Technologies, FAST, USENIX Association, Santa Clara, CA, 2020, pp. 169–182.
[52] Intel, Persistent memory development kit, 2019, http://pmem.io/.
[53] S.K. Lee, J. Mohan, S. Kashyap, T. Kim, V. Chidambaram, Recipe: converting concurrent DRAM indexes to persistent-memory indexes, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP, 2019, pp. 462–477, http://dx.doi.org/10.1145/3341301.3359635.
[54] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC, 2010, http://dx.doi.org/10.1145/1807128.1807152.
[55] X. Li, D.G. Andersen, M. Kaminsky, M.J. Freedman, Algorithmic improvements for fast concurrent cuckoo hashing, in: Proceedings of the Ninth European Conference on Computer Systems, EuroSys, 2014, http://dx.doi.org/10.1145/2592798.2592820.
[56] Libcuckoo, Libcuckoo library, 2019, https://github.com/efficient/libcuckoo.

Xiaomin Zou received the B.E. degree in computer science and technology from the Nanchang University (NCU), China, in 2017. She is currently working toward the Ph.D. degree majoring in computer architecture at the Huazhong University of Science and Technology (HUST), China. Her research interests include non-volatile memory (NVM) and key–value stores. She published a paper at the conference of Mass Storage Systems and Technologies (MSST).

Fang Wang received the B.E., M.E., and Ph.D. degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), China, in 1994, 1997, and 2001, respectively. She is a Professor of computer science and engineering at HUST. She has more than 80 publications in major journals and international conferences, including FGCS, IEEE TPDS, ACM TACO, ICDE, HiPC, ICDCS, HPDC, ICPP, ICCD. Her interests include distribute file systems, parallel I/O storage systems, and graph processing systems.

Dan Feng received the B.E., M.E., and Ph.D. degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), China, in 1991, 1994, and 1997, respectively. She is a professor and vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 80 publications to her credit in journals and international conferences, including IEEE TPDS, JCST, USENIX ATC, FAST, ICDCS, HPDC, SC, ICS, and ICPP. She is a member of the IEEE.


Junhao zhu received the B.E. degree from National University of Defense Technology, Changsha, China in 2019. He is currently a master at National University of Defense Technology. His research interests include non-volatile memory computing and in-memory database systems. He published a paper in the conference of International Conference on Parallel Processing (ICPP).

Renzhi Xiao received the B.E. degree in software engineering from Jiangxi University of Science and Technology, Nanchang, China, in 2013. He is currently working toward the Ph.D. degree majoring in computer architecture at Huazhong University of Science and Technology(HUST), Wuhan, China. His research interests include computer architecture, in-memory key–value store, non-volatile memory, and NVM-based data structures.

Nan Su received the B.E. and M.E. degrees from Shandong University of Science and Technology, China, in 2003 and 2007. She is a engineer at Inspur. She is engaged in research work in distributed storage systems, non-volatile memory storage systems.