HDNH: a read-efficient and write-optimized hashing scheme for hybrid DRAM-NVM memory

Junhao Zhu
zhujunhao19@nudt.edu.cn
National University of Defense Technology
China

Kaixin Huang
kaixinhuang@sjtu.edu.cn
ByteDance Inc.
China

Xiaomin Zou
xiaominzou@hust.edu.cn
Huazhong University of Science and Technology
China

Chenglong Huang
huangchenglong16@nudt.edu.cn
National University of Defense Technology
China

Nuo Xu
oun_ux@163.com
National University of Defense Technology
China

Liang Fang
lfang@nudt.edu.cn
National University of Defense Technology
China

ABSTRACT

With high memory density, non-volatility and DRAM-scale latency, non-volatile memory (NVM) brings evolution to storage systems and durable data structures. And Intel Optane DC persistent memory module (AEP), the first commercial product of NVM, shows some features that are different from previous assumptions: higher read latency, lower bandwidth and block access granularity compared with DRAM. It is reasonable to build up hybrid memory to give full play to the complementary advantages of DRAM and NVM. In this paper, we present a read-efficient and write-optimized hashing scheme for hybrid DRAM-NVM memory, named HDNH (Hybrid DRAM-NVM Hashing). Our design can be summarized into three key points. First, we decouple the storage for data and metadata by placing key-value items in non-volatile table for persistence while placing metadata in Optimistic Compression Filter (OCF) to reduce excessive NVM accesses. Second, we design hot table in DRAM to speed up search requests and propose an efficient replacement strategy called RAFL. Third, we develop a fine-grained optimistic concurrency mechanism to enable high-performance concurrent accesses on multi-core systems. Experimental results on the AEP platform show that HDNH outperforms its counterparts by up to 2.9x under various YCSB workloads.

KEYWORDS

hash table, non-volatile memory, Intel Optane DC Persistent Memory, data consistency

ACM Reference Format:
Junhao Zhu, Kaixin Huang, Xiaomin Zou, Chenglong Huang, Nuo Xu, and Liang Fang. 2021. HDNH: a read-efficient and write-optimized hashing scheme for hybrid DRAM-NVM memory. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3473515

1 INTRODUCTION

Emerging Non-Volatile Memory (NVM) technologies, such as PCM [23], ReRAM [26], and 3D-XPoint [31], are drawing substantial attentions from both academia and industry. With high performance and instant recovery, it is attractive for hash tables to persist data and operate directly on NVM. In the past decade, researchers have proposed a lot of NVM-friendly hashing schemes [12, 18, 20, 34, 36], such as Level Hashing [36], Path Hashing [35], PFHT [9] and CCEH [20]. And these indexes put many efforts in write optimizations, consistency guarantee, or efficient resizing. PFHT is a variant of bucketized cuckoo hashing designed to reduce write accesses to PCM as it allows only one cuckoo displacement to avoid cascading writes. Path hashing is similar to PFHT in that it uses a stash although the stash is organized as an inverted binary tree structure. Level hashing consists of two hash tables organized in two levels. The top level and bottom level hash tables take turns playing the role of the stash. Unlike Path hashing and PFHT, Level hashing guarantees constant lookup time. However, these researches mentioned above cannot be simply adapted to the new features of Intel Optane DC Persistent Memory (AEP). As demonstrated in the recently released technique report [30], AEP has several unique performance features: (1) it exhibits 3x read latency and similar write latency compared with DRAM. (2) its read and write bandwidth are 3x/6x lower than that of DRAM. (3) The granularity disparity between CPU caches and AEP (64 vs 256 bytes). According to these new features, there is still a certain gap between NVM and DRAM. Specifically, we found that the current researches in persistent hashing schemes have following issues:

1. Multiple accesses in NVM for probing data. Existing hashing schemes usually use multi-slot bucket to resolve hash collisions. And searching a bucket typically requires a linear scan of the slots, which is a major source of NVM reads, and especially so for long keys. One or multiple buckets in NVM have to be searched to find out if a key exists [25]. And this is just the indexing process of searching a key without considering block access granularity. Although DRAM is used to accelerate access in HMEH [34], it is still necessary to traverse each slot of the corresponding bucket in NVM, hence leading to extra NVM access overhead.

2. Hotspot issue for searching. In real industrial environment, there is often some hot data that are frequently accessed in highly-skewed workloads. Alibaba [4] observes that 50% (daily cases) to 90% (extreme cases) of accesses only touch 1% of total items, which shows that the hotspot issue becomes unprecedentedly serious in in-memory key-value store. If NVM is accessed every time for these hot data, it will cause longer access latency and waste NVM bandwidth. Therefore, DRAM cache should be employed to store hot dataset in DRAM, where replacement strategy plays an important role. Only a few hashing schemes focus on the caching design. Rewo [12] is a recent work that provides a cached table in DRAM to improve read performance. However, it simply uses LRU algorithm as the replacement strategy. We consider that it has two drawbacks: 1) LRU list consumes a lot of memory space, and 2) LRU cannot cope with random-access workloads efficiently.

3. Coarse-grained lock for concurrency control. Most prior schemes overlook the impact of concurrency control. Locking techniques have been widely used to control concurrent accesses to shared resources. CCEH [20] uses bucket-level lock for concurrency, which incurs unnecessary NVM access for read locks. Although coarse-grained locks protect the hash table, they also prevent concurrent accesses and limit the scalability. In this way, the heavyweight concurrency control can easily exhaust NVM's limited bandwidth. These issues call for new designs that can achieve high scalability while guarantee the concurrency correctness.

To address the limitations of existing NVM-oriented hashing schemes, we present a novel read-efficient and write-optimized hashing scheme called HDNH (Hybrid DRAM-NVM Hashing). Its target is to fully leverage the merits of both DRAM and NVM. Similar to level-hashing, we also employ a dynamic two-level index structure in hybrid DRAM-NVM memory. In NVM, HDNH stores the non-volatile table; while in DRAM, a hot table is provided for caching hot items. To reduce the heavy NVM read overhead caused by key probing, we design an efficient optimistic compression filter in DRAM to filter out unnecessary accesses into NVM. The contributions of this paper are summarized as follows.

Optimistic Compression Filter for indexing procedure. In order to reduce excessive NVM access in indexing procedure, we propose the Optimistic Compression Filter (OCF) in DRAM. OCF is a data structure that uses fingerprints to filter out unnecessary NVM reads. We properly configure data and metadata into hybrid memory and reduce NVM access using OCF as much as possible. Specifically, when searching a key, we first access the OCF to compare fingerprints in it with the fingerprint of the search key. If they match, we then fetch the complete key-value pair in the corresponding position in NVM. Otherwise, the key does not exist in the hash table. In this way, OCF can minimize reads into NVM during data probing and the bandwidth of NVM can be remarkably saved.

Hot table for caching frequently-acessed dataset. In order to reduce frequent NVM read for hot dataset, we keep hot table as a scalable non-volatile table copy in DRAM to speed up search requests. Besides, we devise an efficient synchronous write scheme between non-volatile table and hot table to mask the data synchronization overhead. The hot table uses our proposed RAFL algorithm as its replacement strategy. This replacement strategy inherits the advantages of the LRU algorithm and has much smaller overhead. Unlike LRU, RAFL is friendly with random read. As a result, we can accelerate the read performance of hot data and efficiently utilize the bandwidth of DRAM.

Fine-grained optimistic control for concurrency. In order to solve the performance degradation caused by coarse-grained lock and make full use of the hybrid DRAM-NVM bandwidth, we propose fine-grained optimistic concurrency control to satisfy the high concurrent throughput requirements of multi-core systems. Instead of using bucket-level lock, we apply fine-grained lock for each slot, which consists of one bit lock and a version number. And we implement lock-free search to avoid additional NVM writes for acquiring and releasing read locks. Hence, the concurrent performance of HDNH can be fully utilized.

System Implementation. We implement HDNH and conduct extensive evaluations on the AEP hardware platform. Experimental results show that HDNH can deliver superior performance, high scalability and good space utilization.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 presents the design and implementation of HDNH. Section 4 presents performance evaluation. Finally, section 5 concludes this paper.

2 BACKGROUND AND MOTIVATION

2.1 Intel Optane DC Persistent Memory

Intel Optane DC persistent memory module (AEP), the first commercial product of NVM, is available on the market [13]. It provides excellent characteristics such as byte-addressability, persistence, high capacity, low cost and high performance. A single NVM module offers 512 GB capacity in the maximum, which is attractive for in-memory applications. It currently supports two modes including Memory Mode and App Direct Mode. The former is similar to DRAM, which means NVM is used as a large volatile memory device with conventional DRAM as its cache. In App Direct mode, NVM is used as a persistent storage device. It provides more flexibility and persistence guarantee because applications can ultilize byte-addressable space of both NVM and DRAM.

The current NVM should be equipped in a system with DRAM memory, and they share the integrated memory controller and CPU cache components. It is difficult to guarantee that data is persisted in NVM until all the relevant cachelines have been flushed to NVM. The flush procedure is conducted by either explicitly exploiting flush instructions (i.e., CLFLUSH, CLFLUSHOPT or CLWB) [7] or normal cache evictions. Furthermore, writes to NVM may also be reordered if we do not use fences (MFENCE or SFENCE) to avoid unacceptable reordering. In this situation, the application must reasonably issue fences and cacheline flushes to ensure data consistency. CLFLUSH and CLFLUSHOPT will evict the cacheline that is being flushed, while CLWB does not. Thus, CLWB can give better performance. After a cacheline is flushed, it will reach the asynchronous DRAM refresh (ADR) domain to it form the Asynchronous DRAM Refresh (ADR) domain. Every cache line that reaches the ADR is guaranteed to be persisted, even in the event of a power loss [14].

According to previous assumptions [3, 17, 27, 32], NVM has asymmetric read and write latency, and write is slower than read. However, AEP [14] shows some interesting features that are different from the assumptions. Compared with DRAM, NVM has 3x read latency and similar write latency. In the meantime, the read and write bandwidths of NVM achieve 1/3 and 1/6 of DRAM. The read latency as seen by the software is often higher than write latency when it comes to AEP. The reason can be explained by the fact that writes commit once the data reaches the ADR domain at the memory controller instead of reaching AEP media. Different from write operation, read operation has to touch AEP media upon cache missing, which is very common in hash tables because of inherent randomness. In addition, there will be read amplification problem for persisted hash tables because Optane AEP is accessed in block granularity (256 bytes). Thus, it is as important as optimizing write to reduce NVM reads access for better performance [22, 24, 28], which is also the first principle that guides our scheme design.

2.2 Hashing Index Structure In NVM

The hashing-based indexing structures are widely used in key-value stores [10, 11, 15, 19, 29] and main memory databases due to support constant-complexity point query operations. For the purpose of efficiently adapting to NVM, several hashing-based structures [5, 35, 36] have been proposed. Level-hashing [36] is a recent representative design among many NVM-only hashing schemes.

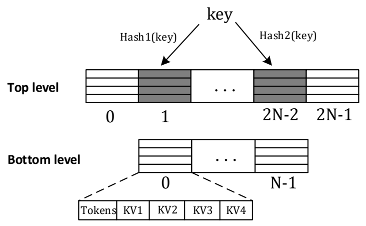


Figure 1: Architecture of level hash.

Level hashing consists of two tables organized in two levels as shown in figure 1. The top level is addressable, which uses two hash functions to locate a key. The bottom level is hidden as a stash. When the bottom level overflows, the items stored in the bottom level are rehashed to a 4x larger hash table and the new hash table becomes the new top level, while the previous top level hash table becomes the new bottom level stash. The items in the previous top level are reused without rehashing. In this way, Level hashing guarantees constant lookup time. To mitigate the shortage of space in bottom-level stash, Level hashing proposes to use the bottom-to-top cuckoo displacement that evicts records from the bottom level stash to the top level hash table. However, Level hashing is purely built in NVM and faces the performance bottleneck we mentioned before. First, bottom-to-top eviction sacrifices excessive NVM access in critical path to improve the load factor and postpone rehashing, which is not worth the loss. Furthermore, bucket-level locking for concurrency control incurs additional NVM access to acquire/release read locks, which also pushes bandwidth consumption towards the limit.

2.3 Hashing Index Structure In DRAM-NVM

The new features brought by AEP have inspired us to rethink the bottlenecks of previous NVM-optimized hash table designs. And we firmly believe that NVM will not replace DRAM overnight. Instead, NVM will coexist with DRAM for a foreseeable future. Although a large number of previous studies have improved hash tables for persistent memory, only a few have attempted to develop hashing schemes based on hybrid memory, such as Rewo [12] and HMEH [34].

There are two tables in Rewo [12], with one in NVM space and the other in DRAM space. Persistent table is mainly used to serve write requests and keep key-value items durable with finegrained consistency guarantee. And cached table is totally used to serve search requests since it keeps a copy of data items in persistent table. If cached table is not overloaded, each value for a request key can be found in hot table. Otherwise, hot table only stores the hot dataset of persistent table and uses LRU as its replacement strategy. HMEH [34] stores key-value items in NVM for directly persisting and places flat-structured directory in DRAM to improve overall system performance. Due to reduce extra NVM accesses to the directory in the critical path, the searching latency of HMEH is much lower than previous schemes.

HMEH has no DRAM cache, which results in NVM accesses to read the hot data every time. Although Rewo has a hot table in DRAM for hot dataset, it cannot be dynamically adjusted. And Rewo uses LRU algorithm as its replacement strategy, which is unfriendly for random access. Furthermore, both of Rewo and HMEH need to scan the slots of bucket for searching in NVM, hence leading to unnecessary NVM reads.

3 HDNH DESIGN AND IMPLEMENTATION

3.1 Overview

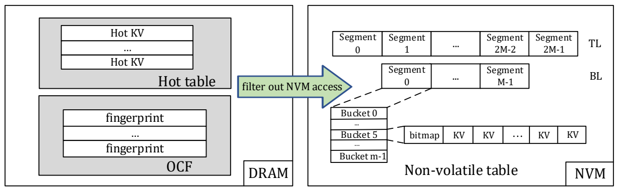


Figure 2: Architecture of HDNH.

In this section, we present the design and implementation of HDNH. As shown in figure 2, we draw on the structural characteristics of Level hashing and set up a two-level structure in NVM called non-volatile table. Instead of buckets as the hashing unit in each level, we choose a larger-granularity, segment [20, 33], as the hashing unit. In order to increase the load factor, we locate segments first and then locate buckets using 2-cukoo strategy. The top level (TL) has 2M segments and the bottom level (BL) has M segments. A segment consists of m buckets. In a bucket, *bitmap* reserves one bit for each slot, to indicate whether the corresponding slot stores a valid record. And we set the size of a bucket to 256-byte, which can better utilize the block access granularity of NVM [9]. We place Optimistic Compression Filter (OCF) and hot table in DRAM space. OCF is used to filter out excessive NVM access and its core elements are fingerprints (see section 3.2). Hot Table is designed for holding hot dataset in DRAM to achieve high read performance. We also develop synchronous write mechanism for hot table to mask the additional overhead.

3.2 Optimistic Compression Filter

There have been many studies on filters [1, 2, 16] in the past. The main concern of them is how to use filters to reduce disk read latency and improve search performance. Their common idea is to place some metadata in DRAM to filter out excessive disk accesses. As we mentioned before, the read and write performance of NVM still has a certain gap compared with DRAM. Therefore, we can also filter out excessive NVM accesses using filter. Inspired by some prior NVM-only indexing schemes [18, 21], HDNH uses fingerprints [21] as its filter. Fingerprints are one-byte hashes of keys for predicting whether a key possibly exists, which can utilize the least significant byte of the key's hash value.

Based on the benefit of fingerprints, we propose Optimistic compression Filter (OCF). The advantage of OCF is that it provides an efficient indexing mechanism for hash index without occupying much DRAM space. The indexing procedure in previous schemes such as level hash [36] and rewo [12] is described as follows: (1) hash the requested key to a candidate bucket; (2) only check the valid slots whose corresponding bits in *bitmap* and then find the matched key; (3) return if a slot match the searched key or restart from (1). The second step generates a large amount of NVM read overhead during probing because the indexing thread has to access the candidate buckets in NVM to match the searched key. However, with OCF, our hashing scheme can avoid a lot of uneccessary NVM reads because many key matching operations are resolved in DRAM. If one fingerprint is not matched with a key's LSB, then there is no need to probe the corresponding slot in NVM.

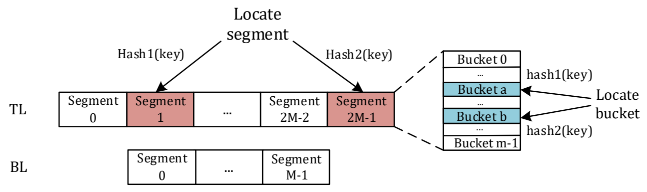


Figure 3: The storage structure of OCF.

bitmap	opmap	version	FP	FP	...	FP

Figure 4: Data Structure of OCF bucket.

As shown in the figure 3, OCF is also a two-level data structure and corresponds with non-volatile table one by one. As shown in figure 4, an OCF bucket is composed by *bitmap*, *opmap*, *version* and fingerprint. *Bitmap* reserves one bit for each slot, which indicates whether the corresponding slot stores a valid record, with 0 meaning empty and 1 meaning valid. *Opmap* also reserves one bit for each slot, which is used to record whether a slot is being occupied by a thread, with 0 meaning free and 1 meaning busy. *Version* is set to 6 bits for each slot and it is used for concurrency control. We utilize hash functions to locate segments first and then locate buckets using 2-cukoo strategy, which means that we have four candidate buckets for each level. For a two-level structure, there are a total of eight candidate buckets. This strategy not only increases the load factor of non-volatile table, but also reduces the frequency of rehashing.

With OCF, we no longer need to index data like previous hashing schemes [35, 36]. In our indexing procedure, we first locate candidate buckets and then traverse the buckets corresponding to the requested key in OCF. Notice that this process is performed in DRAM. Only if the fingerprint in OCF is matched, we need to access non-volatile table in NVM to match the key. We don't need to use hash function again to locate the candidate bucket in non-volatile table because the structure of OCF and non-volatile table correspond to each other. In other words, the slot position in non-volatile table is known. If the key in non-volatile table is matched successfully, we return the value of the key. Otherwise, we continue to check the next fingerprint. With OCF, our hashing scheme is able to serve most probe operations in DRAM, reducing unnecessary data accesses and key matches in NVM. On the other hand, the fingerprint of each key is merely 8 bits while *bitmap*, *opmap* and *version* occupies 8 bits for each entry. An OCF entry only occupies 2 bytes in total. In real workloads, the size of a key-value item is usually 16 to 32 bytes [18]. Therefore, an OCF entry is insignificant comparing with the size of a key-value item.

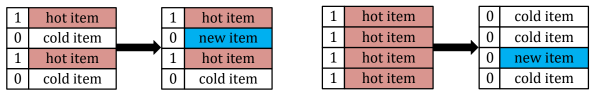
3.3 Hot Table

In order to further improve the read performance of hybrid memory hash scheme, we come up with the idea of building a cache in DRAM. By placing hot key-value items in the hot table in DRAM, we can still decrease the reads into NVM for skewed read workloads even if we have OCF.

bitmap	opmap	hotmap	version	Hot KV	Hot KV	...	Hot KV

Figure 5: Data Structure of hot table bucket.

The structure of hot table is basically the same as OCF. And we place hot items instead of fingerprints in each slot as shown in figure 5. We add a hotmap field in each bucket, which reserves one bit for each slot. The hotmap bit indicates whether the corresponding slot stores a hot item, with 0 standing for a cold item (i.e., the item has not been searched since it was added) and 1 representing a hot item (i.e., the item has been searched after it was added). And the hot table changes the hotmap bit from 0 to 1 when a thread searches the corresponding item. The number of slots for each bucket in hot table is fewer than the non-volatile table. In order to improve the cache locality and reduce the overhead caused by cache missing, we optimize the query procedure for hot table. Instead of using 2-cuckoo strategy for both segment and bucket like OCF, we set the hash function of locating segment and bucket to one. There is one candidate bucket for each level in the hot table. Fewer slots are traversed when searching for candidate buckets in the hot table. If the searched item does not exist in the hot table, the overhead of cache miss can be reduced. Otherwise, read performance can be sped up when the requested key-value item is in the hot table.



(a) Cold replacement. (b) Hot replacement.

Figure 6: Two replacement situation of RAFL.

When the amount of data in non-volatile table exceeds the capacity of hot table, the hot table uses a data-conscious algorithm called RAFL to achieve efficient cache replacement. The hot table stores two types of items including hot item and cold item. When we insert an item to the corresponding bucket which is full, we first traverse hotmaps of the bucket to find a cold item. If a cold item is found as shown in figure 6(a), we delete it and insert the new item into the slot. And if the items in the bucket are all hot items as shown in figure 6(b), we use random method to delete an item and insert the new item into the slot. After that we set all hotmaps of the bucket to 0. Although some hot items may be evicted because of the random replacement in this procedure, the items can be inserted to the hot table again when these items are searched next time. It is noted that all hotmaps in the bucket are set to 0 in the end of hot replacement. This is because we should prevent some items from occupying the hot table for a long time and wasting valuable space resource. Compared with the LRU algorithm, RAFL can also achieve hot and cold data replacement while the overhead is much smaller. When it comes to random access, RAFL will not cause too much additional overhead like LRU. So it is more friendly for random access.

3.4 Synchronous Write Mechanism

Although hot table placed in DRAM can improve the read performance, it also brings extra write overhead in the critical path (for in-DRAM updates). To address this challenge, we propose synchronous write mechanism to minimize the write overhead of hot table by simultaneously writing key-value items to hot table and non-volatile table.

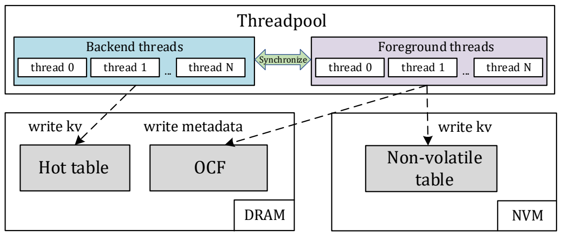


Figure 7: Procedure of synchronous write mechanism.

As shown in figure 7, we divide the working threads into foreground threads and background threads in the actual working environment to support multi-core system. And every write operation actually requires two threads, a foreground thread and a background thread. The work of foreground thread is to write the item into non-volatile table and modify the metadata of the item in OCF. The work of background thread is to write the item into hot table. And the two threads are performed synchronously, which utilizes a synchronous write signal called *sync_write_signal* to communicate with each other. When the foreground thread receives a write request, it initiates *sync_write_signal* to incompletion and then shares it with the background thread. The background thread executes write request in hot table. When the execution succeeds, the background thread will set *sync_write_signal* to completion. After successfully executing the write operation in non-volatile table, the foreground thread will check the status of *sync_write_signal*. And the foreground thread will return after finding that *sync_write_signal* is set to completion. Finally, the two threads will be returned to the thread pool.

3.5 Time-Efficient Read Mechanism

Both hot table and OCF bring benefits to search procedure due to the reduction of a large number of NVM reads. Read threads can achieve fast access to key-value items that are cached in hot table, which accelerates the frequent read of hot dataset and reduces the occupation of NVM bandwidth. In the case of cache missing, we can also use fingerprints in OCF to filter out most of key match operations. The read procedure in HDNH is described in figure 8: (1) search the requested item in hot table and return if successfully (2) match the fingerprint of the key with fingerprints in the candidate bucket if the requested item can't be found in hot table (3) check the key in non-volatile table if the fingerprint is matched successfully in OCF. If the checked key is the requested one, the search thread returns. Otherwise, the search thread will continue to execute step (2) until the key-value item is found. We can divide search procedure into two situations according to whether hot table can contain all data of the non-volatile table.

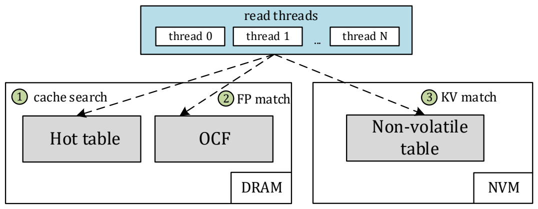


Figure 8: Procedure of Time-Efficient mechanism.

Hot table has not been overflowed. It means that hot table contains all the data in non-volatile table. Instead of searching the key-value item in non-volatile table, read thread can complete all of search requirements in hot table. As mentioned before, the hot table is adjustable and it can synchronize with non-volatile table, which makes it possible to ensure that hot table will not overflow for a period of time. On the one hand, the read procedure during this period can make full use of the advantage of cached table. On the other hand, frequent read of hot data can completely get rid of NVM access. As a result, the NVM bandwidth can be saved for insertion and update operations that require frequent NVM read and write.

Hot table has been overflowed. Although hot table can be dynamically adjusted with non-volatile table, the total capacity of it is less than non-volatile table. Therefore, each read request needs to first search in hot table and return if hitting. If the key-value item is not in hot table, the read thread will match the fingerprint of the key with fingerprints in OCF. Only if the fingerprint in the candidate bucket is matched successfully, the thread will match key in non-volatile table.

3.6 Fine-grained Optimistic Concurrency

In order to meet the high concurrency performance of the multi-core system and make full use of the bandwidth of hybrid DRAM-NVM memory, we propose a fine-grained optimistic concurrency mechanism using atomic write to reduce NVM access. Specifically, we set *opmap* and *version* for each slot of hot table and OCF. The *opmap* (1 bit) is used to indicate whether a slot is being written by a write thread, with 0 meaning busy and 1 meaning free. *Version* (6 bit) is used to detect whether there are conflicts between read and write threads in the corresponding slot. And they can be modified atomically using compare-and-swap (CAS) instruction. When a write thread needs to write a slot, it first changes *opmap* from 0 to 1 using an atomic write and then the thread executes its operation. After finishing its write operation, the thread changes *opmap* from 1 to 0 and incrementing the *version* by one using an atomic write. When it comes to the read thread, it will take a snapshot of *version* and check the *opmap* of the slot first. If the slot is locked by a write thread, the read thread will wait until the slot is free. And then it can read the slot without any lock. After completing the read operation, the thread will read the *version* again. If the *version* number is changed, the read thread should retry the read operation again because the item might be modified by a concurrent write thread. If not, the thread returns successfully. The following is the specific illustration for insertion, update and delete.

The foreground insert thread will perform the following operations: (1) the thread first finds an empty slot, as shown in figure 9(a); (2) insert the fingerprint to the OCF entry and insert the item to the slot, which is shown in figure 9(b); (3) make the slot valid in figure 9(c); (4) release the slot in figure 9(d). And the background insert thread does similar work in hot table: (1) find an empty slot and lock it; (2) insert the hot item to the slot; (3) modify the *bitmap* of the slot atomically; (4) release the lock.

The foreground update thread will perform the following operations: (1) find the old item (OKV) and the old fingerprint (OFP), then lock an empty slot in the bucket, as shown in figure 10(a); (2) insert the fingerprint (NFP) to the OCF entry and insert the new item to non-volatile table, which is shown in figure 10(b); (3) modify the bitmap of the old item and the new item atomically in figure 10(c); (4) release the lock of the new item in figure 10(d). And the background update thread first checks whether the old item is in hot table. If the old item can be found, the thread updates the old item in place to complete the synchronous update. And if the old item cannot be found in the hot table, the thread inserts the updated item to hot table.

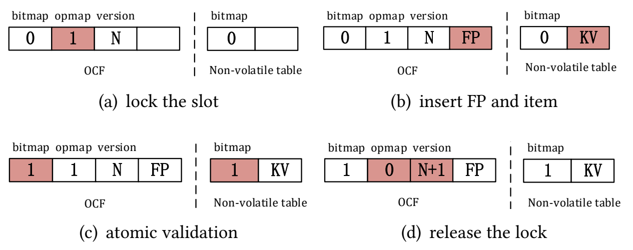


Figure 9: Concurrency control of the foreground insertion thread.

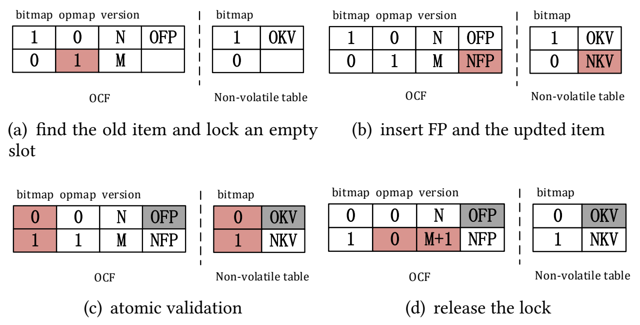


Figure 10: Concurrency control of the foreground update thread.

Compared with other operations, the operation of foreground deletion thread is simple. The thread changes the *bitmap* of the item from 1 to 0, which invalidates the item. And the background deletion thread first checks whether the item is in hot table. If the item can be found, we delete the item. If not, the thread still return successfully because the item is not in hot table.

3.7 Recovery

The fast recovery requires the guarantee for crash consistency. And it is very important for hash table. Since DRAM is volatile, the OCF and hot table in DRAM will disappear after a normal shutdown or a system crash. Below we illustrate the recovery of our scheme.

Recovery after a normal shutdown. In this situation, we need to recover OCF and hot table in DRAM. Bitmaps are persisted to non-volatile table as backup and fingerprints can be calculated by keys. Restoring the OCF only needs to traverse the buckets in non-volatile table to obtain bitmaps and fingerprints of all items. For hot table, we still need to traverse all the buckets in non-volatile table and insert the items to hot table. In order to speed up recovery as much as possible, we can merge the reconstruction procedure of the two data structures. In this way, non-volatile table just needs to be traversed once and the two data structures can be reconstructed quickly. In order to speed up the recovery procedure, we use multiple recovery threads [8] and divide buckets into independent batches for recovery threads to execute.

Recovery after a system crash. After the system crashes, we first use *resizing_flag* to determine whether the system is in the resizing state. If non-volatile table is not in the resizing state, we directly rebuild the OCF and hot table. The procedure of restruction is the same as the recovery after a normal shut down above. For the crash consistency in resizing, we first check the current level of non-volatile table through *level_number* persisted in NVM. The *level_number* '2' indicates that non-volatile table is starting to rehash. There are two possible reasons for this situation. One is that non-volatile table has not successfully applied for a new level. The other is that non-volatile table has applied for a new level but the pointer of top level has not yet pointed to the new level. To deal with this situation, the recovery thread applies for the new level again and let the pointer of top level point to the new level. And the *level_number* '3' indicates that non-volatile table is rehashing. HDNH records the indexes of segment and bucket in NVM when successfully rehashing items in a bucket. To recover after a system crash, resizing threads can read the records of indexes and continue rehashing the next bucket. When non-volatile table completes resizing, the system returns to two-level structure. Then multiple recovery threads rebuild OCF and hot table in DRAM, which divides buckets into independent batches for recovery threads to execute. The procedure of restruction is the same as the recovery after a normal shut down above.Non-volatile table just need to be traversed once to reconstruct the two data structures.

4 PERFORMANCE EVALUATION

4.1 Experimental Setup

All of our experiments are conducted on a single socket, 32-core server with 32KB/1024KB/32MB L1/L2/L3 Caches. And 192GB of DRAM (6 × 32 GB/DIMM) and 1.5TB of DCPMM (6 × 256 GB/DIMM) are equipped in the server. The Optane DC PMMs are configured in the App Direct mode and mounted with ext4-DAX file system. We use ubuntu Linux with kernel 5.4.0 and PMDK 1.7 to run experiments. All the code is compiled using GCC 10.2.0 with all optimization enabled. Threads are pinned to physical cores. We apply CLWB for cache line flushes, which is more efficient than CLFLUSH and CLFLUSHOPT.

In our experiments, we compare the following hashing-based index scheme in NVM:

• **PATH** : Path hashing [35] is a static hashing designed to reduce write accesses. And it uses a stash although the stash is organized as an inverted binary tree structure. Its search time depends on log scale, i.e., O(logB), where B is the number of buckets. For achieveing maximum load factor, we set the reserved level to 8 as suggested by the authors.

• **LEVEL** : Level hashing [36] consists of two hash tables organized in two levels. The top level and bottom level hash tables take turns playing the role of the stash. The level hashing uses slotgrained reader-writer lock for queries and a global resizing lock for resizing.

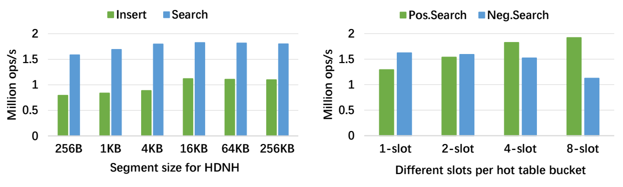
• **CCEH** : CCEH [20] is composed of segments and uses a directory to locate these segments. And CCEH uses linear probing to improve the load factor. It also supports dynamic resizing through segment splitting and possible directory doubling. For fair comparison, CCEH uses 16KB segments and 64-byte buckets. And we set the number of linear probing is 4.

• **HDNH** : HDNH is our indexing scheme, a read-efficient and write-optimized hashing scheme for . HDNH uses 256-byte buckets in order to consist with the granularity of AEP. And we set the slot number of each non-volatile bucket to 8.

In our experiments, we use YCSB [6] to generate microbenchmarks to evaluate the throughput of different hashing schemes including single-threaded and multi-threaded versions. In addition, we tune the zipfian distribution [6] parameter s in YCSB to generate different workloads for skewness. For insertion, we first preload the hash table with 20 million items, then 180 million inserts are excuted in the hashing tables. There are 200 million items in the hash table after completing insertion. For search operation, we run positive search (search for existent keys) and negative search (search for non-existent keys). Concretely, we execute 180 million positive searches and negative searches in the 200-million-item hash table. Similarly, we run 180 million deletions in the hash table. And we use 16-byte keys and 15-byte values for all experiments, which are pre-generated by the benchmark before our testing.

4.2 Sensitivity Analysis Of HDNH Design

In order to get the optimal performance of our proposed scheme, we set up several experimentals to test the optimal configuration. Firstly, we test the performance of single thread insertion and search throughput under different segment sizes to get the best segment size of HDNH. Specifically, the segment sizes are varied from 256B to 256KB as shown in figure 11(a). We find that insertion throughput increases from 256B to 16KB. It is because that the frequency of rehashing decreases with the increase of segment sizes. But insertion throughput decreases from 16KB to 256KB, which causes by the blocking of large segment sizes resizing. And the search throughput improves when segment sizes increase from 256B to 16KB. And the search throughput basically no longer changes after the segment sizes are over 16KB. As the experiment result shown, we set the segment sizes to 16KB so that we can obtain optimal performance.



(a) Different segment sizes. (b) Different hot table slots.

Figure 11: Throughput of different configuration of HDNH

In order to get the best slot number of hot table bucket, we test positive search and negative search throughput with different slots per hot table bucket as shown in figure 11(b). The positive search throughput improves with the slot number growing. The reason is that more data searchs hit in hot table with bigger slot number. At the same time, it degrades the negative search performance due to increase miss overhead of hot table. Considering the above experimental results, it is reasonal to set the slot number of hot table bucket to 4 for balanced performance.

4.3 Access Skewness For Hot Table

HDNH uses hot table to speed up search requests for hot dataset. Hot item is stored in hot table for fast access while persisted in non-volatile table. We compare HDNH with other schemes to evaluate the effect of hot table. And we compare HDNH(RAFL) with HDNH(LRU) to evaluate the effect of our proposed replacement strategy. Figure 12 shows the throughput of different schemes under single thread when the zipfian parameter s varies. We tune s from 0.5 to 1.22, which means that the hotspot issue in workloads become more severe. As can be seen, the performance improvement in both LEVEL and CCEH is not obvious as s increases, since they lack hot-aware considterations. In contrast, the throughput of HDNH significantly improves as s increases, which is attributed to our proposed hot table and time-efficient mechanism. When s is in the range of [0.9,1.22], where hotspot issue is severe, HDNH(RAFL) achieves better performance than HDNH(LRU). Specifically, RAFL outperforms LRU by 1.23x for daily scenarios when s is 0.99. When s is set to 1.22 for extreme cases, RAFL outperforms LRU by 1.4x. This is because RAFL is based on two state bits rather than a linked-list and it has smaller maintenance overhead than LRU.

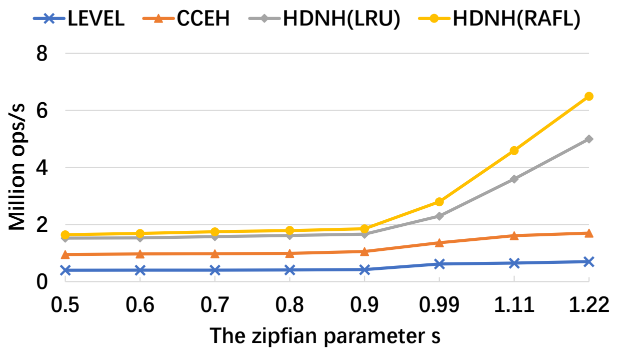


Figure 12: The throughput of different access skewness under single thread.

4.4 Single-thread Performance

For directly reflecting the performance characteristics of each hashing scheme, we first conduct single-thread throughput experiment to verify the advantages of HDNH. As shown in figure 13, we test single-thread Performance of the four hashing schemes.

When it comes to the insertion throughput, HDNH can outperform CCEH/LEVEL hasing by 1.9x/3.7x. It is attributed to OCF reducing NVM accesses by placing metadata in DRAM. In this way, insertion operation can accelerate the procedure of locating an empty slot. We can see that level hashing achieves much lower performance because more NVM reads and frequent lock/unlock operations. It also requires rehashing that blocks insertion operation. For search, we test positive search and negative search throughput. Specifically, HDNH achieves significant improvement for positive search, which is 1.57x/4.33x faster than CCEH/LEVEL hashing. It is attributed to hot table reducing a part of NVM reads for hot data. And hot data can be found in DRAM instead of NVM. For negative

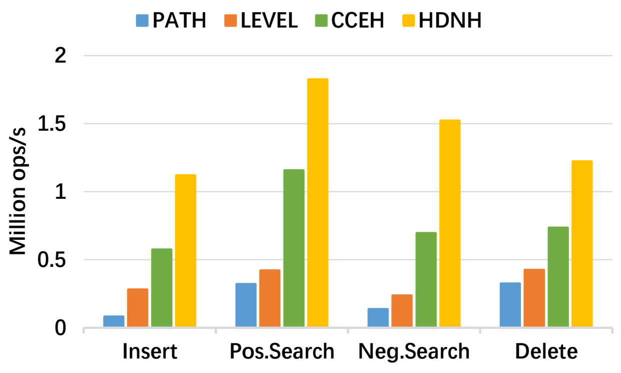


Figure 13: Single-thread performance.

search, HDNH can outperform CCEH/LEVEL by 2.2x/5.6x. It also gets benefits from OCF so that HDNH can read metadata from DRAM and reduce miss overhead. Similarlly, HDNH is 1.7x/2.9x faster than CCEH/LEVEL for deletion.

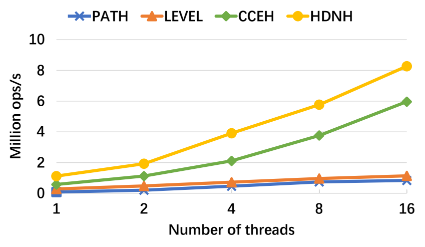
4.5 Concurrent Performance

In this section, we compare our proposed HDNH with other concurrent performance hasing schemes including CCEH, LEVEL and PATH. We evaluate concurrent performance under an increasing number of threads from 1 to 16. And we use YCSB [6] to generate three workloads (180 million): 100% insert, 100% search, and a mixed workload that consists of 50% of insert and 50% of positive search operations.

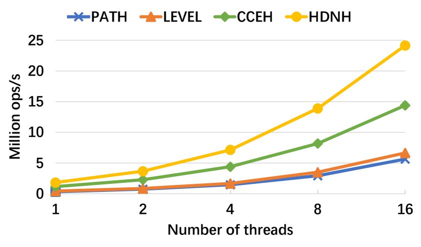
The throughputs of 100% insert workload under a varying number of threads are shown in figure 14(a). Level hashing and Path hasing suffers from poor scalability and heavy rehashing, which causes lots of additional flushes in NVM and blocks insertion. Although CCEH dynamically expands hash table sizes, it still tolerates worse scalability due to the coarse-grained segment lock. HDNH uses fine-grained optimistic concurrency control so that insertion can be executed concurrently. And the improvement of HDNH over others even reaches 1.6x-6.9x.

Figure 14(b) shows average throughputs of 100% search workloads. Being attributed to fine-grained optimistic concurrency control, HDNH outperforms CCEH/LEVEL by 1.9x/4.4x. Level hasing and Path hashing uses coarse-grained locks to protect the hash table, but they also prevent concurrent accesses and limit the scalability. The search throughput of CCEH falls behind due to acquire or release read locks, which generates large amount of NVM writes. Instead of using pessimistic like CCEH, HDNH utilizes optimistic lock and version number to achieve efficient and safe concurrency control. Similarlly, figure 14(c) shows average throughputs of the mixed workloads. HDNH still performs best among all the schemes and outperforms CCEH/level hashing by 1.4x/4.3x.

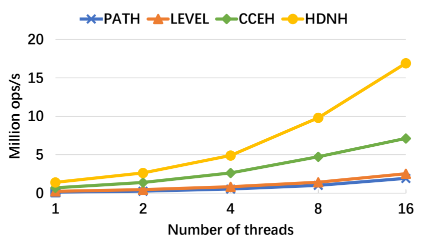
We also measure the tail latency of YCSB-A [6] (high contention case) to check the read starvation problem. Figure 15 illustrates its cumulative distribution functions (CDF) under 16 threads. The maximum latency of CCEH is up to 2.96x higher than that of HDNH (19.2 ms vs. 56.8 ms). And the maximum latency of LEVEL is up to 4.86x higher than that of HDNH (19.2 ms vs. 93.3 ms). It is because



(a) 100% insert workload.



(b) 100% serach workload.



(c) Mixed workload.

Figure 14: Throughputs under different workloads for concurrency performance

coarse-grained lock used by CCEH and LEVEL leads to frequent lock contention and prevents the improvement of concurrent performance. In contrast, HDNH uses OCC to support fine-grained locking and opportunistic concurrent reads/writes, which reduces the contention overhead.

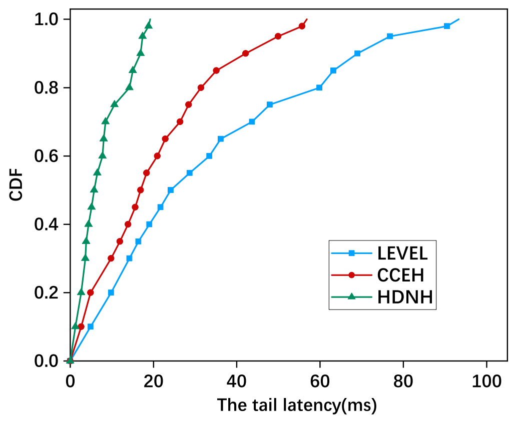


Figure 15: The tail latency CDF under YCSB-A workload.

4.6 Recovery

For reducing service downtime, it is important for hashing table to recover after a system crash. In this experiment, we evaluate the recovery time of HDNH. Specifically, we first preload certain amount of items and then power off the system. Finally, we measure the time HDNH needs to recover the sysytem under single thread. As we mentioned before, we can mainly divide the recovery procedure into two parts including recovering OCF and hot table. Table 1 shows the recovery time of HDNH under different data sizes from 2 million to 200 million. We can see that the recovery time only takes 8.3 ms when the data size is 2 million. And when the data size is up to 200 million, it takes 435.1 ms to recover the system, which is insignificant comparing with the whole excution time. So it is efficient for HDNH to achieve instant recovery.

Table 1: Recovery time of HDNH for different data sizes.

Data size	2 million	20 million	200 million
OCF recovery time(ms)	0.8	9.1	60.8
Hot table recovery time(ms)	6.7	48.6	351.2
HDNH recovery time(ms)	8.3	60.5	435.1

5 CONCLUSION

In this paper, we present a read-efficient and write-optimized hashing scheme for hybrid DRAM-NVM memory, named HDNH. HDNH persists key-value items in non-volatile table while metadatas are placed in OCF for fast access. And HDNH uses hot table in DRAM to speed up search requests. Moreover, we optimize our scheme to satisfy the high concurrent throughput requirements of multi-core systems. Experimental results show that HDNH delivers superior performance and high scalability under various YCSB workloads.

ACKNOWLEDGMENTS

This work is supported in part by the National Key Research and Development Program of China (Grant No.2018YFB1003304), National Natural Science Foundation of China (Grant No.61832007), and the Research Foundation from NUDT (Grant No.ZK20-02). Thanks the calculation support of State Key Laboratory of High Performance Computing, National University of Defense Technology.

REFERENCES

[1] Mahmood Ahmadi and Stephan Wong. 2011. A cache architecture for counting bloom filters: Theory and application. *Journal of Electrical and Computer Engineering* 2011. https://doi.org/10.1155/2011/475865
[2] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An improved construction for counting Bloom filters. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4168 LNCS, 684–695. https://doi.org/10.1007/11841036_61
[3] M. F. Chang, J. J. Wu, T. F. Chien, Y. C. Liu, T. C. Yang, W. C. Shen, Y. C. King, C. J. Lin, K. F. Lin, and Y. D. Chih. 2014. 19.4 embedded 1Mb ReRAM in 28nm CMOS with 0.27-to-1V read using swing-sample-and-couple sense amplifier and self-boost-write-termination scheme. In *IEEE*. 332–333.
[4] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store.. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 239–252.
[5] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory.. In *2020 USENIX Annual Technical*

Conference (USENIX ATC 20). 799–812.

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

[7] I. Corporation. 2009. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1.

[8] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *Proceedings Of The 2018 Usenix Annual Technical Conference*. 373–386.

[9] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu. 2015. Revisiting hash table design for phase change memory. *Acm Sigops Operating Systems Review* 49, 2, 18–26.

[10] B. Fan, D. G. Andersen, and M. Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. *Proc Usenix Nsdi*, 371–384.

[11] B C handramouli, G. Prasaad, D Kossmann, J. Levandoski, and M. Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *the 2018 International Conference*.

[12] Kaixin Huang, Yan Yan, and Linpeng Huang. 2020. Revisiting Persistent Hash Table Design for Commercial Non-Volatile Memory. *Proceedings of the 2020 Design, Automation and Test in Europe Conference and Exhibition, DATE 2020*, 708–713. https://doi.org/10.23919/DATE48585.2020.9116223

[13] Intel. 2019. Intel® Optane™ DC persistent memory. (2019). https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html

[14] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714*.

[15] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 191–205.

[16] Kibeom Kim, Yongjo Jeong, Youngjoo Lee, and Sunggu Lee. 2019. Analysis of counting bloom filters used for count thresholding. *Electronics (Switzerland)* 8, 7. https://doi.org/10.3390/electronics8070779

[17] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, Vol. 37. 2–13.

[18] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. In *Proceedings of the VLDB Endowment*, Vol. 13. 1147–1161.

[19] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Transactions on Storage* 13, 1, 1–28.

[20] Moohyeon Nam, Hokeun Cha, Young-Ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *FAST'19 Proceedings of the 17th USENIX Conference on File and Storage Technologies*. 31–44.

[21] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *the 2016 International Conference*.

[22] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage management in the NVRAM era. In *Proceedings of the VLDB Endowment*, Vol. 7. 121–132.

[23] Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Yi Chou Chen, Robert M. Shelby, Martin Salinga, Daniel Krebs, Shih Hung Chen, Hsiang Lan Lung, and Chung H. Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4-5, 465–479. https://doi.org/10.1147/rd.524.0465

[24] A. V. Renen, V. Leis, A. H. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. 2018. Managing Non-Volatile Memory in Database Systems.

[25] A. V. Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. 2019. Persistent Memory I/O Primitives.

[26] Anne Siemon, Thomas Breuer, Nabeel Aslam, Sebastian Ferch, Wonjoo Kim, Jan Van Den Hurk, Vikas Rana, Susanne Hoffmann-Eifert, Rainer Waser, Stephan Menzel, and Eike Linn. 2015. Realization of Boolean Logic Functionality Using Redox-Based Memristive Devices. *Advanced Functional Materials* 25, 40, 6414–6423. https://doi.org/10.1002/adfm.201500865

[27] Kosuke Suzuki and Steven Swanson. 2015. A Survey of Trends in Non-Volatile Memory Technologies: 2000-2014. In *2015 IEEE International Memory Workshop (IMW)*. 1–4.

[28] T. Wang and R. Johnson. 2014. Scalable logging through emerging non-volatile memory. *Proceedings of the Vldb Endowment* 7, 10, 865–876.

[29] S. Xu, S. Lee, Sang Woo Jun, M. Liu, J. Hicks, and Arvind. 2016. Bluecache: a scalable distributed flash-based key-value store. *Proceedings of the VLDB Endowment* 10, 4, 301–312.

[30] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. 2019. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. *arXiv*.

[31] J. Yang, B. Li, and D. J. Lilja. 2020. Exploring Performance Characteristics of the Optane 3D Xpoint Storage Technology. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*.

[32] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. 167–181.

[33] Donghui Zhang and Per Åke Larson. 2012. LHlf: lock-free linear hashing (poster paper). In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, Vol. 47. 307–308.

[34] Xiaomin Zou, Fang Wang, Dan Feng, Janxi Chen, Chaojie Liu, and Fan Li. 2020. HMEH : write-optimal extendible hashing for hybrid DRAM-NVM memory. *Mass Storage Systems and Technologies*.

[35] Pengfei Zuo and Yu Hua. 2017. A Write-friendly Hashing Scheme for Non-volatile Memory Systems. In *the 33rd International Conference on Massive Storage Systems and Technology (MSST), 2017*.

[36] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*. 461–476.