TurboHash: A Hash Table for Key-value Store on Persistent Memory

Xingsheng Zhao
University of Texas at Arlington
Arlington, Texas, USA
xingsheng.zhao@mavs.uta.edu

Chen Zhong
University of Texas at Arlington
Arlington, Texas, USA
chen.zhong@mavs.uta.edu

Song Jiang
University of Texas at Arlington
Arlington, Texas, USA
song.jiang@uta.edu

ABSTRACT

Major efforts on the design of persistent hash table on a non-volatile byte-addressable memory focus on efficient support of crash consistency with fence/flush primitives as well on non-disruptive table rehashing operations. When a data entry in a hash bucket cannot be updated with one atomic write, out-of-place update, instead of in-place update, is required to avoid data corruption after a failure. This often causes extra fences/flushes. Meanwhile, when open addressing techniques, such as linear probing, are adopted for high load factor, the scope of search for a key can be large. Excessive use of fence/flush and extended key search paths are two major sources of performance degradation with hash tables in persistent memory.

To address the issues, we design a persistent hash table, named *TurboHash*, for building high-performance key-value store. TurboHash has a number of much desired features all in one design. (1) It supports out-of-place update with a cost equivalent to that of an in-place write to provide lock-free reads. (2) Long-distance linear probing is minimized (only when necessary). (3) It conducts only shard resizing for expansion and avoids expensive directory-level rehashing; And (4) it exploits hardware features for high I/O and computation efficiency, including Intel's Optane DC's performance characteristics and Intel AVX instructions. We have implemented TurboHash on the Optane persistent memory and conducted extensive evaluations. Experiment results show that TurboHash improves state-of-the-arts by 2-8 times in terms of throughput and latency.

CCS CONCEPTS

• **Hardware** → **Non-volatile memory**; • **Information systems** → **Point lookups**; • **Theory of computation** → **Concurrent algorithms**.

KEYWORDS

hash table, non-volatile memory, lock-free read

ACM Reference Format:
Xingsheng Zhao, Chen Zhong, and Song Jiang. 2023. TurboHash: A Hash Table for Key-value Store on Persistent Memory. In *The 16th ACM International Systems and Storage Conference (SYSTOR '23), June 5–7, 2023, Haifa, Israel.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579370.3594766

1 INTRODUCTION

Hash table is a fundamental data structure for efficient organization of key-value (KV) data in the memory. It allows data to be quickly located without intermediate index search. This is especially important for organizing a very large number of small KV items, where often it may take only one cache-line memory access to retrieve a data item. Had many non-sequential memory accesses been required on an index structure, such as B+ tree or skip list, the actual cost of reading a small piece of data would be amplified by multiple times [28, 37]. Accordingly, hash tables have been employed to manage key-value cache in the DRAM, such as MemCached [12, 30] and Redis. With emergence of byte-addressable non-volatile memory, efforts have been made to design persistent hash tables on the memory [6, 16, 23–25, 27, 36, 44–47], as well as using persistent memory to build KV stores [10, 19, 20, 39, 41, 43].

One of the major issues and challenges on designing a high-performance persistent hash table is on its efficient support of crash consistency and atomic update. With crash consistency, a data structure can stay in a consistent state, or be restored to a consistent state after an unexpected crash. To have the consistency, one has to enforce a particular order on a set of actions. A simple example is that a hash table's bucket has to be allocated and initialized before its address can be assigned to a pointer in the table's directory. To enforce the order, fence/flush primitives (first fence and then flush), which are expensive, have to be used between the operations. More extensive use of fence/flush is required for updating a directory during a rehashing operation.

For data integrity, updating of a piece of data, such as a key or a value, must be atomic. After a recovery from an unexpected crash, either a version of the data before the update or the one after the update must be recovered. This means that the data cannot be modified in place if it is larger than an atomic write unit (8 bytes). Otherwise, it may destroy the old version without making the new version established at the time of a crash. Therefore, one has to use out-of-place write. However, this raises two potential performance issues for a modify/delete request as it needs to write at different places (creation of a new version and invalidation of the old one). First, currently available persistent memory often exhibits block-like access performance behaviors. For example, Intel Optane DC [18] has a 256-byte internal media access unit (block) [40]. NVDIMM (with flash as its storage media) [1–3] has a page-size (0.5KB or larger) media access unit. Two writes at two different blocks are significantly more expensive than one block write. Second, ideally the new version can be made visible and the old version is invalidated with one 8-byte atomic write. Otherwise, a write order has to be established between the two writes using a flush/fence.

Another major issue in the design of a hash table is how to resolve collision with both time and space efficiency. Upon occurrence of a hash collision in a full bucket, there are three approaches to resolve it. Approach A is to double the size of the hash table via rehashing to make each bucket be only about half occupied. Approach B is to apply Approach A only in a segment of buckets where the collision occurs to avoid global key reshuffling, such as extendible hashing [11]. And Approach C is to look for an idle slot in alternative buckets where the colliding key can be placed, such as cuckoo hashing [32]. These approaches (from A to C) become increasingly less disruptive to existing hash structures and more time-efficient. More extensive structural changes will lead to more space allocations, pointer assignments, and key relocation. Using a less disruptive approach can greatly help reduce impact of collision on the performance, especially on persistent memory where additional costs have to be paid for retaining crash consistency.

Approach C is usually known as open addressing. Example techniques include linear probing, quadratic probing, cuckoo hashing, Horton Tables [5], and Hopscotch hashing [15]. However, not all of the schemes play well with the persistent memory. For example, cuckoo hashing and its variant Horton Tables need to constantly relocate keys along a path to reach an idle slot. It has to frequently use expensive fence/flush. Schemes such as quadratic probing and cuckoo hashing often write new keys to non-consecutive buckets. For a read request to locate the key on its probing path consisting of non-consecutive buckets, its access speed is much lower than linear probing where the probing path represents a contiguous memory space. For example, Optane DC memory is capable of sequential prefetching with a sequential access speed 2-3X higher than that of random access [38, 40].

While a linear probing can be efficient, there are some critical issues to be addressed. First, the adjacent buckets on a probing path must be physically contiguous for search efficiency; Second, the probing scope must be sufficiently large for a high load factor and less frequent rehashing. Load factor is the size ratio of the space used for holding actual data and allocated space. It quantifies the space efficiency of a hash table. Without a carefully designed key placement policy, a search, especially a negative search (whose search key is not in the hash table), may have to cover most or all buckets in a probing scope, compromising read performance.

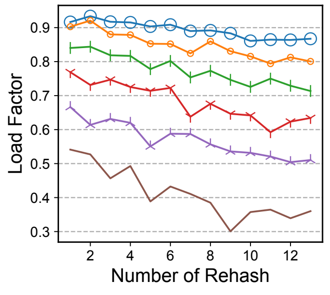
Our Solution In this paper, we propose a persistent hash table design, named TurboHash, for a high-performance key-value store on the persistent byte-addressable memory that exhibits the block-access performance characteristic by addressing all of the aforementioned issues. We use Intel Optane DC as the representative persistent memory (the PMEM hereafter) in the design. TurboHash first hashes keys into multiple shards. Each shard is a small hash table, which is set at a limited capacity (e.g., 1MB for 53,000 16-byte KV items) to cap the worst-case time (the tail latency) of requests serviced within a shard. In the meantime, we lavishly pre-allocate shards so that a TurboHash store's capacity limit can be way higher than the size of the physical PMEM a server can actually have. As each shard consumes only 8-byte metadata, this over-provisioning is well affordable. As an example, for a TurboHash with one million shards to provide an 8TB capacity limit (assuming a 8MB shard capacity), the space cost of representing shards is only 8MB. In return, this shard over-provisioning strategy avoids expensive resizing (or rehashing) over the entire hash table that may seriously compromise tail latency. There are only localized and small-scale resizing within individual shards to minimize tail latency.

TurboHash achieves much desired features, such as lock-free reads and short probing paths, which are often objectives of other hash-table optimization efforts. A unique contribution of TurboHash is its novel design that enables these features by accommodating the PMEM's block-like performance characteristics [43]. It has a 256-byte block access unit. Any small random access, such as updating of a few bytes of metadata, in the table may result in over 10X read/write amplification. Prior optimization techniques that assume an in-DRAM hash table may become ineffective for the PMEM. In one example, for lock-free reads, existing works, such as CLEVEL [6], carry out updating of a KV item usually by creating a new version and atomically switching the pointer pointing to the item from its old version to the new version. As the pointer and new version of the (small) item are often in two different 256B blocks, there would be two block writes, which is not efficient. In another example, to avoid holes due to deletions in a linear probing path, it has been suggested to move the item at the path tail to fill a hole [22]. However, this may involve two writes at different PMEM blocks: one is to invalidate the tail item, and the other is to write it into the hole.

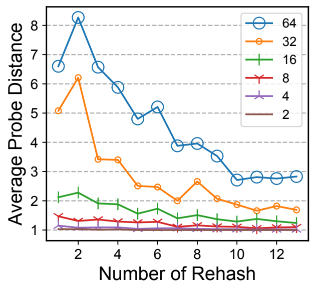
To this end, TurboHash makes a number of innovative design choices to confine multiple writes/reads of data and metadata within one 256B PMEM block to unlock its full performance potential. In particular, TurboHash's buckets are of 256 bytes each and are physically contiguous. It proposes to use near-place update (in the same bucket where the old version stays), instead of out-of-place, and one atomic write to invalidate/validate old/new versions, respectively, of a KV item within the 256B bucket to enable efficient in-PMEM lock-free reads.

In summary, we make a number of contributions:

- Recognizing that a large probing scope is important for a high load factor and infrequent rehashings, we experimentally reveal that actual probing distances can be much shorter than the probing scope. We then propose a strategy that enables only necessary probing distances, especially for negative search.
- We tailor TurboHash's design for a large-scale KV store on persistent memory with its sharded structure and physically contiguous bucket layout for minimal use of fence/flush and efficient memory access.
- This work represents a first effort of extensively exploiting PMEM's block-access-like performance characteristic in its design to customize hash-table optimization techniques.
- We evaluate TurboHash in comparison with recently proposed persistent hash table designs, such as CCEH, dash, and CLEVEL hashing. Experiment results show that TurboHash has 2× to 8× improvements in terms of throughput and latency.

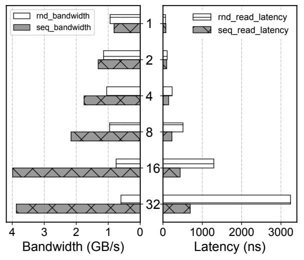


(a)

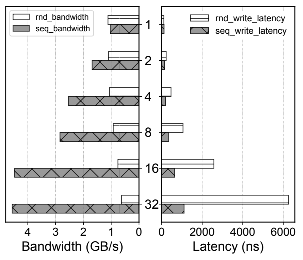


(b)

Figure 1: Load factors and average probing distances with different probing scope (in number of buckets) during insertion of 100 million KV items.



(a) PMEM Read



(b) PMEM Write

Figure 2: PMEM performance (random and sequential) measured by the PCM (Processor Counter Monitor) tool. Each write is followed with a *clwb* to flush the data, and each group of writes is followed with *mfence*.

2 MOTIVATIONS

In this section we present results of an experimental study on the characteristics of linear-probing-based hash tables on the PMEM to serve as rationale of TurboHash's design.

Probing Scope and Distance. In a hash table using open addressing, the number of alternative buckets where a new key under collision can be placed is highly correlated to the table's load factor. The more alternative buckets, the more leeway a key has for its placement at locations other than its home bucket (the bucket initially given by the hash function). A key's probing scope refers to the set of buckets this probe is likely to reach (starting from its home bucket). Increasing the scope is important for high space efficiency. We assume a new key is placed in the first bucket with idle slot(s) on its linear search path. A key's search path grows within its probing scope when more keys are hashed to its home bucket. The path ends at the bucket where the newest key to the home bucket is inserted. In theory, a key probing can terminate at the last bucket on the path. The probing distance is the actual number of buckets a probe has to traverse on the path to either find the search key (positive search) or declare the key doesn't exist in the table (negative search). Therefore, the distance is capped by the path's length and can be shorter than the probing scope's size. Still, increasing the probing scope allows for longer probing distances, which may make a key search more expensive. We study the relationship between probing scope, load factor, and probing distance. To this end, we set up a linear-probing-based hash table of 1024 buckets. Each bucket has 16 16B-slots. The table is rehashed by doubling its size whenever a collision cannot be resolved within a probing scope. We keep inserting keys to an initially empty hash table with a given probing scope size. Figure 1a shows load factors right before every rehashing. As shown, the scope size has a significant impact on the load factor. Small scopes, such as 2 or 4 buckets, can lead to too-low load factors. A search path's length represents the necessary probing distance of a negative search for a key, as the key doesn't exist beyond the path's end. Figure 1b shows average search distances corresponding to load factors in Figure 1a at different probing scope sizes. As shown, the necessary search distance can be much shorter than the corresponding probing scope. For example, at the 6th rehashing the load factor is 0.8 with the 16-bucket scope. But the average probing distance is only 1.7 buckets. This implies that as long as the search path is recognized, the actual probing distance can be short for high access performance, and a large probing scope can be used for high load factor.

Sequential and Random Accesses. To illustrate the performance gap between sequential and random accesses on the persistent memory, we experimentally simulate access patterns where each probing is either on random memory locations or sequential locations. Each probing consists of a group of 8-byte accesses. These accesses are on different 64-byte memory spaces (simulating buckets), which are either randomly or sequentially placed. A probing always starts at a random location. The access group size is set to 1, 2, 4, 8, 16, or 32 (simulating probing distance). Figures 2a and 2b show the read/write throughput and average latency for each access group. As shown, for a reasonably large group, such as those with 4 or more accesses, sequential performance is significantly higher than that of its random counterpart in terms of either throughput or latency. The gap becomes even larger with a larger group. With a group of 16 accesses, the performance gaps are about 4-7X. The reason is that the PMEM is accessed in the 256B unit. With a random access of 8 bytes, one 256B block is actually accessed at the persistent memory's media [40], causing a significant read or write amplification. A sequential access can also benefit from the prefetching mechanism.

3 THE DESIGN OF TURBOHASH

There are a number of challenges we must address to achieve TurboHash's design objectives, including high load factor, sequential and short search path, support of out-of-place updates, crash consistency, lock-free reads, and minimal use of fence/flush. The challenges that have been addressed in the design include: (1) how to limit a search within necessary distance, rather than the entire probing scope, especially for non-existing keys? (2) how to switch from the old version to the new version of a KV item in one atomic primitive? and (3) how to leverage hardware features such as the PMEM's internal block-access characteristic and CPU's SIMD execution support for higher I/O and computation efficiency?

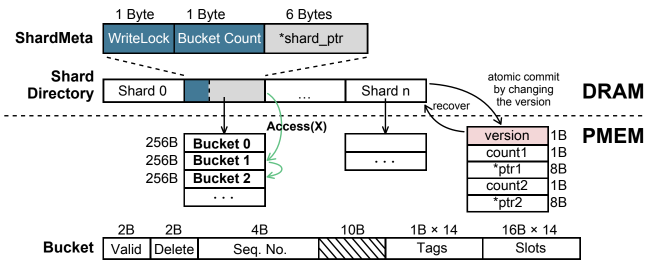


Figure 3: TurboHash's Architecture. When key or value is larger than 8 bytes, the slot in a bucket stores an 8-byte hashed key and an 8-byte pointer to the KV item, which is stored at a separate space.

3.1 The Architecture

As we have stated, the entire key space is partitioned into many shards. Keys in each shard are organized in a hash table. Each of the hash tables can have thousands of 256B buckets. It supports efficient resizing, high load factor, and high-performance access. By using a highly randomized hash function, such as MurmurHash [4] or MD5, keys are uniformly hashed into the shards.

As shown in Figure 3, each shard has a descriptor. All descriptors form a shard directory. The directory stays in the PMEM for its persistency, and is mirrored in the DRAM for access efficiency. Each shard descriptor records bucket count in the shard and a pointer to the shard. There are differences between shard descriptors in the PMEM and in the DRAM. In the PMEM, it includes two sets of bucket count and pointer to allow out-of-place updating of the count/pointer after a shard resizing. Additionally, it has a 1-byte version number indicating which set is currently in use. After writing a new set of count/pointer, an atomic update of the version number makes it effective. Then, count/pointer in the corresponding descriptor in the DRAM are updated. And the old shard will be recycled using the epoch-based reclamation [13].

There is a write lock (WriteLock) in the in-DRAM descriptor that establishes mutual exclusion among service of write requests (i.e., insert, update, or delete) in a shard. Because there can be many (a few thousands or more) shards in the hash table, the impact of the lock on concurrency is limited. In particular, this lock is only applied on writes, which are more expensive as they likely involve writing KV items, updating metadata, and even shard rehashing. In contrast, TurboHash makes service of read requests fully lock-free.

3.2 Establishing the Search Path

A shard is allocated as a whole with all of its buckets in a contiguous space. Each bucket has a fixed number of slots. Each slot holds one KV item. For a linear probing scheme, in today's practice a new KV item can be placed into any empty slot within the probing scope. Slots can become available anywhere in the scope whenever their resident KV items are deleted. New KV items may be placed in any of these empty and other available slots in the scope. While such a placement without restriction is flexible and space efficient, it often makes the search distance much longer. In the search for a non-existing key, which is the operation carried out before every new key insertion, the distance will always be the scope size. A search has to proceed until the search key is found or it reaches the boundary of the probing scope. The buckets do not contain information to establish a search path which can be (much) shorter than the scope size so that only necessary buckets are searched. As indicated, if we can keep new KV items in buckets as close as possible to their home bucket and establish a search path covering the buckets, a search does not have to walk beyond the path. In this case, when the scope is set to a large size for high load factor, the cost of key search does not have to increase proportionally.

The solution appears at first sight to be a straightforward one, which is just to place a new KV item in the available slot on its linear search path and remember the path's last bucket. This is a valid idea. However, the difficult question is how to remember a path's last bucket. An intuitive approach would be to explicitly record this path-end information and update it whenever the path grows. But this approach leads to high time overhead and likely high space overhead. The possible places where the path-end can be recorded may be the home bucket, the currently end bucket on the path, or a separate data structure. In any of the places its updating overhead can be too high. When a new item is written into a new bucket and extends the path, the path-end must be accordingly updated (an ancillary write). These two writes are likely in two different PMEM's blocks, which are actually two block writes. Furthermore, the ancillary write must be completed before the KV write to guarantee following reads will reach the new item and the new item can be reached after a system crash. To this end, a fence/flush is required. If the path-end is recorded in the end bucket, two ancillary writes are required when the path grows (one in the new end bucket for indicating the new path-end and one in the old end bucket for invalidating it). Furthermore, the space overhead can be too high, as a bucket can simultaneously be the end bucket of multiple paths (up to the probing scope size). It would be too expensive for each bucket to pre-allocate such a large space for these possible paths. To address the issue, TurboHash doesn't record and update this path-end information at all. Its approach is motivated by three observations. (1) In most of the time a hash table has abundant empty bucket slots until a rehashing is to be triggered soon. (2) Once a rehashing is carried out, many empty bucket slots are spread out in the table. And (3) even when a rehashing is near, the load factor is still less than 80-90% (see Figure 1a). Therefore, TurboHash uses a non-fully occupied bucket (with empty slots) to indicate a path end. To this end, it gives a slot whose data has been deleted a flag indicating that this slot is available for receiving a new KV item but isn't considered an empty slot. A path grows only when there are not empty slots in its current end bucket (and certainly not in any other buckets on the path). In this way, if a bucket with empty slot(s) are encountered during a search on a path, it is guaranteed that the search key will not be found beyond this bucket and continuous search is not necessary. Admittedly, this bucket may not be an accurate end bucket of the path (as it may not contain key(s) hashed to the path's home bucket). However, this is good enough to significantly reduce number of buckets involved in a search, as will be demonstrated in Section 4.

Another design issue is on the update operation. As we have indicated, for data crash consistency, an out-of-place write, instead of in-place overwrite, is required for an update operation. The challenges are similar to that with updating of the path-end information.

First, there are two writes: one for writing the new KV item and an ancillary one for invalidating its old version. These two writes would be random accesses if without a careful arrangement. Second, the operation that makes the new version visible and the old version invisible has to be an atomic one for correctness if a lock-free read is allowed for high throughput. TurboHash's solution is to introduce near-place update, which limits the out-of-place write within the same bucket where the old-version KV item resides. As a bucket is of 256B and the metadata on (in)validating slots in a bucket are within a 8B atomic write unit, all the writes are in one the access block with high efficiency.

3.3 A Bucket's Data Structure

As we have mentioned, for access efficiency each bucket is set at 256B long, the access unit of the the PMEM. As shown in Figure 3, within a bucket there are 14 slots, each for storing a KV item with a 8B key and a 8B value. The 8B value can be a pointer to another space where the real value is stored should the value is larger than 8B. Besides the data, there are three types of metadata in a bucket. One is about slot status, including the valid bitmap and the delete bitmap. A bit in the 2-byte valid bitmap indicates whether the key in the corresponding slot is valid. A bit in the 2-byte delete bitmap indicates whether the KV item in the corresponding slot has been deleted. A slot's valid bit becomes 1 when a new KV item is written into it. When this item is deleted, its delete bit becomes 1. However, its valid bit remains as 1 as the key in the slot is still meaningful. A slot holding a deleted key isn't considered as an empty slot for the purpose of detecting a path end. Only a slot whose valid bit is 0 is defined as an *empty* one. However, the slot holding the deleted item is available to receive a new KV item by overwriting the deleted one. An insert operation always writes a new KV item in the first slot with a deleted item (if available) on its search path. After the overwriting, this slot's valid and delete bits become 1 and 0, respectively.

The second type of medadata is for concurrency control, including a 4-byte sequence number. This number is incremented by one whenever a write operation (delete, insert, or update) happens in the bucket. It facilitates lock-free reads.

The third type of metadata is for improving performance, including an array of tags. A tag is a 1-byte summary of a key by hashing the key in the corresponding slot. By grouping 14 summary keys in an array, TurboHash can use an SIMD instruction ("*_mm_cmpeq_epi8_mask*") for a quick preliminary search of all keys in a bucket.

As mentioned, TurboHash introduces the near-place update for high efficiency. To this end, it reserves an empty slot in each bucket for out-of-place updating of a KV item in the same bucket. That is, if a slot is the only (last) empty slot in a bucket, it doesn't accept a new KV item. This bucket is considered full for the purpose of indicating a path end.

3.4 Insert, Update, Delete, and Read

We describe the search operation before detailing how the four types of requests are served.

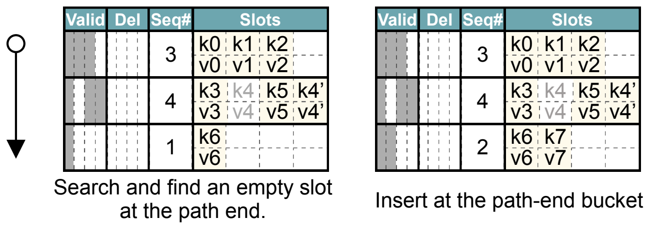


Figure 4: Illustration of inserting a new KV item ("K7V7"). There are four slots in a bucket. Shaded boxes in the bitmap indicate valid or deleted slots. "Seq#" is incremented after an insert to support lock-free reads.

The Search Operation. Search for a given key is the most frequently operation in a hash table. It not only is used to service a read request but also has to be employed before every insert/update/delete operation is performed at a bucket. The difference between a read request and a write request is that the latter one needs to hold the write lock during the search. Like that in any linear-probing hash table, a search begins at the home bucket determined by the search key and the hash function, and continues on the sequential search path. At each bucket on the path, it needs to compare the search key with each of keys in the bucket. To speed up this process, TurboHash compares a 1-byte hashed value of the search key with each of the 14 tags in the bucket in parallel by using an Intel AVX SIMD instruction. Only the slots whose tags are matched and their valid bits are set and delete bits are not set will have their keys compared with the search key. If there is a match with the search key, the search key is found. If there isn't a match of the search key, the search will continue to the next bucket. It will terminate at a bucket where a matched key is found or at the path-end bucket, which is defined as the one with more than one slot whose valid bit is 0, or the one that has at least two empty slots (one of them is reserved for near-place update). The search operation is part of an insert/update/delete/read request service (as shown in Algorithm 1).

Insert. The thread servicing an insert request first acquires the write lock and proceeds with the aforementioned search operation for an empty slot or a valid slot that contains the search key within the search scope. Note that the empty slot does not include the one containing a deleted key. The search remembers position of the first deleted key it encounters. If the search reaches the path end (a bucket with more than one empty slot with one of the them reserved for near-place updating), this is a negative search. If a deleted key has been recorded on the path, the new key is inserted in its slot. Otherwise, it is inserted in one of the empty slots in the path-end bucket, as shown in Figure 4. To facilitate lock-free read, the insert thread takes two steps for the insert operation in a bucket. It first writes the key and value into the 16B-slot's data segment as well as the corresponding tag. It uses fence/flush to secure the data on the PMEM before moving to the second step, in which it makes the new data visible to reads. Specifically, it uses one 8-byte atomic write to set valid and delete bits to '1' and '0', respectively, and increments the sequence number by 1. Again it uses fence/flush to conclude the insert and then releases the write lock. If the search reaches end of the search scope and still cannot find an empty space

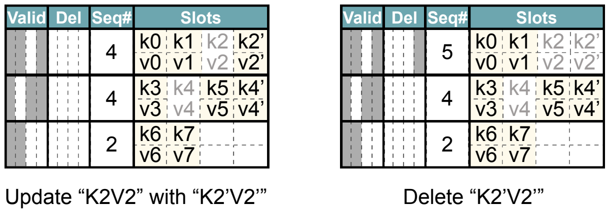


Figure 5: Illustration of the near-place update (left) and delete (right). KV items in the grayed-out slots have been invalidated.

Algorithm 1: Lock Free Search

```
1  Function Search(key, callback):
2      [slot, isfind] = FindSlot(key)
3      if isfind then
4          callback (slot)
5          return TRUE
6      return FALSE
7  Function FindSlot(key):
8      key_hash = Hash(key)
9      shard = LocateShard(key)
10     bucket = LocateBucket(shard, key_hash)
11     hash_tag = ExtractTag(key_hash)
12     probe_distance = 0
13     while probe_distance < MAX_DISTANCE do
14         seq_no = bucket.meta.seq_no
15         match_pos = bucket.SIMDmatch(hash_tag)
16         foreach pos ∈ match_pos do
17             slot = buckets.slots[pos]
18             if key == slot.key then
19                 if bucket.NeedRetry(seq_no) then
20                     bucket = LocateBucket(bucket.id)
21                     go to 14 // Retry in current bucket
22                 return {slot, TRUE}
23         if bucket.ReachSearchEnd() then
24             return {"", FALSE}
25         bucket = bucket.Next()
26         probe_distance++
27     return {"", FALSE} // Reach maximum probe
           distance.
```

or deleted space for an insertion, it performs shard rehashing and then inserts the key in the enlarged shard before releasing the write lock. If the search arrives at a valid slot holding the search key not yet deleted, this insert is an update . The pseudocode for insert and update is in Algorithm 2.

Update. As mentioned, TurboHash's update operation is a near-place update. Like the insert operation, an update within a bucket also takes two steps. Holding the WriteLock, the thread uses an empty slot in the bucket as the reserved slot to write the new KV item and updates the corresponding tag followed with a fence/flush. It then uses one atomic write to the slot's valid bitmap to turn the old version invalid and this new version visible, and increments the sequence number by one, as shown in the left graph of Figure 5. With the atomicity, the valid bits for the old and new version slots

Algorithm 2: Insert and Update

```
1  Function Insert(key, value):
2      shard = LocateShard (key)
3      WriteLockGuard guard(shard)
4      is_find, bucket_id, slot_id, old_slot_id =
         FindSlotForInsert(key)
5      if is_find == TRUE then
6          bucket = LocateBucket(bucket_id)
7          bucket.Insert(key, value, slot_id, old_slot_id)
8          return TRUE
9      else
10         shard.Rehash()
11         go to 2
12 Function Bucket::Insert(key, val, slot_id, old_slot_id):
13     slot = slots[slot_i]; tags[slot_i] = key.tag
14     slot.key = key; slot.val = val
15     CLWB, SFENCE
16     new_meta = this->meta;
17     new_meta.valid = new_meta.valid | ( 1 << slot_i )
18     if old_slot_i ≠ -1 then
19         new_meta.valid ⊕ = ( 1 << old_slot_i )
           // Epoch based reclamation.
20         epoch.markForDeletion(slots[old_slot_i])
21     new_meta.delete & = ~( 1 << slot_i )
22     new_meta.seq_no++
23     this->meta = new_meta
24     CLWB, SFENCE
```

can only be '10' or '01' at any moment during the operation. Without holding any lock, a read thread can see one and only one version at any moment. After this, the WriteLock is released.

Delete. After a delete service thread identifies the slot storing the same valid key that has not yet been deleted using the search operation with the WriteLock, it uses one atomic write to set the slot's delete bit and increments the sequence number by one, as illustrated in the right graph of Figure 5. It then releases the lock. Note that it does not reset the slot's valid bit, and thus does not break any search paths (as shown in Algorithm 3).

Read. A read thread does not need any lock. It uses the search operation to look for the key. If it is not found within the probing scope, the read completes, declaring non-existence of the key. Otherwise, if a valid and non-deleted key is found in a slot, we cannot simply return the value in the slot. Because without holding a lock, the read value is likely modified right before the read and is thus a wrong one.

Within the slot there are two phases of read. One is carried out by the search operation, including reading the tags, valid/delete bits, and the keys for comparison. If the first phase finds the read key, the second phase is to read the corresponding value. These two phases of read are not atomic and cannot prevent other write threads from interfering. To eliminate a potential hazard, the read thread reads the sequence number before and after reading the value. Fence is placed between the two reads to ensure the ordering. It then compares the two numbers. If they are equal, the correct value has been read. Otherwise, the value may have been modified and could be wrong. And the read operation is retried.

Algorithm 3: Delete

```
Function Delete(key):
    shard = LocateShard (key)
    WriteLockGuard guard(shard)
    bucket = LocateBucket(shard, key_hash)
    probe_distance = 0
    while probe_distance < MAX_DISTANCE do
        match_pos = bucket.SIMDmatch(hash_tag)
        foreach pos ∈ match_pos do
            slot = buckets.slots[pos]
            if key == slot.key then
                bucket.Delete(pos)
                /* Epoch based reclamation */
                epoch.markForDeletion(slot)
                return TRUE
        if bucket.ReachSearchEnd() then
            return FALSE
        bucket = bucket.Next()
        probe_distance++
    return FALSE // Reach the maximum probe distance.
Function Bucket::Delete(slot_id):
    new_meta = this->meta;
    new_meta.delete |= 1 << slot_id
    new_meta.seq_no++
    this->meta = new_meta
    CLWB
    SFENCE // Persist the 8-byte bucket meta.
```

To explain why reading wrong values will not occur with the use of sequence number, we examine the timeline of the relevant read/write events. For the read thread there are four read events, which are R1 (read the tags), R2 (an atomic read of valid/delete bits and sequence number) for all slots with valid/non-deleted bits and matched tags, R3 (read the relevant keys and then value of the matched key), and finally R4 (read the sequence number again). We consider a write request (delete, insert, or update) is completed at its last atomic write to update bitmaps and increment the sequence number. If the two sequence numbers are equal, there must not be any write requests completed between R2 and R4 because any such a completion will cause the sequence number to be incremented. Furthermore, because service of write requests is serialized, there is at most one write request between R2 and R4. Otherwise, there must be a completion of a write request. Fortunately, if there is a write request between R2 and R4, it is guaranteed that it will not modify the bitmaps/sequence-number, or the read key and its value in the bucket. First, if the write request is an insert, it must have matched with a key different from the read key. Otherwise, the valid/non-deleted status of the read key would keep it from inserting in the bucket. Second, if it is a delete or an update, both do not change the read key and value. Therefore, if the read thread confirms that the sequence number does not change, the read value must be correct. In addition, we show that a read thread can always find its target key and value if they are in the table. Obviously this is not an issue if the target key is not involved in any write requests during the read service. The target key is not available for the read if it is read after a delete or before an insert of the key. This is not an issue either. If the key is the target of an update during the read (between the R2 and R4 read events), the read thread can always

Algorithm 4: Ancillary Functions

```
Function FindSlotForInsert(key):
    key_hash = Hash(key)
    shard = LocateShard(key)
    bucket = LocateBucket(shard, key_hash)
    bucket_id = -1
    slot_id = -1
    hash_tag = ExtractTag(key_hash)
    probe_distance = 0
    while probe_distance < MAX_DISTANCE do
        match_pos = bucket.SIMDmatch(hash_tag)
        foreach pos ∈ match_pos do
            slot_key = bucket.slots[pos].key
            if key == slot_key then
                slot_id = bucket.PickEmptySlot()
                return {TRUE, bucket.id, slot_id, pos}
        if slot_id == -1 And bucket.meta.delete ≠ 0 then
            bucket_id = bucket.id
            slot_id = bucket.PickDeleteSlot()
        if bucket.ReachSearchEnd() then
            if slot_id == -1 then
                bucket_id = bucket.id
                slot_id = bucket.PickEmptySlot()
            return {TRUE, bucket_id, slot_id, -1}
        bucket = bucket.Next()
        probe_distance++
    if slot_id ≠ -1 then
        return {TRUE, bucket_id, slot_id, -1}
    return {False, -1, -1, -1}
Function Bucket::SIMDmatch(hash_tag):
    hash_vec = _mm_set1_epi8 (hash_tag)
    res = _mm_cmpeq_epi8_mask (hash_vec, tags);
    return res & valid & (~ delete)
Function Bucket::NeedRetry(old_seq_no):
    if old_seq_no ≠ meta.seq_no then
        return TRUE
    return FALSE
Function Bucket::PickEmptySlot():
    return __builtin_ctz(~ valid )
Function Bucket::PickDeleteSlot():
    return __builtin_ctz(delete)
Function Bucket::CanInsert():
    return delete != 0 or ReachSearchEnd()
Function Bucket::ReachSearchEnd():
    return __builtin_popcount(valid) < 13
```

read a value (assuming the key is not yet deleted). If the update completes before R4, a retry will be performed to read the new value. Otherwise, it will read the old value.

3.5 Shard Resizing

Shard resizing is performed by an insert thread when it cannot find an available slot within the probing scope. Then the shard, whose size is up to a few MB, is sequentially read to the DRAM, where an enlarged shard is built. The new shard is then sequentially written to the PMEM. During the period of time, read requests can still be served on the old version of the shard. When the new shard has been persistently written, the pointer to the shard in the shard's descriptor is updated (first in the PMEM descriptor then atomically updated in the DRAM). After this, all requests will be served on the new version. When the read threads on the old shard complete their service, the space for the old version can be reclaimed when no other threads access it.

3.6 Variable-Size Items and Consistency

TurboHash stores an oversized KV item into a dynamically allocated space. It then writes its hashed-key and a pointer to the space into an 8-byte/8-byte bucket slot in the hash table. There are three potential consistency-related issues. (1) Updating indirectly a remote KV item is not atomic and may compromise the data. For this, TurboHash uses out-of-place updating, which writes the new version into a separate space and then frees the old space. (Note that out-of-place updating is also used for regular 8B-key/8B-value KV items for consistency). (2) When a dynamically allocated space is being reclaimed, on-going lock-free reads may still be on the space. Therefore, TurboHash uses the epoch-based reclamation technique [13], where pointers to to-be-freed spaces are first sent to a garbage pool, and a space is reclaimed only when no threads are accessing it. (3) Dynamically allocated space and its pointer recorded in a bucket slot may not be crash-consistent (e.g., a crash may occur after the space is allocated and before the pointer points to the space). TurboHash uses a leak-free PMEM allocator to avoid memory leaks. Intel's Persistent-Memory-Development-Kit(PMDK) supports atomic allocation and free operations(e.g., `pmemobj_alloc()` and `pmemobj_free()`). They take address of the pointer to the allocated space, atomically allocate/free the space and update the pointer in a thread-safe and fail-safe manner. The transactional operations are guaranteed to be entirely completed or discarded on recovery.

3.7 Failure Recovery

TurboHash does not change its directory structure. It introduces a new shard with an 8-byte atomic update by modifying its version number in the PMEM and the 8-byte ShardMeta in the DRAM (shown in Figure 3). All the operations are protected with the Intel PMDK's transactions support, which protects the data structure from corruption due to a power failure. Therefore, a system crash will not lead to consistency issue for the directory. As all writes are only available after committing the valid/delete bitmaps, partially updated KV items are not visible to users. The only operation for a failure recovery is to read the directory in the PMEM and rebuild the in-DRAM directory.

4 EVALUATION

In this section, we evaluate TurboHash by comparing it with several state-of-the-art hash tables for persistent memory, including CCEH [27], Dash [25], P-CLHT [24], and clevel hashing (CLEVEL) [6]. We implement TurboHash using Intel's Persistent Memory Development Kit (PMDK) [17]. CCEH is a dynamic hash table for persistent memory. It supports resizing through segment splitting and directory resizing. Dash is also a dynamic hash table. It is similar to CCEH, but with several optimizations, including fingerprinting [31], and version-based search. P-CLHT is a linked-list based persistent hash table derived from CLHT [8]. Each bucket of P-CLHT can store three KV items. CLEVEL is an upgraded version of level hashing [45] by enabling asynchronous resizing.

Table 1: Comparison of Design Choices

Hash Table	Bucket Size	Probing Scope	Search Strategy
CCEH	64B	8 contingous buckets	search all
CLEVEL	64B	4-8 random buckets	search all
CLHT	64B	all buckets on linked-list	search all
Dash	256B	2 buckets + stash buckets	search all
TurboHash	256B	16 contiguous buckets	buckets on a search path (avg < 2)

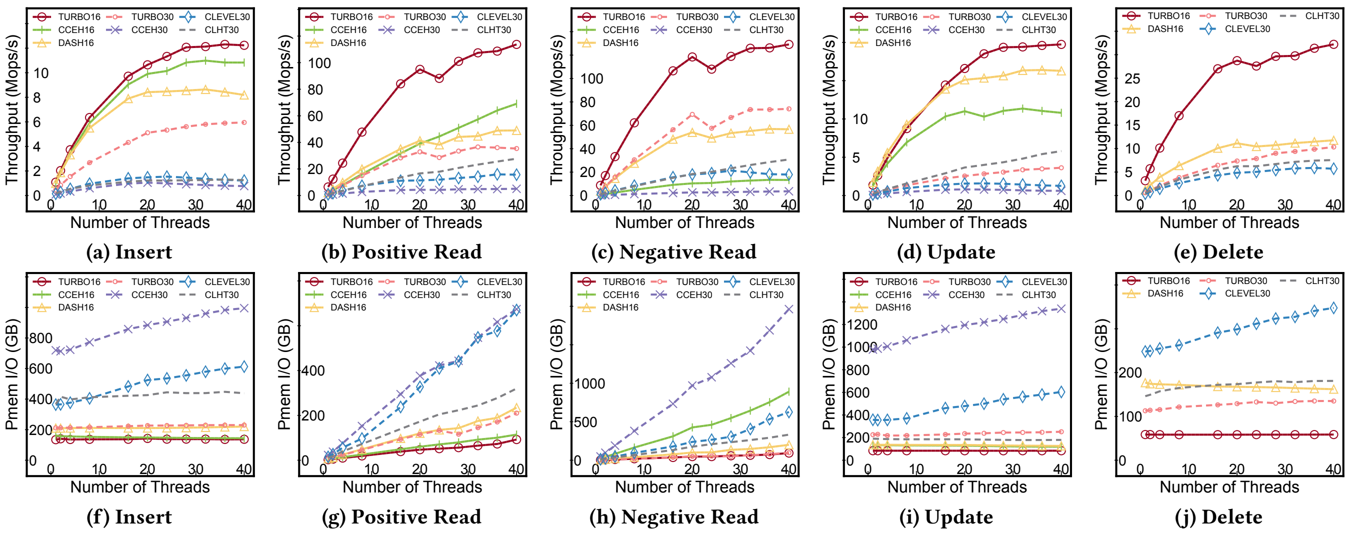
4.1 Experiment Setup

In our experiments, we use two different key-value sizes for comparison, 16 bytes (8B key and 8B value) and 30 bytes (15B key and 15B value). In the case of 30-byte KV items, real key and value are in a separately allocated space, and a 8B hashed key and a 8B pointer to the space are stored in the hash table. *CCEH16* and *DASH16* assume 16-byte KV item (8B key and 8B value). *CCEH30*, *CLEVEL30*, and *CLHT30* use 30-byte key-value items (15B key and 15B value). They are implemented using `libpmemobj` by authors of the clevel hashing paper (`git:#13ad3f2`) . The TurboHash with 16B and 30B KV items are named *TURBO16* and *TURBO30*, respectively. All the threads in an experiment are pinned to one socket using `numactl`. For a fair evaluation, other hash tables in comparison are pre-sharded(into segments/buckets). All of the hash tables are initialized with a capacity of 12 million KV items. In the experiments, TurboHash is initialized with 64K shards (each has 16 buckets to have a total of 12 million slots at the beginning), and uses a 16-bucket probing scope. CCEH16 uses a 8-bucket scope (we use the copy-on-write version in the evaluation as it provides better read performance). CCEH30 uses a 4-bucket scope, as suggested in their code. CLEVEL30's scope is between 4-8 buckets, depending on its level count. CLHT30 conducts probing within a hash bucket. Its buckets are organized on a linked list for collision resolution. Table 1 compares basic design choices of the hash tables.

The experiments are run on a server with an Intel Xeon Gold 6230 20-core processor, 64GB DRAM, and 6×128GB Intel Optane DC. .

4.2 Overall Performance

To evaluate the performance of the hash tables, we conduct extensive experiments, including insertion of new KV items (*Insert*), reading existent keys (*Positive Read*), reading non-existent keys (*Negative Read*), overwriting existing KV items (*Update*), and deleting all KV items (*Delete*). Experiment results are shown in Figure 6. In each experiment different number of threads (from 1 to 40 threads) are used. For *Insert*, *Update*, and *Delete*, each thread sends $120million/Number\_of\_threads$ requests. For *Positive Read* and *Negative Read*, each thread sends 10 million requests. Figure 6 reports throughput of the hash tables (number of requests serviced per second) and the corresponding raw PMEM I/O volume. This I/O volume represents all read/write data amount on the Optane PMEM's media, including amplified I/O due to its 256B access unit.



(a) Insert (b) Positive Read (c) Negative Read (d) Update (e) Delete

(f) Insert (g) Positive Read (h) Negative Read (i) Update (j) Delete

Figure 6: Throughput and PMEM I/O volume with different requests per threads.

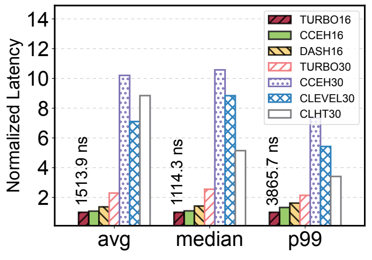
It is measured with `ipmwatch`, available in the Intel VTune Amplifie tool.

Insert. TurboHash conducts about four rehashings within a shard during the insertions. Other hash tables in comparison also conduct four rehashings. All the hash tables have a load factor of around 50%. For small key-value size (8B key and 8B value), CCEH16, DASH16, and TURBO16 store the data in the hash table slots. As shown in Figure 6a, with fewer than 20 threads TURBO16 and CCEH16 have similar *insert* performance. When the number of thread is more than 20, TURBO16's throughput is 10% higher than CCEH16. The performance gap between TURBO16 and CCEH16 is due to use of lock: TURBO16 uses a DRAM spinlock while CCEH16 uses an in-PMEM writer lock on its hash table segment. The lock on the PMEM causes more performance penalty because of the contention on the PMEM XPBuffer and contention in the iMC [40]. DASH16 tries to displace the KV items between the buckets to have a balanced insert, which causes more I/O during insertion. As shown in Figure 6f, DASH16 has 30% more I/O than TURBO16 and CCEH16. For 30-byte KV size, TURBO30's throughput is about 4× higher than the others. For CCEH30, we see a much larger amount of PMEM I/O (10× more than TURBO30's) (see Figure 6f). The main reason is that in order to support the atomic update operation for data size larger than 8 bytes, CCEH30 uses `libpmemobj`'s transaction feature for out-of-place writes. It uses undo logging for application object updates, and introduces large write amplification [42]. Because of the limited PMEM bandwidth, this much increased I/O leads to CCEH30's large write throughput degradation. While CLHT30 uses a linked list to organize its buckets, searching on the list causes random small access on the PMEM. As shown in Figure 2, random access to the PMEM can be 4X slower than sequential writes. CLEVEL has a bottom-to-top searching strategy in its multi-level structure. It has to search all of the possible buckets to ensure the key does not exist before any insertion. Searching the randomly located buckets introduces large read amplification, causing its PMEM I/O volume 5× more than TURBO30, as shown in Figure 6f.

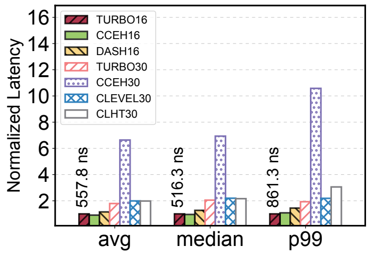
Positive Read. We see significant performance improvement for TURBO16 over CCEH16. The source code of CCEH16 implements double hashing for read, which aims to increase load factor at the cost of larger read amplification. As we can see in Figure 6g, CCEH16 has more I/O than TURBO16. TURBO16 also has better read performance than DASH16. This is due to DASH16's higher I/O volume because of the stash search. For 30-byte key-value items, TURBO30 has up to 3X throughput improvement over the others. The major reason is that TurboHash uses sequential read as much as possible to minimize read amplification and thus has the least I/O volume, as shown in Figure 6g.

Negative Read. TurboHash has the best negative-read performance among the schemes. A major reason is that it reads much less amount of data during the key search (see Figure 6h). TurboHash reads only KV items before the end of a search path. DASH16 has to search all the target buckets and the stash buckets. CCEH also needs to search its entire scope. CCEH30 reads more data than CCEH16 because CCEH30 doubles the bucket size to accommodate large key and value, causing more data to be read in a probing scope. CLEVEL30 and CLHT30's search is on a path consisting of non-contiguous memory locations, which compromises the performance. With reduced search paths and sequential PMEM access, the negative read performance of TurboHash is much higher than the others (Figure 6c).

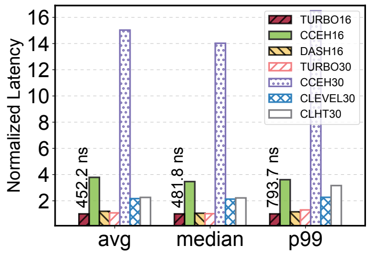
Update. In the *Update* experiment all existent KV items are updated. *Update* is a special case of *insert*. Compared to *Insert* throughput, TURBO16's *Update* throughput improves the most (almost 2 × as high as its *Insert* throughput). There are two reasons. One is that *Update*'s search path is shorter than *Insert*'s. The other is that *Update* does not cause shard rehashing. TURBO30's performance is lower than CLHT30, this is because CLHT30 uses 30-byte bucket size and does in-place update, while TURBO30 applies indirection for variable-size KV items and out-of-place update. Hence,



(a) Insert Latency.



(b) Positive-Read Latency



(c) Negative-Read Latency

Figure 7: Latency comparison between hash tables (120 million insertions and 160 million reads with 16 threads)

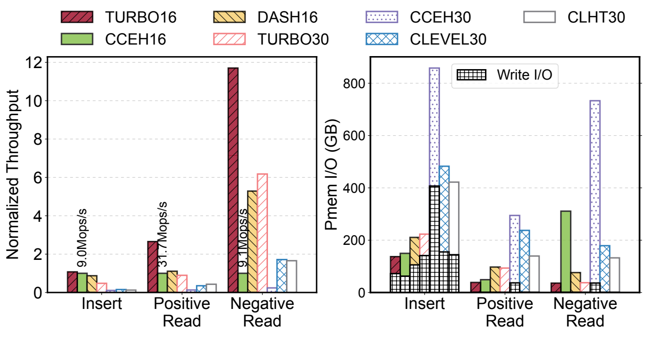


Figure 8: Throughput and PMEM I/O volume (120 million Inserts and 160 million Reads using 16 threads). Grey bars in the right graph indicate the portion of write volume.

TURBO30 introduces more I/O because the old items have to be recycled, as shown in Figure 6i, TURBO30 has 30% more I/O than CLHT30. Though CLHT30's performance is higher, it lacks the ability to support variable-size KV items.

Delete. TurboHash achieves the highest *Delete* throughput because it has the lowest I/O volume due to its shorter and sequential search paths Note that *Delete* is not implemented in CCEH.

4.3 Latency Measurements

In this section, we evaluate the read/write latency of all hash tables. We use 16 threads to write 120 million KV items. Each thread sends 10 million read requests.

As shown in Figure 7, TurboHash almost always has the lowest latency among the hash tables in different types of workloads (*Positive Read*, *Negative Read*, and *Insert*). By applying the linear probing and using only necessary probing length, TurboHash achieves the lowest I/O volume. It also avoids random access to the PMEM. As shown in Figure 8, CCEH16 has 4× I/O volume as much as that of TURBO16 for *Negative Read*, because it reads more buckets than necessary to find a non-existent key. The extra I/O leads to the higher latency.

In most of the experiments, TURBO16 outperforms all the others. It is because TurboHash only probes the necessary buckets during a search. CCEH30 has 3×, 5×, and 5× more I/O volume than TURBO30 for *Positive Read*, *Negative Read*, and *Insert*, respectively. In addition to its large read volume and random access, CCEH30 amplifies write volume in its support of 15-byte atomic writes with undo logging. Thus, CCEH30 has the largest I/O traffic and the worst latency for 30-byte KV items. CLEVEL30 has 3× higher average *Insert* latency than TURBO30. There are several reasons. First, it carries out the search from the bottom to the top levels of its multi-level hash table before an insertion can be carried out. That is, a fixed overhead is added to any *Insert*. Second, its bucket size (64 byte) does not match the PMEM access block size (256 bytes). Therefore, each read to a bucket introduces 4× read amplification. As we can see in the left graph of Figure 8, CLEVEL30 has 2.5× more I/O volume than TURBO30. Third, as buckets on the search path are non-contiguous in the PMEM, CLEVEL30 introduces more random access on the PMEM, which compromises latency. As a consequence, both the write latency and throughput are compromised, as shown in Figure 7a and the left graph of Figure 8.

For average *Positive Read* latency, TURBO30 is about 5% lower than CLEVEL30. The improvement is not significant even though CLEVEL30 has more I/O and random access. This is because current load factor for both hash tables is around 50%, which means that most of keys can be found at their home buckets in the search paths. The performance gap is more noticeable for *Negative Read*. We see 2× lower average read latency for TURBO30 compared with CLEVEL30, which has to search all the buckets (around 4 to 8) at random locations. CCEH30 has the highest read latency. As shown in the right graph of Figure 8, CCEH30 generates some write volume at the PMEM during service of read requests. It places a read lock in the PMEM and frequently updates the lock, which degrades its latency.

CLHT30's latency is similar to that of CLEVEL30. For *Insert*, it also reads all the buckets at random locations along its search path (a linked list) before any insertion. So the I/O amplification and write latency are both high. As a consequence, the write throughput becomes lower, which is only 25% of TURBO30, as shown in the left graph of Figure 8.

4.4 Results of YCSB Benchmarks

The Yahoo! Cloud Serving Benchmark (YCSB) [7] is a popular benchmark used to evaluate performance of NoSQL databases. We wrote a test bench for the hash tables to support YCSB benchmarks, which generate uniform workloads of different access behaviors ("Load", "A",.."F") (see Fig 9). As the results for skewed workloads are similar to that for uniform workloads, we omit their results due to space limits. We run 20 threads and send 10 million requests in each workload. For the all-write workload ("*Load*") that loads 120 million

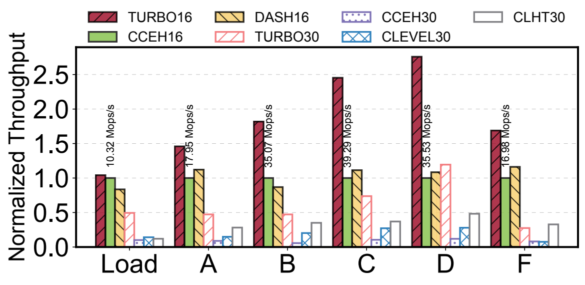


Figure 9: Throughput normalized to CCEH16. Load: 100% write. A: 50% write, 50% read, B: 5% write, 95% read, C: 100% read, D: 5% write, 95% read the latest, F: 50% read-modify-write, 50% read.

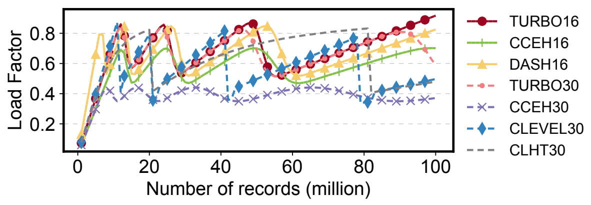


Figure 10: Load factor per one million inserts. Hash tables are initialized with a capacity of holding 12 million items.

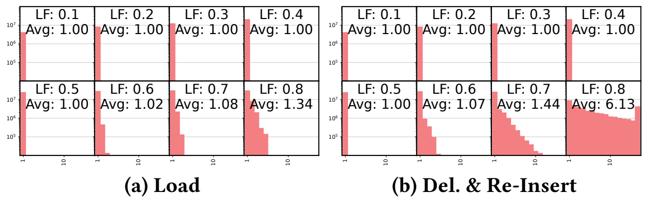


Figure 11: Histogram and average probe distance (Avg) for non-existent keys at varying load factors (LF). (a) shows histograms after loading keys to reach different load factors. (b) shows results after deleting and reloading keys. The probing scope is 16 buckets.

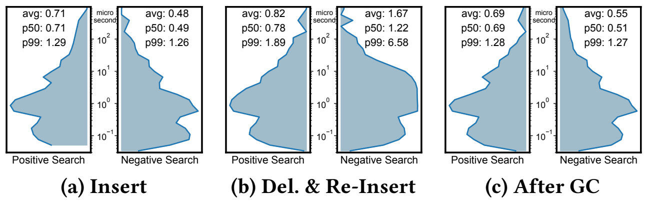


Figure 12: Read latency histograms at 0.8 load factor. (a) Loading new keys. (b) Reload after deletion. (c) After garbage collection.

items to the hash table, as expected TurboHash (either TURBO16 or TURBO30) has a higher throughput than others, as shown in Fig 9. Each of the other workloads starts after the "*Load*" and consists of only or mostly read requests. For all the workloads, TurboHash has up to 2.6× higher throughput than CCEH. CCEH30 has the lowest throughput, as it always requires a bucket lock for reads. The extra write due to lock acquisition during a search compromises its performance. TURBO30 has 1.5×-2.5× higher throughput over CLHT30 and CLEVEL30, except for workload F. Workload F contains half of update requests, and as mentioned in Section 4.2, TURBO30 introduces more I/O than CLHT30 because of CLHT30's out-of-place update to support variable-size key-value pairs.

4.5 Load Factor

Load factor is a critical metric measuring a hash table's space efficiency. One may trade space efficiency for access performance of a hash table. TurboHash achieves both space efficiency and high performance with short search path and sequential PMEM access. In this section we compare load factor variations during insertion of 100 million KV items in the hash tables. The results are shown in Figure 10.

The maximum load factor of CCEH30 is less than 44% because it only probes at most 4 buckets before a hash collision occurs. CCEH16 has a higher peak load factor (70%) because the implementation optimized by its inventors enables double hashing inside the segments. Though both DASH and TurboHash achieve the highest peak load factor (85%), TurboHash has a much higher throughput (see Figure 6). Though CLEVEL30 and CLHT30 can achieve similarly high peak load factor (80%), they allow more flexible placement of KV items for collision resolution, which leads to probing at random PMEM locations and compromises access performance. TURBO30 always searches buckets sequentially for strong spatial locality, which produces up to 8× higher performance in terms of both throughput and latency (see Figure 7).

4.6 Probing Distance

To evaluate the probing efficiency in terms of probing distance, we measure the distance with the searching of non-existent keys in TurboHash under different load factors. As shown in Figure 11, though the probing distance increases with the load factor, its impact on the distance is limited. First, the increase even with a large load factor is still small. For example, with the 80% load factor, the longest distance is only 6 (much shorter than the probing scope size, which is 16) and the average distance is 1.34. Second, long distances hold only a small fraction of all of the distances in a histogram (note that the graph is of a logarithmic scale).

As slots with deleted keys do not break a search path, they may unintentionally make the path longer and compromise probing efficiency. To observe the effect, for each of the hash table shown in Figure 11(a), we delete all of its keys and then insert another set of keys to the same load factor level. We then redo the experiments to obtain a new set of histogram graphs (see Figure 11(b)) Now with a high load factor, such as 80%, the average probe distance for non-existing keys increases to 6.13, which causes the average negative-read latency increases to 1.67μs (from 0.48μs), as shown in Figures 12(a) and (b). However, this experiment represents a rare scenario. And an easy remedy exists. When it is determined that there are too many deleted keys in a shard, TurboHash may conduct a garbage collection operation, which is a special shard rehashing, to remove the deleted keys. The only difference of this special shard rehashing from a regular one is that it will keep the new shard of the same size. Figure 12 shows the latency histograms before and after a delete-and-reinsert, as well as after a garbage collection. It shows that while a delete-and-reinsert operation can increase the latency, especially for the long-tail latency of negative search, its good performance is recovered after a garbage collection.

4.7 Shard Rehashing

The rehashing time is proportional to the shard size. Each rehashing doubles number of buckets in the shard, and each key will be relocated to a bucket in the new shard. Because the shard size is doubled, TurboHash can simply double the index of a bucket in the old shard (a bucket array) to obtain the index of a bucket in the new shard where all of its keys are relocated. Even if a key is larger than 8 bytes and the full key is not in the bucket, the relocation doesn't involve additional PMEM access because the hashed keys are stored in the table. For a shard of 8192 buckets, the average rehashing time is 4 ms. While the rehashing operation and its impact on the tail latency can hardly be avoided in any hash table design, TurboHash has its unique design that helps to bound its impact. TurboHash assumes a large number of shards in its design by over-provisioning. For a hash table of 128K shards and each shard initially of 16 buckets, it can store 10 billions 16-byte KV items when a shard is resized to 8192 buckets. At this time the hash table occupies 256GB PMEM and 1 MB DRAM for a directory of 128K shards. The 1MB directory can be effectively buffered in the LLC cache for high performance. For hosting a large KV store, the directory can be comfortably set at a larger size to accommodate more relatively smaller shards with little concern on its DRAM demand.

5 RELATED WORK

While hash tables have been studied in many works [14, 21, 22, 29, 34, 35], TurboHash's focus is on the design of a high-performance dynamic hashing by fully exploiting Intel Optane DC PMEM's block-access-like performance characteristic. As commercial persistent memory has been available in recent years, a well-designed hash table for persistent memory is on demand to leverage PMEM to provide high performance service. There have been some works on designing hash tables for PMEM, such as PFHT [9], Level hashing [45], CLevel hashing [6], CCEH [27], and P-CLHT [24]. None of the works are aware of and accommodate the performance implication of the PMEM's internal 256B-block access.

PFHT [9] is a variant of cuckoo hashing for persistent memory that avoids cascading writes by allowing only one cuckoo displacement. It uses a stash to prevent full resizing and improves the load factor. Level hashing[45] has two hash tables, making it a two-level structure (top level and bottom level). It adopts a fine-grained locking and each insertion may lock up to two slots. For both PFHT and Level hashing, there can be at least two block writes when an insertion causes the relocation of existent kV items to make room for new items. Unlike them, TurboHash only has one 256B block write, and therefore has better write performance.

CLEVEL hashing[6] is a variant of the level hashing. It has eight 8-byte slots in the 64-byte bucket. Each slot stores the pointer to a KV item. It uses the Compare-and-Swap (CAS) primitive to atomically change the pointer from the old version to the new version to support lock-free reads. As we have mentioned, writing a new version and updating the pointer lead to two block accesses. P-CLHT[24] is a linked-list-based persistent hash table. Each bucket of P-CLHT is of 64 bytes and can store three 16-byte KV items. This may introduce 4× read amplification in the PMEM as the bucket size is 64 bytes, a quarter of the PMEM's block size. Different from them, TurboHash confines the data and metadata within one 256B PMEM block to minimize access to the raw PMEM for higher throughput.

CCEH [27] is a linear-probing-based hash table. It needs to search all of the buckets on the probing path to know if a key exists in the hash table because *Delete* in CCEH leaves holes on the path. To avoid holes during linear probing, "deletion can be implemented by rearranging the elements" [22]. However, this may require two 256B-block writes in the PMEM for the rearrangement: one is removing the tail element of the search path, and the other is overwriting the deleted element with the tail element. FolkloreHT [26] uses dummy elements (tombstones, replacing the key with DEL_KEY mark) for deletion to avoid the rearrangements. However, the deleted space will not be available for future insertions. Inconsistencies may occur if it attempts to reuse the deleted space because concurrent readers may read the deleted value when a concurrent writer replaces the DEL_KEY with new key. Unlike them, TurboHash establishes the search path and avoids two 256B block writes in deletion with the help of its near-place update.

Dash [25] proposes 256-byte bucket design, similar to TurboHash. However, they introduce stash buckets and re-displacing KV items when a hash collision happens, which causes more write/read amplification. IcebergHT [33] consists of three levels. Level 1 is a static table where items are hashed to a single bucket. Level 2 is similar to a cuckoo hash table where items are hashed to two buckets. Level 3 consists of a simple chaining hash table. A negative search in IcebergHT has to read at least 4 buckets, while TurboHash just reads around 2 buckets on average on the search path.

6 CONCLUSIONS

We introduce TurboHash, a persistent hash table designed for high-performance key-value stores in this paper. By enabling out-of-place update at a cost equivalent to that for an in-place write, conducting probing on a path sequentially and only for a necessary length, and utilizing Intel Optane DC's hardware feature, TurboHash minimizes the PMEM I/O traffic and achieves 2× to 8× improvement of access performance over state-of-the-art PMEM hash table designs in terms of both throughput and latency.

Source code of our TurboHash implementation can be found at *https://github.com/hansonzhao007/TurboHash*.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported by the US National Science Foundation under grant CCF-1815303.

REFERENCES

[1] [n.d.]. Diablo Memory Channel Storage. https://www.vladan.fr/ssd-storage-closer-to-cpu-thats-memory-channel-storage-by-diablo-technologies/.
[2] [n.d.]. SanDisk ULLtraDIMM. https://en.wikipedia.org/wiki/ULLtraDIMM.
[3] [n.d.]. Viking Technology SATADIMM. https://www.prnewswire.com/news-releases/viking-technology-satadimm-increases-ssd-capacity-in-solidfires-storage-system-219244711.html.

[4] Austin Appleby. 2008. Murmurhash. https://sites.google.com/site/murmurhash/.

[5] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. 2016. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 281–294. https://www.usenix.org/conference/atc16/technical-sessions/presentation/breslow

[6] 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)*. USENIX Association, online, 799–812. https://www.usenix.org/conference/atc20/presentation/chen

[7] 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* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[8] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 631–644.

[9] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. 2015. Revisiting Hash Table Design for Phase Change Memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads* (Monterey, California) *(INFLOW '15)*. Association for Computing Machinery, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/2819001.2819002

[10] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 1414–1425. https://doi.org/10.14778/1920841.1921015

[11] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible Hashing—a Fast Access Method for Dynamic Files. *ACM Trans. Database Syst.* 4, 3 (Sept. 1979), 315–344. https://doi.org/10.1145/320083.320092

[12] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 371–384. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan

[13] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report UCAM-CL-TR-579. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-579

[14] H. Gao, J.F. Groote, and W.H. Hesselink. 2004. Almost wait-free resizable hashtables. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 50–. https://doi.org/10.1109/IPDPS.2004.1302969

[15] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In *Distributed Computing*, Gadi Taubenfeld (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 350–364.

[16] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent Memory Hash Indexes: An Experimental Evaluation. *Proc. VLDB Endow.* 14, 5 (Jan. 2021), 785–798. https://doi.org/10.14778/3446095.3446101

[17] Intel. [n.d.]. Persistent Memory Development Kit (PMDK). https://pmem.io/pmdk/.

[18] Intel. 2021. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.

[19] 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)*. USENIX Association, Boston, MA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[20] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 993–1005. https://www.usenix.org/conference/atc18/presentation/kannan

[21] Don Knuth. 1963. Notes On "Open" Addressing.

[22] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., USA.

[23] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 773–787. https://www.usenix.org/conference/atc21/presentation/krishnan

[24] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[25] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (April 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[26] Tobias Maier, Peter Sanders, and Roman Dementiev. 2019. Concurrent Hash Tables: Fast and General(?)! 5, 4 (2019). https://doi.org/10.1145/3309206

[27] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 31–44. https://www.usenix.org/conference/fast19/presentation/nam

[28] Fan Ni and Song Jiang. 2019. RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-Based Deduplication Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 220–232. https://doi.org/10.1145/3357223.3362731

[29] Jesper Puge Nielsen and Sven Karlsson. 2016. A Scalable Lock-Free Hash Table with Open Addressing. 51, 8 (2016). https://doi.org/10.1145/3016078.2851196

[30] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[31] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[32] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *J. Algorithms* 51, 2 (May 2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[33] Prashant Pandey, Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, Guido Tagliavini, and Rob Johnson. 2022. IcebergHT: High Performance PMEM Hash Tables Through Stability and Low Associativity. *arXiv preprint arXiv:2210.04068* (2022).

[34] Swapnil Patil and Garth Gibson. 2011. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, Vol. 11. USENIX Association, San Jose, CA, 13–13. https://www.usenix.org/conference/fast11/scale-and-concurrency-giga-file-system-directories-millions-files

[35] Frank Schmuck and Roger Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (Monterey, CA) *(FAST '02)*. USENIX Association, USA, 19–es.

[36] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. Nap: A Black-Box Approach to NUMA-Aware Persistent Memory Indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 93–111. https://www.usenix.org/conference/osdi21/presentation/wang-qing

[37] Xingbo Wu, Fan Ni, and Song Jiang. 2017. Search Lookaside Buffer: Efficient Caching for Index Data Structures. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 27–39. https://doi.org/10.1145/3127479.3127483

[38] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the Performance of Intel Optane Persistent Memory: A Close Look at Its on-DIMM Buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 488–505. https://doi.org/10.1145/3492321.3519556

[39] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Kenry Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory. *Proc. VLDB Endow.* 14, 10 (2021), 1872–1885. http://www.vldb.org/pvldb/vol14/p1872-yan.pdf

[40] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[41] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, online, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[42] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 897–912. https://www.usenix.org/conference/atc19/presentation/zhang-lu

[43] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. ChameleonDB: A Key-Value Store for Optane Persistent Memory. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 194–209. https://doi.org/10.1145/3447786.3456237

[44] Pengfei Zuo and Yu Hua. 2018. A Write-Friendly and Cache-Optimized Hashing Scheme for Non-Volatile Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 5 (2018), 985–998. https://doi.org/10.1109/TPDS.2017.2782251

[45] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 461–476. https://www.usenix.org/conference/osdi18/presentation/zuo

[46] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 15–29. https://www.usenix.org/conference/atc21/presentation/zuo

[47] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 128 (Oct. 2019), 26 pages. https://doi.org/10.1145/3360554