HMEH: write-optimal extendible hashing for hybrid DRAM-NVM memory

Xiaomin Zou[1], Fang Wang[1*], Dan Feng[1], Janxi Chen[1], Chaojie Liu[1], Fan Li[1] and Nan Su[2]
[1]Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, China
[2]State Key Laboratory of High-end Server & Storage Technology, Shandong Massive Information Technology Research Institute

***Abstract*—Emerging non-volatile memory (NVM) is expected to coexist with DRAM as a hybrid memory to fully exploit the complementary strengths of DRAM's low read-write latency and NVM's high density, persistence, and low standby power. However, existing hashing schemes cannot efficiently reap the benefits of such a hybrid memory. In this paper, we present a hybrid DRAM-NVM write-optimal and high-performance dynamic hashing scheme, named HMEH (Hybrid Memory Extendible Hashing). In our design, key-value items are persisted in NVM while the directory is placed in DRAM for faster access. To rebuild the directory upon recovery, HMEH also keeps a radix-tree-structured directory in NVM with negligible overhead. Furthermore, HMEH proposes a cross-KV strategy to write back items through natural eviction, which can ensure data consistency with no performance degradation from persist barriers. Experimental results show that HMEH outperforms the state-of-the-art NVM-based hashing structures by up to 2.47×. And concurrent HMEH also delivers superior performance and high scalability under YCSB workloads with different search/insertion ratios.**

***Index Terms*—hybrid DRAM-NVM memory, dynamic hashing, data consistency, write optimization**

I. 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]. The byte-addressable NVMs are promising candidates for replacing DRAM with the disk-like durability and near-DRAM access performance. However, current NVM technologies still suffer asymmetric read-write latencies and limited write endurance [4]. As a result, NVM is expected to coexist with DRAM to form a hybrid DRAM-NVM memory [4]–[7].

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. A large body of prior research has been done to improve tree-based indexing structures for the systems equipped with NVM [5], [6], [8]–[11]. Due to hash-based indexing structures' constant time complexity, i.e., O (1), for point accesses, which is superior to the tree-based structures, hashing variants have been proposed recently for NVM-oriented memory, such as PFHT [12], path hashing [13], level hashing [14] and CCEH [15]. However, it is a commonly held belief that NVM will not replace DRAM overnight and will instead coexist with DRAM for a foreseeable future. To the best of our knowledge, there is no study on hash-based indexing structures for hybrid DRAM-NVM memory systems where such structures can be greatly beneficial, particularly if the complementary advantages of DRAM and NVM are fully leveraged. To this end, we devote to effectively improving the hash-based structure for the hybrid DRAM-NVM memory.

Most NVM-oriented hashing schemes focus on static hashing structures [12]–[14], but they are limited by two inherent weaknesses. First, the size of hash table must be estimated in advance for allocating adequate memory space. However, it is almost impossible given the dynamic nature of most applications. Second, the rehashing operation of static hashing is complicated that needs 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. It not only incurs massive extra writes but also blocks all foreground user query requests, drastically degrading application performance. In terms of NVM-based memory with relatively high latency and low endurance, it suffers a higher performance penalty and even exacerbates the wear-out problems.

Fortunately, extendible hashing [16] can grow and shrink gracefully according to the data size, which has been widely applied in file systems and database systems [17]–[19]. It is a dynamic hashing scheme that consists of a set of buckets and an array of bucket addresses, called directory. The unique characteristics of NVM force extendible hashing to make some changes, as reduction of NVM writes and data consistency should be taken into consideration. Most of existing researches apply persist barriers to guarantee consistency. However, these instructions are proved to incur performance degradation [6], [11]. In this paper, we present a write-optimal and consistent extendible hashing called HMEH (Hybrid Memory Extendible Hashing), which significantly reduces the overhead of data consistency and delivers high performance.

HMEH stores key-value items in NVM for directly persisting and places flat-structured directory in DRAM to improve overall system performance, since extra accesses to the directory are in the critical path and the read/write latencies of DRAM is much lower than NVM. However, because of residing in volatile DRAM, the flat-structured directory cannot recover from system failures or normal shutdowns,so we

maintain a radix-tree-structured directory in NVM to resolve this problem since updates of radix tree do not incur extra NVM writes and are easy to ensure consistency. To guarantee data consistency with low overhead, we leverage a cross-KV strategy to rearrange key and value of an item into multiple failure-atomic 8-byte slices and they are written back to NVM by natural evictions without expensive persist barriers. Furthermore, we exploit delayed flush to reduce the overhead of data persistence when splitting segments.

In summary, the main contributions of this paper are:

- 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 flat-structured directory from system crashes or normal shutdowns, we keep a radix-tree-structured directory in NVM which only incurs negligible overhead but realizes a fast recovery.
- We propose a cross-KV strategy that alternately stores 4 bytes of key and 4 bytes of value, and the like, to form multiple 8-byte units. We can leverage the hash key to verify the correctness of the 8-byte units in case of unexpected system failures, ensuring data consistency without memory fence and cache line flush instructions.
- We elaborate on the process of failure-atomic segment split that carefully enforces the ordering of modification to guarantee data consistency without costly logging or CoW. And to mitigate the overhead of data persistence, we leverage a delayed flush method which only persists the split segment when a new node of the radix-tree-structured directory is created.
- We implement HMEH and conduct extensive evaluations to show the efficiency of these designs. The experimental results demonstrate that the write latency of HMEH is up to 1.49× and 2.47× shorter than that of state-of-the-art CCEH [15] and level hashing [14] respectively. The concurrent HMEH also shows best performance and high scalability under YCSB workloads with different search/insertion ratios.

The rest of this paper is organized as follows. Section II describes the background and related work. Section III presents the detail designs and implementation of HMEH. In Section IV we implement a concurrent and consistent HMEH in hybrid memory. The performance evaluation is shown in Section V. Finally, Section VI concludes this paper.

II. Background and Motivation

A. Extendible Hashing

Extendible hashing is a dynamic hashing technique optimized for time-sensitive applications, which can dynamically allocate and deallocate hash buckets on demand [16]. It consists of multiple buckets each of which stores a fixed number of items, and a resizable array, called directory, where each entry stores a pointer to the bucket.

As Figure 1 illustrates, extendible hashing distributes items to buckets by hash keys, and uses the trailing (least significant) bits of the hash key to index the directory. 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 directory. To improve memory efficiency, the number of buckets does not require to be the same as the number of directory entries. Hence, each bucket also keeps a local depth (LD) which expresses the number of the common bits in the bucket. When a bucket is only pointed by one directory entry, LD equals to GD. And if n directory entries point to a bucket, LD = GD-$\log_2$n. Taking an example in Figure 1, bucket 1 is pointed by two directory entries and GD is 2, so the LD of bucket1 is 1.

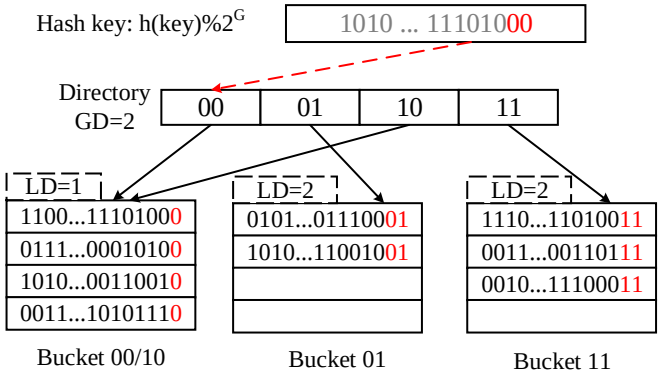


Fig. 1. Extendible hashing structure.

As shown in Figure 1, suppose that we insert a new item into bucket1 in which there is no empty slots. And the local depth of bucket1 is smaller than the global depth which means bucket1 is pointed by multiple directory entries. Therefore, we can directly split bucket1 into two buckets and modify directory entry10 to point to new split segment. Next, the LDs of two buckets are increased to 2. However, if bucket3 whose LD is equal to GD overflows, we first need to resize the directory, creating a double-sized directory and migrating all entries in the old directory to the new one. And then we perform a bucket split.

B. Hashing Index Structure in NVM

The hash-based indexing structures support constant-complexity point query operations, widely used in key-value stores [20]–[22] and main memory databases [23]–[25]. Recently several hashing-based structures have been proposed to efficiently adapt to NVM. PFHT [12] is a PCM-friendly cuckoo hashing variant which only allows one cuckoo displacement to avoid cascading NVM writes and designs a chained stash to store conflicting items. Path hashing [13] is a write-friendly hashing that logically organizes storage cells as an inverted binary tree and exploits position sharing to solve hash collisions. However, neither of them takes data consistency issues into account and do not provide a constant-level lookup.

TABLE I
THE LLC CACHE MISS RATES OF DIRECTORY IN CCEH.

Number of Indexed Items	0.16 million	1.6 million	16 million
Directory size	8KB	64KB	512KB
LLC cache miss	16.14%	74.31%	98.77%

Level hashing [14] proposes 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 the hash table needs to resize, the items in the old bottom level are rehashed to a new 4× larger hash table and the previous top level can be reused as the new bottom level. The evaluation presents that level hashing significantly outperforms PFHT and path hashing. But actually, the rehashing overhead of level hashing is similar to other hashing schemes [15].

The mentioned NVM-based hashing schemes are all static hashing schemes with unacceptable resizing latencies. CCEH [15] is a variant of extendible hashing that dynamically splits and merges hash buckets as needed to overcome the shortcoming of full-table or 1/3-table rehashing. It employs a three-level structure to balance access performance against the directory size. Furthermore, CCEH has two versions which are different in the way of segment splits. In-place CCEH reuses old segment and updates items in place, while CoW CCEH creates two new segments and performs CoW splits.

The main disadvantage of CCEH is that every insertion or search needs an extra access to the directory. To measure its overhead, we use the PAPI library [26] to count the LLC cache miss rate of the directory under different workloads. As illustrated in table 1, with the increase of data size, a bigger directory leads to a higher cache miss rate, even up to 98%. In terms of NVM whose read latency is much higher than DRAM, the extra accesses to the directory are in the critical path, drastically diminishing the system performance.

C. Data Consistency for Hashing Schemes in NVM

When NVM complements or substitutes DRAM as the main memory in the computer system, it is a fundamental requirement to ensure data consistency, i.e., data recoverability and correctness. Because when a system failure occurs, data in volatile CPU cache will be lost, but incomplete data modifications still exist in non-volatile memory, causing an inconsistent issue. To guarantee data consistency, it is essential to make NVM writes become durable in a desired order [5], [8], [10]. However, memory writes may be reordered by CPU or memory controller for better performance. Consequently, we have to use persist barriers to form ordered memory writes as existing schemes [9], [10], [14]. Specifically, memory fence instructions (MFENCE), e.g., *mfence* and *sfence*, stall thread until all its preceding operations are finished, and Cache line flush instructions (CLFLUSH), e.g., *clflush*, *clflushopt* and *clwb*, write back dirty cache lines to memory [27].

Different from block-based storage devices, the failure-atomic unit of NVM is only 8 bytes [5], [6], [10], [11]. Thus, updates with larger size may be partially written after unexpected system failures. Existing works exploit logging or copy-on-write (CoW) to ensure the atomicity of data larger than 8 bytes [8], [28], [29]. But these techniques cause substantial extra NVM writes and implicitly aggravate ordering overhead, e.g., the write ordering constraints of data blocks during committing a log.

In light of NVM-based hashing schemes, most of them exploit a sequence of persist barriers to guarantee data consistency [14], [15]. For example, when inserting a new item into a hash bucket, 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 its key is not valid for the hash bucket.

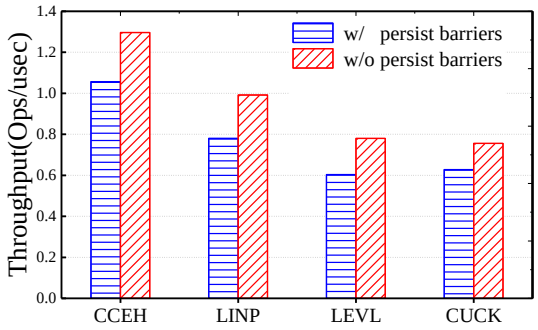


Fig. 2. The insertion throughputs of different hashing schemes.

However, persist barriers are expensive and their overhead is proportional to the amount of NVM writes. To quantify their cost, we measure the average insertion throughputs in common hashing schemes, (a) linear probing [30] and (b) cuckoo hashing [31], (c) level hashing [14], a state-of-the-art NVM-based static hashing, (d) CCEH [15], a recently proposed dynamic hashing, with and without persist barriers (referred as w/ persist barriers and w/o persist barriers respectively). The details of the experimental setup are presented in Section V-A. As shown in Figure 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 more important to reduce the number of persist barriers.

III. HMEH DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of HMEH, a write-optimal and flexible extendible hashing for hybrid DRAM-NVM memory. As shown in Figure 3, we place flat-structured directory in DRAM for fast access and place hash table in NVM for persistence. To instantaneously recover flat-structured directory, we also maintain a radix-tree-directory in NVM. Similar to previous work [15], [16], [32], we introduce an intermediate segment layer to balance the directory size and access performance. As Figure 2 shows, a segment consists of multiple buckets and a stash for colliding items. For better CPU cache efficiency, we exploit cacheline-sized buckets. And we use the most significant bits (MSB) of

the hash key to index segments and exploit the least significant bits (LSB) as the bucket index.

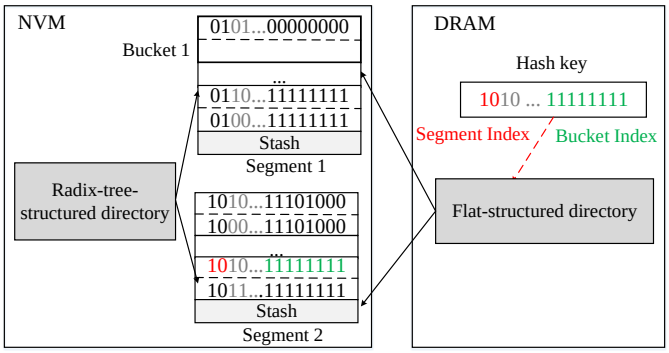


Fig. 3. Architecture of HMEH.

A. Two Structures of Directory

As we know, for existing extendible hashing in NVM, the access to the directory is in the critical path which significantly increases the latency of operations. Since the write/read latency of DRAM is much lower than that of NVM, we keep a flat-structured directory (FS-directory) in DRAM while storing hash table in NVM to leverage the performance characteristics of hybrid memory. DRAM-based FS-directory offers two benefits: First, as mentioned before, it delivers faster access that can significantly improve the overall performance. Second, we do not require to guarantee its crash consistency.

However, after a system crash or normal shutdown, FS-directory in volatile DRAM will be lost. Thus, we design a secondary directory in NVM to recover FS-directory. Fortunately, we found that the characteristics of radix tree can be efficiently utilized in NVM. Specifically, the radix tree structure is determined by the prefix of the inserted key which coincides with segment indexes (the MSB bits of hash keys). And when expanding size, it reuses the old nodes and directly adds a new node without modifying any other existing nodes, which can be completed with an atomic 8-byte update operation.

Hence, we retain a radix-tree-structured directory (RT-directory) in NVM to rebuild FS-directory upon recovery. Figure 4 shows the corresponding relationship between two directories. We set the size of RT-directory node as a cache line for high CPU cache efficiency. When a segment splits, we first update the corresponding entries in its RT-directory node and then modify FS-directory. The experimental results in Section V-B show the RT-directory only causes negligible overhead but achieves an instantaneous recovery.

B. Cross-KV Mechanism

As described in Section II-C, to guarantee data consistency, existing hashing schemes for NVM exploit persist barriers to enforce the order of NVM writes, which incurs significant overhead.

Therefore, we propose a state-of-the-art cross-KV strategy to bypass persist barriers in most situations. Since modern

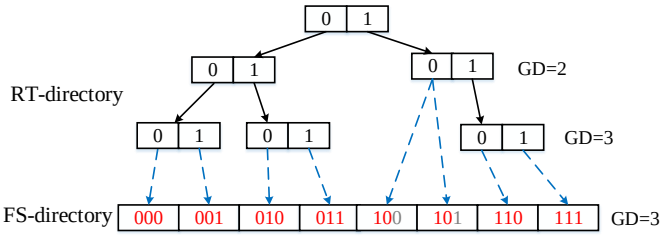


Fig. 4. The relationship of two directories.

processors support 8-byte atomic write, we split an item into several pieces, and then alternately store key and value as several 8-byte atomic slices. Figure 5 shows the storage structure of cross-KV. Suppose the sizes of key and value are both 8 bytes. We divide key (value) into key1 and key2 (value1 and value2), next we combine key1 and value1 as cross-KV1, the same to cross-KV2. 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 not, it indicates the key is not valid, and also means the value is partially written.

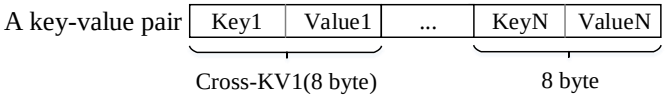


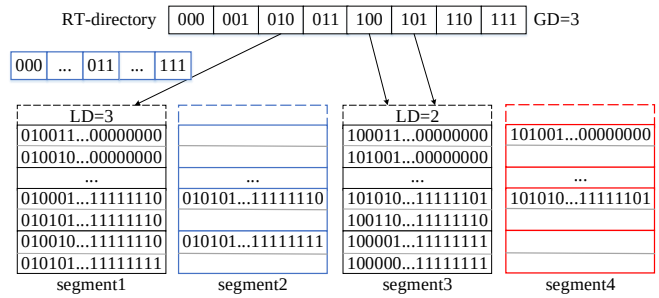
Fig. 5. The storage structure of cross-KV.

However, there is a special case that cross-KV can not guarantee data consistency. That is, partially written items may still index the same segment and bucket. Thus, the key cannot prove whether the value is correctly persisted to NVM. To address this issue, before insertion, we form all possible partially written keys that may be produced by a system crash. And 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 very low, so the cross-KV mechanism is still efficient.

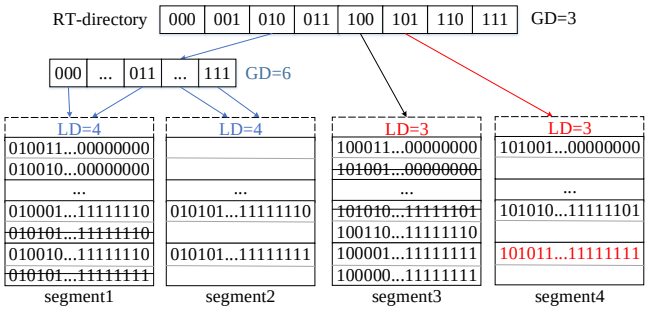
Though HMEH leverages a unique cross-KV structure to avoid the overhead of persist barriers in most cases, it also sacrifices a little performance. First, when searching a key, we require to read the entire item unlike other hashing indexes that only need to read the key. However, this read overhead can be ignored since we employ cacheline-sized buckets and a single access can prefetch multiple cross-KVs belong to the same item. Second, we require to check the above special case. Fortunately, the calculation overhead only incurs minor overhead and is much less than the overhead of persist barriers. To support variable-length key and value sizes, like previous researches, we store key-value pairs outside the hash table and place their pointers and the short summary of the key in the hash table [14], [33].

C. Low-overhead In-place Segment Split

To reduce NVM writes and mitigate the overhead of segment split, we propose a low-overhead in-place segment split mechanism. The basic idea of our segment split mechanism is to reuse the old segment and rehash items to a new-created segment, whose data consistency can be ensured by persist barriers without logging or CoW.



(a) Step1: create new segment (radix tree node) and migrate records to new segment



(b) Step2: split and lazy deletion

Fig. 6. Examples of failure-atomic segment split.

Two Examples of In-place Segment Split. We use two examples of Figure 6 to elaborate on the process of segment split. Since we do not need to guarantee FS-directory consistency, Figure 6 only shows the updates of RT-directory. In the first example (marked in red), with the given key $101011...11111111_{(2)}$, we use the leftmost 3 bits of the hash key to index FS-directory and then find segment3, but there are no free slots in segment3. Therefore, we require to create a new segment. Since the LD of segment3 is 2 smaller than GD, we do not need to resize RT-directory in advance. As Figure 6(a) shows, HMEH creates a new segment4 and copies the items with prefix 101 from segment3 to it. To reduce NVM writes, we do not delete those migrated items in segment3, because they become invalid with the modification of LD, and subsequent inserted items will overwrite them directly.

Next, we need to update the RT-directory entry of segment4 and LDs of two segments in a particular order. As illustrated in Figure 6(b), we firstly change the LD and the pointer for segment4, then we modify LD of segment3. If these updates are out of order, we cannot recover our hash table from a system failure. Therefore, we exploit persist barriers to constrain the ordering of these updates. Moreover, we also support segment merge which is an inverse process of segment split.

The other example of segment split involves the expansion of directories, as depicted in Figure 6 (marked in blue). When splitting segment1, we require to increase the size of RT-directory, since the LD of segment1 is equal to GD. Thanks to the structure of radix tree, we only need to allocate a new RT-directory node instead of doubling the entire directory. Then we create a new segment2 and rehash items of segment 1 into segment 2. In the next step shown in Figure 6(b), we also ensure the ordering of RT-directory updates to survive system failures. That is, (1) we update the entries of the new RT-directory node and modify LD for segment2. Second, (2) we change entry010 to the pointer of the new RT-directory node. Finally, (3) we update LD of segment1 to 4.

Reducing the Persistency Overhead. Traditionally, when a segment splits, to identify invalid items in case of system failures, we also need to write back items of the new split segment to NVM with expensive CLFLUSH instructions. Since our cross-KV can distinguish partially written items upon recovery, in order to alleviate the overhead of data persistence, we use a delayed flush method that exploits normal cache evictions to write back the new split segment and leverages unique RT-directory structure to flush segments regularly, which prevents certain cache lines from residing in the cache for a long time.

The normal cache evictions may cause 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 overwrite 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. Then we create two new segments and rehash all items of the old segment into them. Thus, 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 CLFLUSH to flush one segment because an RT-directory node contains eight segment addresses. Therefore, this method mitigates the cost of data persistence, meanwhile, ensures data consistency. The performance gained by this design is shown in Figure 8.

D. Improvement of Load Factor

The load factor is another essential parameter for hashing structures in memory and caches with limited space, and a higher load factor means more items can be stored. However, extendible hashing schemes can only split segments to resolve hash collisions even if there are lots of free buckets in old segments, leading to a low load factor. In this section, we present our method to improve the load factor.

Several previous studies attempted to address the hash collisions. For example, chained hashing [34] stores colliding items in linked lists. But it requires frequent memory allocation

and pointer access, resulting in low CPU cache efficiency. Cuckoo hashing [31] exploits several hash functions and allows multiple cuckoo evictions, but it suffers from cascading writes which induce high insertion latency. Linear probing [30] scans the following buckets until it finds empty slots to store conflicting items. Thus, it is cache-friendly and we can sequentially access items. However, as the load factor increases, finding target keys needs to traverse plenty of slots, which degrades lookup performance.

In HMEH, we resolve hash collisions by combining linear probing and stash schemes.The default probing distance is four buckets (256 bytes), which is based on a recently released technique report that Intel Optane DC Persistent Memory is accessed by 256-byte block granularity [35]. When a hash collision occurs, we first probe the following four buckets to find a proper slot, and if fails, we insert the target item to stash. Specifically, the stash is array-structured secondary storage that is non-addressable and used to store colliding items. Every segment has a stash that all buckets in this segment share it. Therefore, we can obtain a higher load factor in a simple but efficient way.

E. 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 RT-directory to NVM, and then stores a flag to indicate a normal shutdown. When rebooting, HMEH only reads the RT-directory and rebuilds FS-directory in DRAM. We first get the global depth (GD), and then do a breadth-first search for RT-directory to retrieve the local depth (LD) and the starting position in FS-directory of each segment. As Figure 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 FS-directory according to starting positions and reference counts.

Recovery after a system crash. For a system crash, HMEH first requires to recover RT-directory. We exploit the global depth of RT-directory and the local depth of segments to check the entry consistency. As discussed in Section III-C, if LD is smaller than GD, there are several RT-directory entries pointing to the same segment. And if LD equals to GD, one entry corresponds to one segment. Therefore we can calculate the reference count of each segment and check if the corresponding entries are equal.

Since data consistency problems only happen in leaf nodes of RT-directory, we just need to recover leaf nodes. Algorithm 1 presents the pseudo-code of recovering an RT-directory node. We recover RT-directory nodes from left to right. We first access the leftmost entry to obtain LD and GD, and calculate the reference count, denoted as RefCount. Then we compare the LD of first entry with that of the following entries in the same reference count sequentially. If the LD of latter entry is different, we make this entry equal to the first entry, i.e., between line 5 and line 7 in Algorithm 1. We iteratively detect the inconsistencies of leaf nodes until the recovery of RT-directory is completed. At last, we rebuild FS-directory from RT-directory as mentioned before.

Algorithm 1 RT-directory Node Recovery

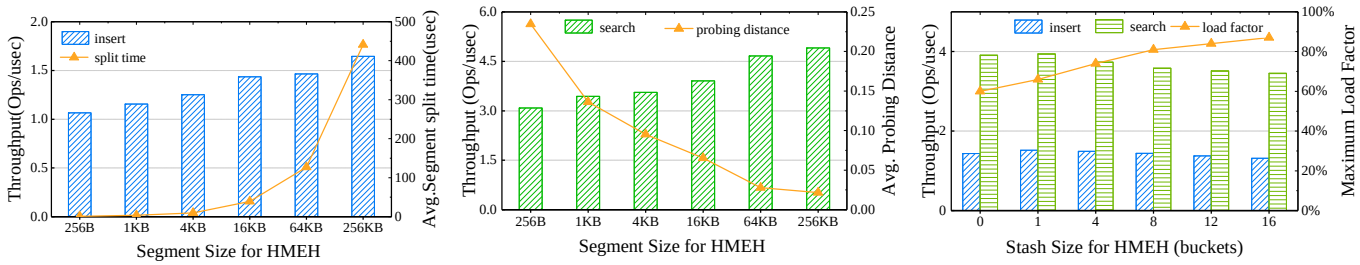
```
1: for i ← 0; i<RT-directorynode.capacity; i←i+1 do
2:     Depth ← NodeSlot[i].LD;
3:     Stride ← 2^(GD−Depth);
4:     Buddy ← i+Stride;
5:     for j ← Buddy-1; i<j; j ← j-1 do
6:         if NodeSlot[j].LD != Depth then
7:             NodeSlot[j] ← NodeSlot[i];
8:     i ← Buddy-1;
```

IV. Concurrent and Consistent HMEH

Multi-threaded concurrency is a key issue to enhance program performance in modern multi-core systems. To achieve better performance and high scalability, HMEH applies different concurrency schemes for specific situations.

Mutex and Version Number for Directories. We apply mutex to protect two directories from being updated inconsistently. And during FS-directory doubling, we must atomically update the global depth and the pointer to the FS-directory. Suppose that when GD is updated but the pointer is not, the index calculated by new GD and hash key may be out of the range of FS-directory. Conversely, if the pointer is swapped but GD is old, we probably access wrong segments, resulting in data loss or search failures. We can simply use cmpxchg16b instruction [36] to update two metadata, but this instruction is expensive and in the critical path. Hence, we apply version number to determine whether they are consistent. We store version numbers into the rightmost six bits of FS-directory pointer and the leftmost six bits of GD. Then, before indexing segments, we first check whether the version numbers are the same. If not, we will load them repeatedly until they have the same value. In our evaluation, the version number scheme has much better scalability.

Fine-grained Lock for Segment split and lock-free read. We use the segment-level lock to protect hashing table expansion. Generally, before splitting a segment, we must acquire the lock first, and other threads cannot modify the segment until the split is finished. Since the directory entries of the split segment remain unchanged, we can release the lock of the split segment after preparing a new segment to improve scalability. However, in this circumstance, an insertion should verify whether the key is belonging to the old segment, if not, this insertion will be repeated until succeeds. As for read operations, we implement a lock-free search but need to verify the local depth of the target segment. Specifically, after finishing the search, we will check whether the LD has been modified, and if so, we require to retry the query operation. That is because the updated LD indicates the target segment has been split, therefore, the target item might be invalid as the concurrent write might have modified it.



(a) Insertion throughput (b) Search/deletion throughput (c) Insertion throughput

Fig. 7. Throughput of different segment sizes.

Compare-and-swap (CAS) Instructions for Slots of Buckets. We apply CAS to protect slots in every bucket. For example, when a write transaction inserts an item to a bucket, we first search for a free slot. Then we use CAS instructions to mark the leading 8 bytes of item (cross-KV1 in our scheme) as a sentinel that indicates the operation to be performed. So that other write transactions will find other vacant slots. Hence, we can implement the lock-free write in a slot of buckets.

V. Performance Evaluation

A. Experimental Setup

We evaluate the performance of HMEH against the state-of-the-art NVM-based hashing indexes on Intel Optane DC Persistent Memory Module (DCPMM). All experiments are conducted on a 2-socket, 36-core Linux server (kernel version 5.0.0) equipped with 1.5 TB DCPMM, 186GB DRAM, and 32MB Last Level Cache(LLC). We use the ext4-DAX file system and the APP Direct mode of Optane DC to perform all experiments. We apply *clwb* [37] for cache line flushes, which is more efficient than *clflush* and *clflushopt*.

We compare our HMEH with existing hashing schemes, i.e., CCEH, Level hashing, persistent linear hashing and persistent cuckoo hashing (referred to as CCEH, LEVL, P-LINP, P-CUCK). We use the libvmem library from PMDK [38] to support traditional allocation interfaces on a volatile memory pool built on a memory-mapped file. In our experiments, the initial hash table of every scheme is sized for 2048 key-value items. And we use 160 million random integers as workload, whose key and value are both set to 8 bytes. We also measure the scalability of our concurrent HMEH in mixed workloads of YCSB [39], the industry standard for evaluating key-value indexes. In the experimental results, each value is the average of five executions.

B. Experimental Results and Analysis

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 256B to 256KB. 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.

Figure 7(a) illustrates the average throughput of inserting 160 million random items in HMEH with different segment sizes. Small segments that store fewer items will incur frequent segment splits, increasing the entire execution time. However, small segments only require to flush fewer cache lines into NVM, which minimizes the latency of a single segment split. As shown in Figure 7(a), the insertion throughput improves as the segment size grows. But the average latency of segment split sharply increases, and it even reaches 441 usec when the segment size is 256KB. From the experimental results, to balance average insertion throughput and latency of segment split, the reasonable segment size is in the range of 4KB to 16KB.

Generally, a search operation only requires to accesse one bucket to find the target item. However, the methods to address hash collisions increase the number of bucket accesses. In Figure 7(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, thus it avoids probing more buckets to find the target items. As Figure 7(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 16KB, and we vary stash sizes from 1 bucket to 16 buckets. And we also evaluate their load factors which are closely related to stash configuration. As Figure 7(c) shows, with the increase of stash size, the search performance degrades slowly but the maximum load factor linearly grows. From aforesaid observations, the optimal stash size is between 1 bucket and 8 buckets.

2) Comparative Performance: According to the experimental results of HMEH design, we set the segment size as 16KB with a stash whose size is 4 buckets for the rest of the experiments. We first analyze the performance gains brought by the designs of our HMEH. We insert 160 million random items and break down the average latency of inserting an item into maintaining directory time (denoted as directory), and other time spent in segment (denoted as segment). We apply persistent extendible hashing as our experimental baseline. As

Figure 8 shows, the design of hybrid memory yields significant performance gains, and the overhead of hybrid directories is smaller than that with FS-directory in NVM. This is because RT-directory can reuse the old nodes and the operations on FS-directory are executed in DRAM.

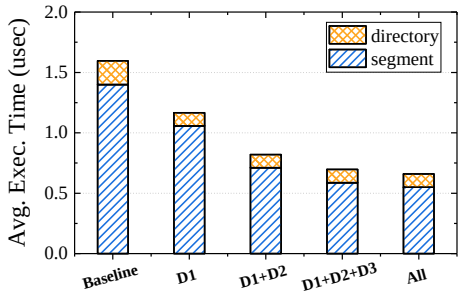


Fig. 8. Insertion Latency of HMEH when adding different designs. (Baseline: persistent extendible hashing; D1: the changes of structure; D2: cross-KV for insertion; D3: delayed flush; All: HMEH that uses D1+D2+D3+stash).

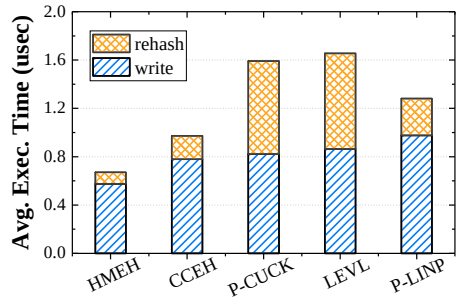


Fig. 9. Insertion Latency of different hashing schemes.

Then, we compare the average insertion latency of our HMEH against those of other state-of-the-art 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). For P-CUCK, we allow its insertion to try 16 evictions before rehashing. P-LINP performs full-table rehashing when the load factor achieves 95%. For LEVL, we optimize the bucket size as a cache line which can leverage high cache efficiency. For CCEH, we exploit the in-place version which can reuse the split segment, outperforming the copy-on-write version.

As shown in Figure 9, HMEH exhibits the best insertion performance. Compared with CCEH, P-CUCK, LEVL, and P-LINP, we observe that HMEH speeds up the insertions by over 1.49×, 2.37×, 2.47×, and 1.91×. HMEH 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.

3) Maximum Load Factor: To evaluate the maximum load factor, we insert 1 million items into empty HMEH, CCEH, and LEVL 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. For a fair comparison, we also evaluate CCEH in the same way.

LEVL has a two-level structure and each bucket has several slots. Furthermore, it uses two hash functions and allows one cuckoo eviction. Hence, there are multiple sharing slots to store collision items in LEVL. As shown in Figure 10, the maximum load factor of LEVL can achieve up to 92%. As linear probing distance and stash size grow, the max load factors of HMEH increase stably and all exceed 74%. 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.

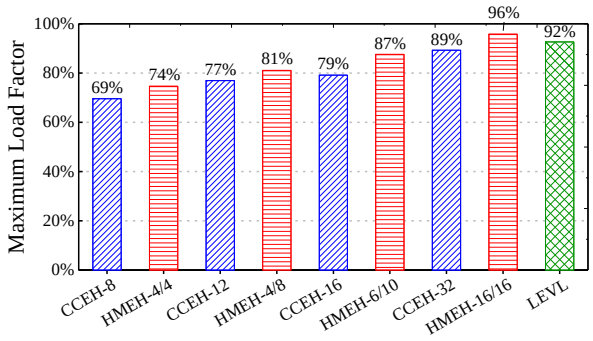
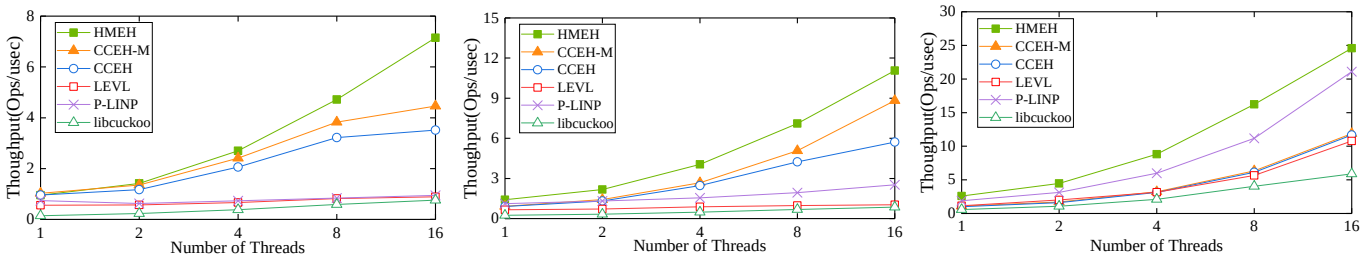


Fig. 10. Maximum load factors of hash tables. (# in the NAME-x/y# indicates the linear probing distance and the size of stash in buckets.)

4) Concurrent performance: We compare the concurrent performance of different concurrent hashing schemes. We first evaluate the scalability under an increasing number of threads, including a single thread, 2, 4, 8, 16 threads. We use YCSB to generate three workloads: 100% insert, 50% insert and 50% search, and 100% search workloads which all have 160 million items. We compare our concurrent HMEH with CCEH, LEVL, P-LINP, and libcuckoo [40], a state-of-the-art concurrent cuckoo hashing scheme. For libcuckoo, we use its open-source C++ implementation [41]. Furthermore, as CCEH and HMEH can employ the same concurrent strategy, we improve CCEH in our way as described in Section IV, denoted as CCEH-M.

The average throughputs of the insert-only workload are shown in Figure 11(a), our concurrent HMEH shows the highest throughput and best scalability. We see that CCEH-M obtains significant performance improvement than CCEH. This is because our multi-thread control decreases the waiting



(a) Scalability on YCSB: 100%insert (b) Scalability on YCSB: 50%insert-50%search (c) Scalability on YCSB: 100%search

Fig. 11. Scalability on YCSB workloads.

time of operations in other threads when segments split, described in Section IV. To prevent multi-thread from modifying a slot at the same time, libcuckoo requires to lock an entire cuckoo path, blocking all operations in other threads that access the slots in this cuckoo path. Therefore, libcuckoo suffers from frequent locking operations and has a worst performance. Note that the throughputs of LEVL and P-LINP do not scale with the increase of threads. The main reason is that their full-table rehashing operations incur high latencies and block all insertions in other threads.

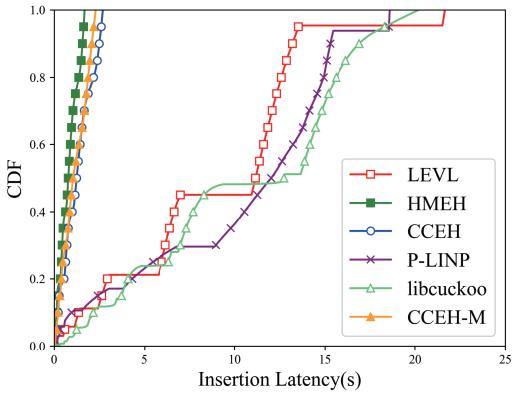


Fig. 12. Insertion Latency CDF.

Figure 11 also shows the average throughputs of 50% insert-50% search, and 100% search workloads. Getting benefits from hybrid memory and lock-free read, HMEH still performs best under all workloads. Note that P-LINP has good search performance, only behind HMEH. This is because P-LINP places collision items in the following buckets and one memory access can prefetch several adjacent buckets to CPU cache, which efficiently reduces memory accesses.

We also measure the tail latency of concurrent insertion. Figure 12 illustrates its cumulative distribution functions (CDF). Since static hashing schemes require to lock the whole hash table during rehashing, which blocks substantial concurrent queries and incurs dramatic tail latency, the CDF graphs of LEVL, LINP, and libcuckoo have several flat regions that indicate the time they take for each rehashing. As Figure 12 shows, the maximum latency of LEVL is 21.7 sec and is much higher than that of HMEH, 1.6 sec.

5) 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. As Figure 13 shows, HMEH has a higher

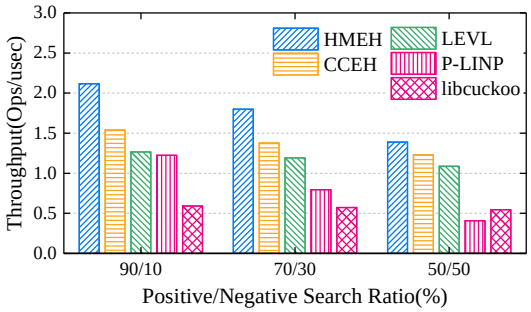


Fig. 13. Average throughput of positive and negative searches.

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. LEVL employs two functions, multi-slots and two-level structure all of which increase search overhead of non-existent keys. Interestingly, the search throughputs of level hashing and libcuckoo hardly decline as the radio of negative search increases. Because they are both based on cuckoo hashing which is optimized for read-intensive workloads. We also investigate that the search throughput of P-LINP drops dramatically with the increase of negative search ratio. The main reason is that P-LINP requires to scan the successive buckets to find the target item when performing a negative search.

6) 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

TABLE II
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

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. And 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 can be negligible. Therefore, directories of HMEH can achieve an instantaneous recovery.

VI. CONCLUSION

In this paper, we propose HMEH, a high-performance variant of extendible hashing for the hybrid DRAM-NVM memory. HMEH places a flat-structured directory in DRAM to obtain faster access, and keeps a radix-tree structure in NVM to rebuild FS-directory upon recovery. Moreover, HMEH leverages cross-KV and delayed flush schemes to reduce the overhead of persist barriers. We also implement concurrent HMEH that efficiently supports multi-thread operations. Experimental results present that HMEH achieves lower latency of querying than state-of-the-art hashing schemes. And concurrent HMEH also shows highest performance and good scalability under mixed workloads of YCSB.

ACKNOWLEDGMENT

This work was supported by National Key R&D Program of China NO.2018YFB1003305, NSFC No. 61832020, No. 61821003, Natural Science Foundation of Shandong Province (No. ZR2019LZH012). We thank anonymous reviewers for their comments. We thank Hong Jiang for the discussions.

REFERENCES

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

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

[3] T. Kawahara, "Scalable spin-transfer torque RAM technology for normally-off computing," *IEEE Design & Test of Computers*, vol. 28, no. 1, pp. 52–63, jan 2011.

[4] M. K. Qureshi, V. Srinivasan, and 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)*.

[5] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and 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)*.

[6] J. Yang, Q. Wei, C. Chen, C. Wang, and K. L. Yong, "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)*. [Online]. Available: http://dl.acm.org/citation.cfm?id=2750482.2750495

[7] F. Xia, D. Jiang, J. Xiong, and 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. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[8] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment (PVLDB)*, pp. 786–797, feb 2015.

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

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

[11] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *9th USENIX Conference on File and Storage Technologies (FAST) (2011)*, pp. 61–75. [Online]. Available: http://www.usenix.org/events/fast11/tech/techAbstracts.html\#Venkataraman

[12] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and 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)*.

[13] P. Zuo and 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)*.

[14] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2018)*. Carlsbad, CA: USENIX Association, Oct., pp. 461–476. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/zuo

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

[16] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing-a fast access method for dynamic files," *ACM Transactions on Database Systems*, pp. 315–344, sep 1979.

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

[18] 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.

[19] "Postgresql," http://www.postgresql.org/.

[20] "Memcached," https://memcached.org/, 2018.

[21] "Redis," https://redis.io/, 2018.

[22] S. Li, P. Dubey, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, and S. Lee, "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)*.

[23] H. Garcia-Molina and K. Salem, "Main memory database systems: an overview," *IEEE Transactions on Knowledge and Data Engineering*, pp. 509–516, 1992.

[24] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: dependably fast multi-core in-memory transactions," in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD) (2017)*.

[25] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and 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)*.

[26] C. D. Philip J Mucci, Shirley Browne and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[27] "Intel corporation intelr 64 and ia-32architectures software developer's manual," http://www.intel.com/content/www/us/en/processors/architectures-software-developermanuals.html.
[28] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys) (2014)*. ACM Press, 2014.
[29] H. Volos, A. J. Tack, and 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)*.
[30] B. Pittel, "Linear probing: The probable largest search time grows logarithmically with the number of records," *Journal of Algorithms*, vol. 8, no. 2, pp. 236–249, jun 1987.
[31] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, may 2004.
[32] H. Mendelson, "Analysis of extendible hashing," *IEEE Transactions on Software Engineering*, no. 6, pp. 611–619, nov.
[33] B. Fan, D. G. Andersen, and 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.
[34] L. R. Johnson, "An indirect chaining method for addressing on secondary keys," *Communications of the ACM*, pp. 218–222, may 1961.
[35] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182.
[36] "Intel64 software developers manual (vol 2, ch 3.2)," 2013.
[37] "Intel architecture instruction set extensions programming reference." https://software.intel.com/sites/default/files/managed/69/78/319433-025.pdf.
[38] "Pmdk," The libvmem library. https://pmem.io/vmem/libvmem/.
[39] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing (SoCC) (2010)*.
[40] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys) (2014)*.
[41] "Libcuckoo library," https://github.com/efficient/libcuckoo.