A Read-Efficient and Write-Optimized Hash Table for Intel Optane DC Persistent Memory

Zexuan Li, Kaixin Huang*

[a]*Shanghai Jiao Tong University, Shanghai, China*
[b]*JaguarMicro Inc., Shanghai, China*

Abstract

Emerging non-volatile memory technologies are driving the next generation of storage systems and durable data structures. Among them, many hash table proposals employ NVM as the storage layer for both fast access and efficient persistence. Most of them are based on the assumption that NVM has cacheline access granularity, poor write endurance, DRAM-comparable read latency and relatively higher write latency. However, a commercial non-volatile memory product, namely Intel Optane DC Persistent Memory (Optane), has some interesting features that are different from previous assumptions, such as 256-byte OptaneLine access granularity, higher read latency than DRAM and DRAM-comparable write latency, limited read/write bandwidth, and hardware-layer wear-leveling. Confronted with the new challenges brought by Optane, we propose Rewo-Hash, a novel read-efficient and write-optimized persistent hash table. Our incremental contributions over our previous work are summarized as follows. First, provide a more detailed technical description for cached table-inclined read mechanism and log-free atomic write mechanism. Second, we devise a consistent shadowing synchronization scheme to mask the data synchronization overhead. Third, we propose a non-blocking lightweight resizing scheme and elaborate the crash recovery mechanism. Fourth, we conduct a comprehensive analysis of the implications of Intel ceasing the production of Optane, and provide a forward-looking perspective on the future of non-volatile memory. The experimental results show that compared with state-of-the-art NVM-Optimized hash tables, Rewo-Hash gains remarkable performance improvement.

Keywords:
non-volatile memory, hash table, indexing

1. Introduction

Due to the significant challenges in density scaling and power leakage of traditional DRAM technology, emerging non-volatile memory (NVM, also termed as persistent memory) technologies such as PCM [1], STT-RAM [2] and 3D XPoint [3] are promising candidates for building future memory systems. In addition to non-volatility and byte-addressability, academia assumes that NVM also has DRAM-like cacheline access granularity, low read latency, high write latency, and poor write endurance [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18].

Since hash tables are flat data structures that are able to achieve constant lookup time complexity (i.e., O(1)) [19, 20, 21, 22], a lot of researchers focus on designing NVM-optimized hash tables to obtain both fast access and efficient persistence. For example, PFHT [23], Path-Hash [7] and Level-Hash [8] are proposed to make key-value items durable into NVM with fast read/write operations and wear-leveling guarantee. CCEH [9] is designed to support efficient resizing while maintaining the benefit of constant lookup time.

However, all these designs are developed prior to the availability of the hardware, and are based on anticipated behaviors of NVM, which are 1) DRAM-like cacheline access granularity, 2) DRAM-comparable read latency and relatively higher write latency, 3) DRAM-comparable read and write bandwidth, and 4) poor write endurance. Unfortunately, these assumptions do not accord with the characteristics of the latest commercial NVM product, Intel Optane DC Persistent Memory (Optane for short). As a recent empirical paper [24] demonstrates, Optane has four highlighted features. First, Optane is accessed in block granularity (i.e., 256-byte OptaneLine), which is much larger than the cacheline size (i.e., 64 bytes). Second, Optane has relatively higher read latency (2x or 3x DRAM) but lower write latency (1x DRAM). Third, the bandwidth of Optane for both read and write is limited and has poor scalability. Finally, write endurance may not be an urgent issue in software layer [25].

Existing persistent hash tables may not achieve the expected performance due to the lack of consideration for Optane's unique features. Concretely, they may suffer from slow read/write latency with 1) probing multiple NVM blocks for high load factor and wear-leveling, 2) overhead of logging and persistence ordering, and throughput scalability issue bottlenecked by the bandwidth of Optane. Therefore, we choose to revisit the persistent hash table design for Optane and other Optane-like NVM products and propose Rewo-Hash to address the bottlenecks mentioned above. In storage organization, Rewo-Hash is decoupled into two parts: an in-NVM *Persistent Table* and an in-DRAM Cached Table. Both tables utilize bucket-based cuckoo

*Kaixin Huang is the corresponding author.
Email addresses: `lizx_17@sjtu.edu.cn` (Zexuan Li), `kerwin.huang@jaguarmicro.com` (Kaixin Huang)

hashing, as in PFHT [23] and Level-Hash [8]. The differences of using cuckoo hashing are: 1) Rewo-Hash sets the bucket in NVM as block-aligned (i.e., 256 bytes per bucket) and 2) Rewo-Hash relaxes the limitation on iterative evictions to two. The design of *Cached Table* is to bypass the high read latency of Optane for key probing which exist in both search and write requests. To reduce the flush/fence overhead and bandwidth consumption, we utilize the atomicity rule of CPU's 8-byte write to fulfill log-free and atomic writes to *Persistent Table*. Also, we optimize the synchronization overhead between *Persistent Table* and *Cached Table* for data coherence. A previous version of Rewo-Hash [26] is published in DATE and this paper extends our previous work with the following incremental contributions.

• We provide a more detailed design and implementation of cached table-inclined read (CATI) mechanism and log-free atomic write (LOFA) mechanism. We also propose a consistent shadowing synchronization scheme between the *Persistent Table* and *Cached Table*.

• We devise a non-blocking lightweight resizing scheme for Rewo-Hash. During the entire resizing procedure, only about half of the total items need to be migrated and in the meanwhile, the throughput drop can be very limited. We also detailedly analyze the crash recovery mechanism of Rewo-Hash.

• We conduct a comprehensive analysis on the implications of Intel ceasing the production of Optane, and provid a forward-looking perspective on the future of non-volatile memory.

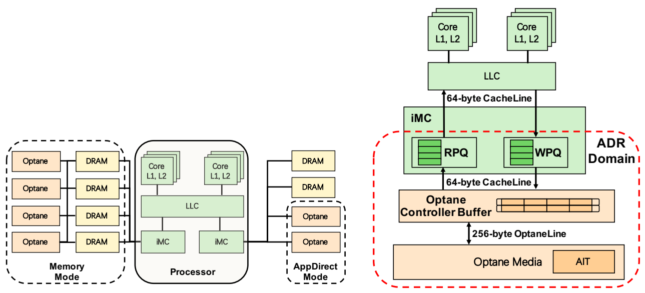
• We conduct extensive experiments to evaluate the performance of Rewo-Hash using both DRAM-simulated and Optane-based platform. Results show that Rewo-Hash outperforms its counterparts by 1.57x-3.39x for search and 1.04x-2.98x for write in terms of latency. The concurrent throughput of Rewo-Hash is also improved to 1.70x-13.7x for different YCSB workloads.

Organization. The remainder of this paper is organized as follows. Section 2 introduces the background and motivation of our work. Section 3 describes the design details of Rewo-Hash. In Section 4, we present a comprehensive analysis of the implications of discontinuing Optane and the future development of NVM technologies. Section 5 presents experimental results. We review related work in Section 6 and conclude this paper in Section 7.

2. Background and Motivation

2.1. Non-Volatile Memory

Emerging non-volatile memory (NVM) technologies such as PCM [1], STT-RAM [2] and 3D XPoint [3] are promising to close the gap of performance and capacity between low-latency, volatile memory technologies (e.g. DRAM) and high-capacity, persistent storage technologies (e.g. disk, SSD, Flash). Attaching NVMs to the main memory bus provides a raw storage medium that can be orders of magnitude faster than modern persistent storage medium such as disk and SSD [1, 2, 3]. For NVM, data consistency is a significant issue. There are two reasons. First, current processors only supports 8-byte atomic write. If the size of update area is larger than, then only a part of



(a) Working Mode of Optane (b) Data Path of Optane

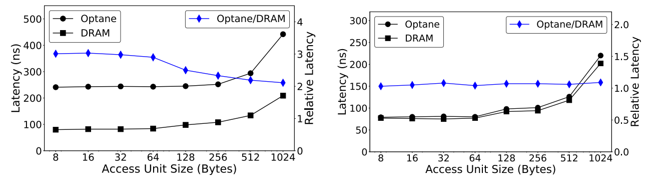
Figure 1: Overview of Optane Platform (figures are adapted from work [24])

the whole data structure may be persisted in NVM during system crashes, causing inconsistent partial writes [27, 28]. Second, current CPU and caches may reorder data writes to memory, without respect to program order. For non-exchangeable data updates in a transaction, wrong order may occur when crash happens, which certainly leads to data inconsistency [29, 30, 31]. Most of existing works utilize *clflush/clwb+mfence* primitives to guarantee persistence ordering [32, 8, 33, 4] and adopt logging techniques to support atomic update [5, 6, 13, 34, 35] in NVM. Since real NVM platforms are not available in previous works, researchers always used DRAM to simulate NVM based on a few common assumptions, which includes: 1) DRAM-like cacheline access granularity 2) high write latency but low read latency, 3) DRAM-comparable read/write bandwidth and 4) poor write endurance [7, 8, 5, 6, 36, 4, 23, 11, 37, 38].

2.2. Intel Optane DC Persistent Memory

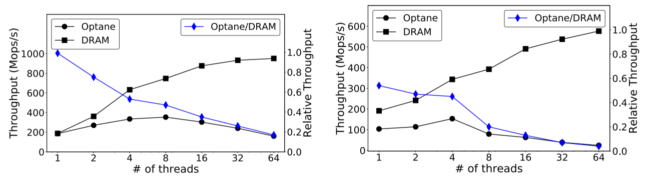
After nearly a decade of anticipation, non-volatile memory DIMMs are finally commercially available with the release of Intel Optane DC Persistent Memory [39, 40]. However, this commercial product has invalidated our knowledge about the characteristics of NVM.

Figure 1 shows the architecture of Optane-based platform. As shown in Figure 1(a), Optane is connected to the processor's integrated memory controller (iMC) through the memory bus, which is similar to traditional DRAM. Currently a processor die can support six Optane DIMMs at most across its two iMCs [41]. Optane can be configured in two working modes: memory mode and AppDirect mode. Memory mode uses Optane to extend the capacity of the main memory without logical persistence guarantee. In other words, although data flushed into Optane can be physically persistent, the operation system does not promise that such data can be probed after reboot. In this mode, Optane is placed on the same memory channel as conventional DRAM DIMM, which serves as a direct-mapped cache with 4-KB cache block size. The processor's memory controller manages the cache block transparently. From the perspective of operation system, Optane is simply regarded as a larger volatile main memory. In contrast, AppDirect mode supports persistence and does not utilize DRAM as a transparent cache. In this mode, Optane works as a separate non-volatile



(a) Read Latency (b) Write Latency

Figure 2: Access Latency of Optane



(a) Read Bandwidth (b) Write Bandwidth

Figure 3: Bandwidth of Optane

memory device that is paralleled with DRAM. Persistent memory management systems, PM-aware file systems or other management layers can provide allocation, namespace, and access control to persistent data. In this paper, we focus on AppDirect mode of Optane since data persistence is crucial for our design.

To ensure better persistence efficacy, the iMC is placed within the asynchronous DRAM refresh (ADR) domain, as illustrated in Figure 1(b). Intel's ADR feature promises that any stores that reach the ADR domain can survive across a system crash (i.e., data will be flushed to the Optane media within a hold-up time) [24]. The iMC maintains read pending queues (RPQs) and write pending queues (WPQs) for each single Optane DIMM to optimize read and write for persistent data. WPQs are within the ADR domain, and hence when data reaches WPQs, it can be considered as (logical) persistent data.

Memory accesses to Optane are coordinated by an on-DIMM Optane controller, which serves as a fast middle layer for reads and writes. The iMC communicates with the Optane controller via the DDR-T interface in conventional cacheline granularity (i.e., 64 bytes). However, the Optane controller transmits data to or loads data from the Optane media in 256-byte OptaneLine granularity. As such, the Optane controller is responsible for combining adjacent cacheline-aligned writes from the upper layer (e.g., LLC, iMC) into OptaneLine granularity whenever possible. A small write-combining buffer is maintained in the Optane controller. However, as the Optane media access granularity is 256 bytes, the Optane controller may cause read and write amplification issues.

Similar to SSDs and Flash media, Optane also maintains an address translation component, named address indirection table (AIT) for wear-leveling and bad block management [42]. It is reported by Intel that Optane will not be worn out for five years with even 350 PB (for a 256 GB module) lifetime writes [25].

A bad news comes in 2022 that Intel announced their Optane product line would be discontinued in favor of recent trends toward Compute Express Link (CXL) [43]. We expect that while Optane was abandoned, research based on it still provides valuable insights for future non-volatile memory products. We discuss the effects and anticipation for non-volatile memory in Section 4.

Based on the description for Optane above and published empirical evaluation reports [41, 24] on it, we argue that Optane's features are different from previous assumptions in four aspects. First, despite of byte-addressability, Optane is accessed in block granularity (i.e., 256-byte OptaneLine), which is much larger than the DRAM cacheline size (i.e., 64 bytes). Thus, Optane has better performance for sequential reads than random reads. Second, Optane has relatively higher read latency (2x-3x DRAM) but lower write latency (1x DRAM). Third, Optane has a limited bandwidth scalability for both reads and writes (1/3 DRAM for read and 1/8 DRAM for write with 16 threads). Fourth, Optane's write endurance is not of critical concern.

To verify the performance nature of our own Optane platform, we have conducted a series of experiments which randomly read from and write to Optane (the system configuration is given in Section 5) with different granularities. The latency results are illustrated in Figure 2, from which we have two key observations. First, the read latency of Optane is about 3.0x of DRAM when the access granularity is small (e.g., smaller than 64 bytes) while keeping steady (about 2.1x) when the access granularity is larger. It implies that the large block access granularity (256 bytes) of Optane causes higher latency for random access of small-grained data. Second, the write latency of Optane is comparable to DRAM, with about merely 1.03x-1.09x DRAM latency among all different access unit sizes. The reason is that writes to Optane are durable immediately when data enters the ADR domain, which guarantees that data will be flushed to the Optane device media should there be a system crash [41]. Figure 3 shows the bandwidth results of Optane for ten million 8-byte random reads/writes. We make two observations of the results. First, for both read and write, Optane suffers from poor scalability for concurrent throughput. For instance, read from Optane has the same throughput (about 180 Mops/s) as DRAM with one thread; but when the number of threads increases to 8, Optane obtains about 40% throughput of DRAM. With 64 threads, the ratio drops to 20%. Second, write bandwidth of Optane is much lower than read and has more severe scalability problem. When the thread number is 1, Optane obtains only 50% throughput of DRAM. The peak throughput is gained with four threads, reaching about 350 Mops/s, which is about 40% of DRAM's throughput. With the number of threads increasing to 64, the throughput of Optane finally drops to 4% of DRAM (24 Mops/s), which is even 80% lower than the single-thread case.

The results of low write latency and poor write throughput of Optane may seem controversial. However, this can be observed and explained by revisiting the data path of Optane (see Figure 1(b)). Writes to Optane are firstly pushed into the Optane controller, which communicates with the iMC and LLC via DDR-T interface. Therefore, the idle latency for Optane write can be ultra-low. However, since the write-combining buffer size of Optane is limited, the throughput will be bottlenecked by the bandwidth from the Optane controller to the physical Optane media.

2.3. Cuckoo Hashing

Cuckoo hashing [20] is a form of open addressing in which each non-empty entry of hash table contains a key-value item. The basic idea of cuckoo hashing is to resolve collisions by using k hash functions ($k = 2$ is common in use) instead of only one. This provides k possible locations in the hash table for each key. Lookup requires probing just k locations in the hash table, which takes constant time in the worst case. This is in contrast to many other hashing algorithms, which may not have a constant worst-case bound on the time to do a lookup. Deletions, also, can be performed by blanking the entry containing a key, in constant worst case time.

When a new key should be inserted, and all k entries have been already full, it will be necessary to move other keys to their alternative locations (or back to their first locations) to make room for the new key. A greedy algorithm is used: the new key is inserted in one of its k possible entries, evicting any key that might already reside in this location. This displaced key is then inserted in its alternative location, again evicting any key that might reside there. The process continues in the same way until an empty entry is found, completing the algorithm. However, it is possible for this insertion process to fail, by entering an infinite loop or by finding a very long chain (longer than a preset threshold). In this case, the hash table should be rebuilt using a new hash function or resized to a larger table.

With k-way hashing and iterative evictions to resolve collisions, cuckoo hashing can obtain high occupancy (i.e., maximum load factor) and is widely used for many storage applications [44, 45]. That's why we adopt it as our building block. The variant we make is similar to some existing proposals [23, 46]: we use a bucket-based cuckoo hashing design, where each bucket can allow multiple key-value items to reside.

2.4. Limitations of Existing NVM-Optimized Hash Tables

Existing NVM-optimized hash tables, such as PFHT [23], Path-Hash [7], Level-Hash [8] and CCEH [9], are optimized for emerging non-volatile memory. Concretely, PFHT [23] is a write-friendly hash table for PCM that adopts bucket cuckoo hashing as its indexing scheme only allows one-time eviction for write collision to reduce writes to persistent memory. In order to improve the capacity occupancy, PFHT uses a stash (i.e., linear array) to store the items failing to be inserted into the hash table. Path-Hash [7] logically organizes the buckets in the hash table as an inverted complete binary tree. Each bucket stores one item and only the leaf nodes are addressable. When a hash collision occurs in the leaf node of a path, all non-leaf nodes in the same path can be used to store the conflicting key-value item. Level-Hash [8] makes two major modifications based on Path-Hash. First, it contains only two levels for storing items: one addressable top level and one hidden bottom level; each bottom-level bucket acts as a candidate bucket to address collisions for two top-level buckets. Second, each bucket is set as cacheline-aligned to hold multiple key-value items, which enhances data locality. Third, it adopts 2-way cuckoo hashing for both top-level buckets and bottom-level buckets and at most one eviction is allowed to resolve a hash collision. CCEH [9] aims at minimizing the resizing cost by utilizing a dynamic hashing scheme. It is made up of multiple bucket-based segments and while one bucket in a segment is full, a new segment can be generated dynamically to resolve hash collisions. As such, only about half of the records in the old segment need to be migrated.

However, these proposals are all highly dependent on previous assumptions for NVM. First, since the access granularity is assumed to be small (i.e., cacheline granularity), key-value items in these hash tables are either separately-distributed [7] or cacheline-aligned in a bucket [8, 23]. Second, they serve read requests directly in NVM because of the assumption that NVM read latency is comparable with DRAM. Third, they do not consider to reduce unnecessary reads or writes to NVM for reducing bandwidth consumption because they assume that the bandwidth of NVM is similar to DRAM, which can be hard to saturate. Fourth, they design wear-leveling mechanisms by reducing writes to the same NVM buckets or entries. PFHT and Path-Hash introduce high extra lookup overhead due to the large linear stash [23] and multiple separate path buckets [7], respectively. The new features brought by Optane have inspired us to rethink the bottlenecks of previous NVM-optimized hash table designs. We summarize the effects of the Optane features on existing works in Table 1. A detailed explanation is provided as follows.

Block Access Granularity. Since existing works employ much smaller bucket size (e.g., cacheline size, single key-value entry size) than Optane's access granularity, there will be read amplification issue for each bucket access.

High Read Latency. As all the three hash tables serve search requests directly in NVM, they are at the cost of 2x-3x DRAM latency for each single NVM read. To make matters worse, multiple NVM reads may be required for obtaining a request key when the load factor is high.

Low Write Latency. All the proposed hash tables can benefit from this feature because the write access will be faster than expected. However, the low-latency merit can only be achieved for idle workload where bandwidth is underutilized.

Limited Bandwidth. This feature will become the concurrent throughput bottleneck for all existing designs. Concretely, the slow NVM reads will limit the throughput scalability of read-heavy workload. Extra writes caused by data logging or metadata logging will consume the scarce write bandwidth of Optane, hence affecting the write throughput.

Hardware-layer Wear-Leveling. This feature will indirectly decrease the performance of PFHT and Path-Hash since they introduce extra lookup overhead for read requests due to their wear-leveling mechanisms, which can be removed if they are developed on Optane-based platform.

All these limitations of existing NVM-optimized hash tables motivate us to propose a new design to fully exert the potential of Optane while avoiding its drawbacks. As a result, we develop Rewo-Hash, a read-efficient and write-optimized persistent hash table.

Table 1: **Optane Feature Effects on Existing NVM-Optimized Hash Tables.** Inside the table, "✓" represents positive effect; "×" indicates negative effect; "–" means neutral effect.

Features	PFHT [23]	Path-Hash [7]	Level-Hash [8]	CCEH [9]
block access granularity	×	×	×	×
high read latency	×	×	×	×
low write latency	✓	✓	✓	✓
limited bandwidth	×	×	×	×
hardware-layer wear-leveling	×	×	-	-

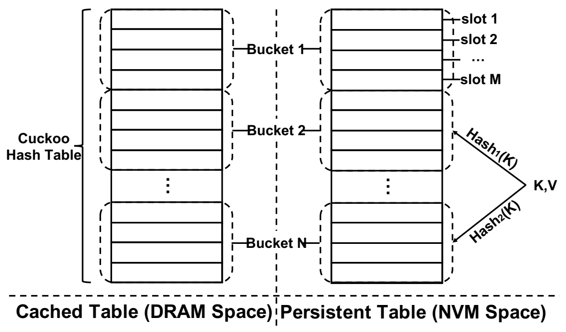


Figure 4: Architecture Overview of Rewo-Hash.

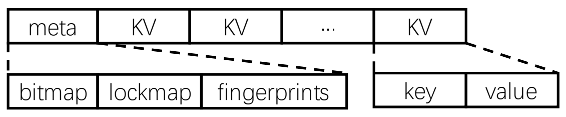


Figure 5: Data Structure of Rewo-Hash Bucket.

3. Rewo-Hash

3.1. Architecture

Considering the characteristics of commercial NVM product, Optane, we have two main design considerations: 1) reduce the read latency and improve the read throughput by keeping the read path in DRAM space and 2) minimize the write overhead by designing a log-free write mechanism.

Figure 4 shows the storage architecture of Rewo-Hash. Rewo-Hash consists of two physical tables, with the *Persistent Table* in NVM space and the *Cached Table* in DRAM space. The *Persistent Table* is mainly used to serve write requests and keep key-value items durable in NVM with fine-grained consistency guarantee. On the contrary, the *Cached Table* is used to serve search requests by keeping a subset, if not all, of the items in the *Persistent Table*.

Both of the *Cached Table* and *Persistent Table* employ *k*-cuckoo hashing as the hashing method and we set *k* as 2. Each table is composed of multiple buckets and each bucket contains several slots. A slot is the basic storage unit for a single key-value item. When data collision occurs for a key to be inserted, a victim slot in the hashed bucket will be chosen and migrated to the other bucket. After such a successful eviction, the current request key will be inserted.

3.2. Hash Table Bucket

The detailed data structure of bucket in Rewo-Hash is shown in Figure 5. In addition to multiple slots to store key-value items, we also add a metadata zone in each bucket (i.e., *meta*), which contains three fields: *bitmap*, *lockmap* and *fingeprints*. Notice that *meta* is an 8-byte data structure, and hence it can be modified atomically. In bucket metadata, *bitmap* records the used/free status of each slot in a bucket, with 1 being used and 0 being free. *lockmap* aims at resolving concurrent write collisions. A bit in *lockmap* indicates if the corresponding slot is being locked or not. Similar to previous NVM-optimized tree designs [47, 48], *fingerprints* is provided for each bucket to reduce unnecessary read overhead for requested keys. Specifically, a fingerprint is an *m*-bit hash of the key stored in the corresponding slot.

Both of the *Persistent Table* and *Cached Table* adopt this bucket data structure, except for two differences. (1) Bucket size: to best utilize the space locality of Optane, the bucket size of the *Persistent Table* is set as 256 bytes (one OptaneLine). As for the *Cached Table*, we set its bucket size as 128 bytes (two cachelines) to exploit cache locality. (2) Use of fingerprints: since cacheline-aligned read can be efficient in DRAM space, the *fingerprints* field is regarded as padding bits and ignored by the *Cached Table* to reduce calculation overhead.

3.3. CATI: Cached Table-Inclined Read

Since the *Cached Table* can keep a data copy of the *Persistent Table*, all search requests will be served in DRAM space first for read efficiency. There are two states for the *Cached Table*: full-loaded and partial-loaded. If the *Cached Table* is in full-loaded state, then it stores all the inserted elements as *Persistent Table* does, and hence each value for a request key should be found in the *Cached Table* directly if it exists. If the *Cached Table* is in partial-loaded state, it only stores a portion of the key-value items in the *Persistent Table*. In this case, we may have a cache miss, and the request would have to be directed to the *Persistent Table*. The search is first served in the *Cached Table*. Only when the *Cached Table* search fails and not all key-value items are cached, *Persistent Table* search should be conducted.

The search procedure in the *Cached Table* is provided in algorithm 1. *k* bucket offsets are calculated based on cuckoo hashing and further lookup occurs in these *k* buckets. The procedure of per-bucket search can be divided into three steps: (1) check each bit of *bitmap* field to verify if the corresponding data slot is valid and compare the *key* field of a valid data slot

```
Algorithm 1: cache_search(key)
1  find_flag ← FALSE;
2  foreach hash_method in k_hash_methods do
3      off ← cache_hash(key, hash_method);
4      tbucket ← CBuckets[off];
5      foreach i in [1 : SLOT_NUM_PER_CBUCKET] do
6          if tbucket.meta.bitmap[i] == 1 and
            tbucket.kv[i].key == key then
7              while tbucket.meta.lockmap[i] == 1 do
8                  // wait until the lock is released
9              htm_begin();
10             if tbucket.meta.bitmap[i] == 1 and
                tbucket.kv[i].key == key then
11                 ret_value ← tbucket.kv[i].value;
12                 find_flag ← TRUE;
13             htm_end();
14             if find_flag == TRUE then
15                 return TRUE;
16 return FALSE;
```

```
Algorithm 2: pm_search(key, ret_value)
1  find_flag ← FALSE;
2  foreach hash_method in k_hash_methods do
3      off ← persistent_hash(key, hash_method);
4      tbucket ← PBuckets[off];
5      foreach i in [1 : SLOT_NUM_PER_PBUCKET] do
6          if tbucket.meta.bitmap[i] == 1 and
            tbucket.meta.fp[i] == fp_hash(key) then
7              if tbucket.kv[i].key == key then
8                  while tbucket.meta.lockmap[i] == 1 do
9                      // wait until the lock is released
10                 htm_begin();
11                 if tbucket.meta.bitmap[i] == 1 and
                    tbucket.kv[i].key == key then
12                     ret_value ← tbucket.kv[i].value;
13                     find_flag ← TRUE;
14                 htm_end();
15                 if find_flag == TRUE then
16                     submit_cache_insert(key, val);
17                     return TRUE;
18                 else
19                     return retry_pm_search(key,
                         ret_value);
20 return FALSE;
```

with the request key (lines 5 to 6); (2) if the *key* matches, check the lock bit of corresponding data slot and only move forward when the data slot is not locked (lines 7-9); (3) start a transaction using hardware transaction memory (HTM) to guarantee atomic read and return the *value* in the corresponding data slot (lines 10 to 18). Step 2 is to avoid reading a non-persisted (or uncommitted) inserted key-value item.

Hardware transactional memory enables all memory operation within one transaction to execute atomically (to other threads). It has also been adopted in some prior NVM-based index designs, such as HiKV [5], FPTree [47] and LB+Tree [49]. However, unlike these works, we only use HTM in Rewo-Hash for search operation, and each transaction is limited within loading two cachelines (i.e., one cacheline including the 8-byte metadata and the other including the data slot), which is lightweight and minimizes the transaction failure rate.

When the request key is not found in the *Cached Table*, an additional search may happen in the *Persistent Table*, as shown in algorithm 2. The search procedure, however, is a little different to *Cached Table* search. The key search will first occur in the *fingerprints* field to check if the fingerprint of the requested key matches with a reserved fingerprint (line 6). Only if they match, the actual read in the corresponding slot will happen. The fingerprint technique can effectively reduce extra cacheline reads into NVM.

To reduce extra NVM reads for negative search (i.e., query keys are not in the hash table), a special metadata *cache_overflow* is kept in Rewo-Hash to represent the state (i.e., full-loaded or partial-loaded) of the *Cached Table*. When the *cache_overflow* field is `FALSE`, it means that the *Cached Table* is full-loaded and all keys have already been stored in the *Cached Table*. When the *cache_overflow* field is `TRUE`, it means that the *Cached Table* is partial-loaded. An additional search in the *Persistent Table* is required if the cache search fails. Fortunately, the *fingerprints* field can also help to filter out unnecessary reads to NVM bucket slots, reducing the overhead of negative search in NVM.

Notice that in Rewo-Hash, keys not found in the *Cached Table* but in the *Persistent Table* should be cached upon a cache read miss (see line 15 in algorithm 2). If there is a hash collision which cannot be resolved through iterative cuckoo evictions and the *Cached Table* reaches the DRAM capacity threshold (i.e., resizing is not allowed), we adopt the LRU algorithm for data replacement. A cold item will be deleted from the hashed bucket and provides a free slot for the requested item. When the cache replacement first occurs, the *cache_overflow* field should be updated to `TRUE`.

Thanks to CATI mechanism, Rewo-Hash can not only achieve low latency for each single search request, but also exploit the high concurrent read bandwidth of DRAM.

3.4. LOFA: Log-Free Atomic Write

In order to take advantage of the nature of low write latency while overcoming the obstacle of limited write bandwidth of Optane, we design a log-free atomic write mechanism. It minimizes the overhead caused by extra writes to the *Persistent Table*, such as redo/undo logging, data flushes (i.e., `clflush/clwb`) and memory fences (i.e., `mfence/sfence`).

Algorithm 3 illustrates the procedure of log-free atomic insert. Similar to the read procedure, Rewo-Hash first calculates the hashed bucket offset. In the candidate bucket, Rewo-Hash

Algorithm 3: pm_insert(key, value)

```
1  foreach hash_method in k_hash_methods do
2      off ← persistent_hash(key, hash_method);
3      tbucket ← PBuckets[off];
4      foreach i in [1 : SLOT_NUM_PER_PBUCKET] do
5          if tbucket.meta.bitmap[i] == 0 then
6              /* atomically lock the corresponding bit in
                 bucket lockmap (if its lock state is free) */
7              if try_cas_lock(tbucket.meta.lockmap, i) ==
               TRUE then
8                  ntstore(&tbucket.kv[i].key, &key,
                     KEY_SIZE);
9                  ntstore(&tbucket.kv[i].value, &value,
                     VALUE_SIZE);
10                 sfence();
11                 bitmap_update(tbucket.meta.bitmap, i);
12                 fp_update(tbucket.meta.fp, i, key);
13                 clwb(&tbucket.meta); sfence();
14                 unlock(tbucket.meta.lockmap, i);
15                 return TRUE;
16 /* evict one item when there is no available free slot */
17 return cuckoo_move(key, value, INSERT);
```

will execute five steps: (1) find an available free slot according to each *bit_flag* information in the *bitmap* field and atomically lock[1] the free slot (lines 5-7); (2) write new key-value data into the locked free slot and persist it (lines 8 to 10); (4) update bitmap, fingerprint and persist the bucket metadata to commit this write (lines 11 to 13). (5) finally unlock the corresponding slot to allow concurrent reads and writes (line 14). When there is no available free slot in both of two candidate buckets due to hash collision or lock contention, a cuckoo eviction (line 24) should be executed to release a free slot in one of the candidate buckets. With the released free slot, the insert operation can be successfully completed.

Notice that the raw method of cuckoo eviction may cause the double record issue, which leaves a persistent inconsistent state across the system crash. A naive idea is to check whether there is a double record for every single key-value item during system restart [8]. However, it will bring heavy burden to the crash recovery algorithm. To resolve this problem, we design the cuckoo eviction in Rewo-Hash to be performed in three steps with strict persistence ordering: 1) lock the slot (i.e., set its corresponding lock bit from 0 to 1 in the *lockmap* field of bucket metadata) which is chosen to be evicted to a candidate bucket; 2) insert the selected slot to the candidate bucket as normal *insert* operation; 3) unlock and invalidate the chosen slot with a single atomic update (one $1 \rightarrow 0$ flip in *lockmap* and one $1 \rightarrow 0$ flip in *bitmap*). With such design, our recovery al-

Algorithm 4: pm_update(key, value)

```
1  foreach hash_method in k_hash_methods do
2      off ← persistent_hash(key, hash_method);
3      tbucket ← PBuckets[off];
4      foreach i in [1 : SLOT_NUM_PER_PBUCKET] do
5          if tbucket.meta.bitmap[i] == 1 and
           tbucket.meta.fp[i] == fp_hash(key) then
6              if tbucket.kv[i].key == key then
7                  if try_lock_cas(tbucket.meta.lockmap, i)
                   == TRUE then
8                      old_off ← i; break;
9                  else
10                     /* may be updated or deleted by a
                         concurrent threads, hence retry */
11                     retry from line 2;
12     /* find a free slot to store the updated key-value */
13     if old_off exists then
14         foreach i in [1 :
           SLOT_NUM_PER_PBUCKET] do
15             if tbucket.meta.bitmap[i] == 0 then
16                 if try_cas_lock(tbucket.meta.lockmap, i)
                   == TRUE then
17                     ntstore(&tbucket.kv[i].key, &key,
                         KEY_SIZE);
18                     ntstore(&tbucket.kv[i].value,
                         &value, VALUE_SIZE);
19                     sfence();
20                     bitmap_update(tbucket.meta.bitmap,
                         old_off, i);
21                     fp_update(tbucket.meta.fp[i], key,
                         i);
22                     clwb(&tbucket.meta); sfence();
23                     unlock(bucket.meta.lockmap,
                         old_off, i);
24                     return TRUE;
25         break;
26 if old_off exists then
27     /* evict one item when there is no available free
         slot */
28     return cuckoo_move(key, value, UPDATE);
29 else
30     /* no corresponding key is found, execute as insert
         */
31     return pm_insert(key, value);
```

[1]The atomic lock is implemented by compare-and-swap (CAS) primitive. CAS guarantees that the target data is successfully modified only when it matches a specified value. Notice that CAS supports atomic update for data within size of 8 bytes.

Algorithm 5: pm_delete(key)

```
1  foreach hash_method in k_hash_methods do
2      off ← persistent_hash(key, hash_method);
3      tbucket ← PBuckets[off];
4      foreach i in [1 : SLOT_NUM_PER_PBUCKET] do
5          if tbucket.meta.bitmap[i] == 1 and
           tbucket.meta.fp[i] == fp_hash(key) then
6              if tbucket.kv[i].key == key then
7                  if try_cas_lock(tbucket.meta.lockmap, i)
                   == TRUE then
8                      bitmap_invalidate(tbucket.meta.bitmap,
                         i);
9                      clwb(&tbucket.meta); sfence();
10                     unlock(tbucket.meta.lockmap, i);
11                     return TRUE;
12                 else
13                     /* may be updated or deleted by a
                         concurrent thread, hence retry */
14                     retry from line 1;
```

gorithm can quickly recognize key-value items that may have double records through its lock status (see Section 3.7).

Algorithm 4 provides the logic of log-free atomic update, which contains several important steps. (1) find the slot location of the request key in the candidate buckets first (lines 1 to 6). (2) atomically lock the slot; (3) select a free slot within the same bucket where the original key-value resides and atomically lock this slot; (4) write the updated key-value item in the free slot and persist (lines 22 to 24). (5) update the corresponding bits for both the old slot and the new slot, the fingerprint of the new slot and then persist the metadata to commit this update operation; (6) finally unlock the two slots to allow concurrent reads and writes for the updated key-value item (line 28). If there is no free slot within the same bucket, cuckoo eviction (line 38) will be triggered automatically to release a free slot. Afterwards, the procedure from lines 22 to 29 will be replayed. Notice that if none of the existing keys matches with the update request, then Rewo-Hash will automatically transform the update operation into an insert operation (line 42).

The procedure of log-free atomic delete is shown in algorithm 5. Deleting a key in Rewo-Hash is much simpler than insert and update operations. Four important steps are required: (1) find the target slot where the key matches with the assistance of the fingerprint (lines 4 to 6); (2) atomically lock the slot which is to be invalidated (line 7); (3) invalidate the slot by setting its corresponding bit of *bitmap* field as 0 and then persist the metadata to commit the delete operation (lines 8 to 9); (4) finally unlock the current slot (line 10). Notice that there is no need to update the corresponding fingerprint, because the fingerprint will never be used when a slot is invalid.

The algorithms shown above are all targeted for concurrent accesses. If only one thread is running, then all the lock and unlock steps are unnecessary. Notice that Rewo-Hash replaces all *store* + *clflush/clwb* instructions for key-value slots that are commonly used in previous works [23, 7, 8, 9] with simple *ntstore* instructions (i.e., non-temporal store in x86 architecture), which are shown in algorithms 3 and 4. That is, all the data slot writes in the *Persistent Table* are conducted in an LLC-bypass method. There are two benefits with utilizing *ntstore* in our design. First, it reduces the LLC space consumption of data items in the *Persistent Table*, leaving more room for the items in the *Cached Table* to be stored in LLC. Therefore, the problem of double cache copy in LLC caused by read-and-modify operation can be minimized. Second, it avoids unnecessary cacheline flushes due to data evictions in LLC or persistence ordering requirements, and hence decreases the write overhead in the critical path. Remember that read requests in Rewo-Hash are served by the *Cached Table* first. With more (hot) key-value items in the *Cached Table* being stored in LLC, the read performance of Rewo-Hash can be further promoted.

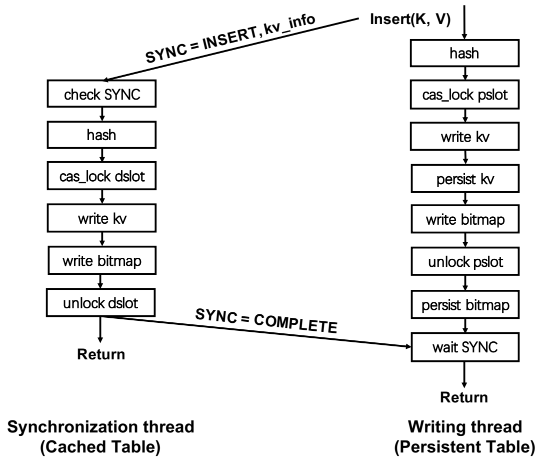
In conclusion, the atomic write mechanism in Rewo-Hash is totally log-free and eliminates the overhead of data logging. Also, by exploiting the atomic write feature on 8-byte *bmeta* field, the Optane write bandwidth consumption is minimized efficiently, which can contribute to higher write throughput. In addition, the log-free atomic write mechanism is also applied to the in-DRAM *Cached Table*. This is because higher concurrency can be supported with this technique. The only difference is that no persistence guarantee (i.e., *ntstore/clwb* + *sfence*) is required for the *Cached Table*. Since an update operation never modifies the *value* field of the original slot, non-blocking reads can be conducted on the requested key-value item when it is being updated without incurring any dirty read or phantom read.

3.5. Shadowing Synchronization

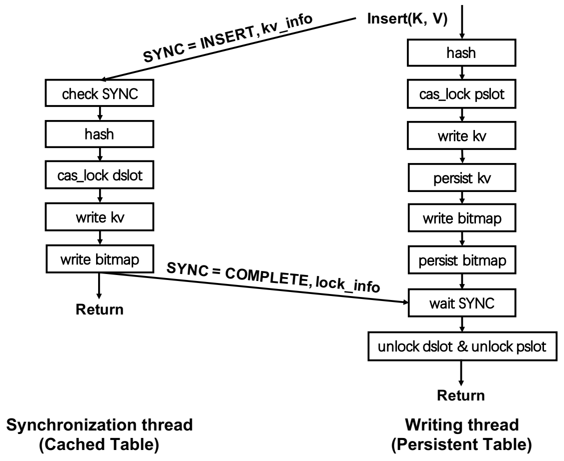
Since we need to guarantee the key-value items in the *Cached Table* consistent with that in the *Persistent Table*, data synchronization is required. A naive idea is to sequentially make modifications to the data in the persistent table and cached table. However, such a simple procedure will incur nontrivial write delay in the critical path due to the DRAM-comparable write latency of Optane.

We propose an efficient and consistent synchronization scheme, named shadowing synchronization, to minimize the overhead of synchronization (i.e., write data to the *Cached Table*) in the critical path. Figure 6 describes the work flow of shadowing synchronization. The main idea is that we employ two types of threads to mask the write latency of synchronization: writing thread (i.e., frontend working thread) and synchronization thread (i.e., backend shadowing thread).

Initially, we design the synchronization scheme as Figure 6(a) presents. The writing thread is responsible for data modifications in the *Persistent Table* while the synchronization thread is used to make update in the *Cached Table*. The writing thread and synchronization thread utilize a *SYNC* signal to communicate with each other. When the writing thread recognizes a new write request, it initiates a *SYNC* signal by marking it as the operation type (i.e., `INSERT`, `UPDATE` or `DELETE`) and then delivers it to the synchronization thread. The synchronization thread executes the same write request in the cached table and only



(a) With Concurrency Consistency Issue



(b) With Full Concurrency Consistency Guarantee

Figure 6: **Procedure of Synchronization Scheme.** The writing thread installs update to *Persistent Table* while the synchronization thread installs update to *Cached Table*.

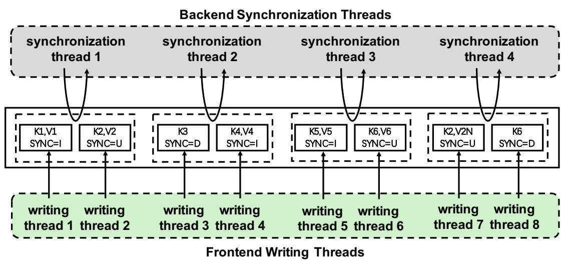


Figure 7: **Frontend-Backend Thread Pool.** The *SYNC* signal has five states: NULL(N), INSERT(I), UPDATE(U), DELETE(D) and COMPLETE(C).

when the execution succeeds, will the *sync* signal be updated to COMPLETE. After successfully executing the write operation in the persistent table, the writing thread will check the state of the *SYNC* signal. The writing thread will not return until the *SYNC* signal is found to be COMPLETE.

However, we observe that such a synchronization scheme may not work correctly for concurrent reads and writes with two tables (i.e., *Cached Table* and *Persistent Table*). For instance, a writing thread WT tends to update a key-value item from $(k1, v1)$ to $(k2, v2)$, and it has invoked a synchronization thread ST to process the write; in the meanwhile, a reading thread RT is issued to read key $k1$. In this case, it is possible that RT read an updated value $v2$ because ST has completed the write procedure in the *Cached Table*. However, at this moment, the bucket metadata (e.g., bitmap) may not have been persisted yet in the *Persistent Table*, which indicates that $(k2, v2)$ is not durable (or committed). Suppose a crash happens just before the updated metadata is persisted, then $v2$ will be lost and RT has just made an inconsistent read. In database community, this issue is called "read uncommitted", which violates the ACID principle of transaction.

To address this problem and provide full concurrent consistency guarantee, we modify the synchronization scheme to be as Figure 6(b) shows. There are two key changes. First, in the *Persistent Table*, we unlock the updated slot only after the bitmap field is persisted, which ensures that the update is durable before it is visible for concurrent readers. In this way, no "read uncommitted" issue will occur in the *Persistent Table*. Second, instead of releasing the lock of updated slot in the *Cached Table* by the synchronization thread, now we require the writing thread to complete that job. With this restriction, any concurrent read threads finding the target slot locked in the *Cached Table* will wait for the lock to be released, as lines 7 to 9 in algorithm 1 shows. Therefore, "read uncommitted" issue is avoided in the *Cached Table* as well.

We use the thread pool mechanism to avoid the context switch overhead of initiating and destroying a synchronization thread. Figure 7 illustrates an example for frontend-backend coordination where 8 writing threads are executing in the frontend and 4 synchronization threads are executing in the backend. As there is no need to guarantee persistence ordering in DRAM, the latency overhead of data synchronization is trivial. In addition, DRAM has much higher write bandwidth and write scalability than Optane. We measure that 4 synchronization threads can serve 4 to 64 writing threads with less than 2% performance compromise.

Notice that we employ the non-allocating write technique for *Cached Table* writes. That is, if Rewo-Hash cannot find a free slot for a key to insert, or fail to locate the original slot for a key to update in the *Cached Table*, it will not make modifications to the *Cached Table*. On such occasion, the *cache_overflow* field should be updated to TRUE to guarantee correct search for future requests. The reason why we choose this way is to minimize unnecessary writes to the *Cached Table*, which is fully responsible for search requests.

Although the shadowing synchronization mechanism is inspired by the frontend-backend coordination model proposed in HiKV [5], there are three key differences between Rewo-Hash and HiKV. First, the object and target of backend modification is different. HiKV uses backend threads to write to in-DRAM B+-Tree to support fast range query. However, we use backend threads to update in-DRAM hash table to reduce NVM read overhead. Second, the commit procedure is different. HiKV commits a write when it is persisted in NVM-resided hash table, while the B+-Tree is updated asynchronously. Our design, by contrast, commits a write only when the lightweight in-DRAM modification is also completed. Third, the concurrency mechanism is different. HiKV will block future writes to the in-NVM hash table when a range query is served in the DRAM-resided B+-Tree. By comparison, Rewo-Hash does not block any writes to the *Persistent Table* during updating the *Cached Table*.

3.6. Non-Blocking Lightweight Resizing

Resizing is a basic ability of hash table to extend or shrink its storage space when the load factor is too high or too low. There are two challenges for designing NVM-based table resizing scheme. First, resizing often incurs data migration for every item in the hash table, leading to high time cost and write bandwidth consumption. Second, resizing may seriously block concurrent write and even read requests to avoid data inconsistency, and hence leads to low request throughput during the resizing interval. Rewo-Hash's resizing mechanism optimizes the performance hit with two distinguished features. First, it is lightweight in data migration and metadata management. Second, in order to achieve high throughput, it is non-blocking for concurrent requests. In this paper, we introduce the resizing scheme with *extend* operation as the example. The procedure of *shrink* operation is exactly a mirror to the *extend*.

In most cases, Rewo-Hash only needs to migrate half of the existing keys during resizing, which is lightweight. The main idea is that we hold a directory field in the hash table metadata, which allows a logical *Persistent Table* to be composed by several scattered data tables. Each data table is a separate bucket-based cuckoo hash table and the bucket number of each data table can be different. The request key in Rewo-Hash is first matched onto a data table (i.e., owner table) based on its most significant bits (MSB) and then hashed to a bucket in the owner table.

Figure 8 shows an example for *Persistent Table* organization after three times of the resizing procedure. There are five directory entries in the hash table metadata and directory *T0* is a reserved entry for full data migration. Upon the first allocation of hash table space, Rewo-Hash only contains one data table which consists of 2 buckets. Both of directory entries *T0* and *T1* point to the data table. All the request keys are automatically matched onto the default data table. When the first resizing occurs, a new data table is allocated to extend the entire hash table to be 2x space size. Directory entry *T2* will point to the new data table. The bit set of *T1* and *T2* will be set as [0] and [1], respectively, indicating that all the keys with MSB being 0-bit should be matched onto *T1* while those starting with 1-bit should be matched onto *T2*. As a result, only those keys with 1-bit MSB in the old data table will be migrated to the new data table. The procedure for the second resizing and third resizing is similar: a new data table is allocated and a new directory entry is used to point to the data table. All the bit sets of used directory entries will be updated to rehash keys from the old data tables. Notice that the size of newly-allocated data table is equal to the size of current *Persistent Table*. When all the directory entries are used and hash table should be resized

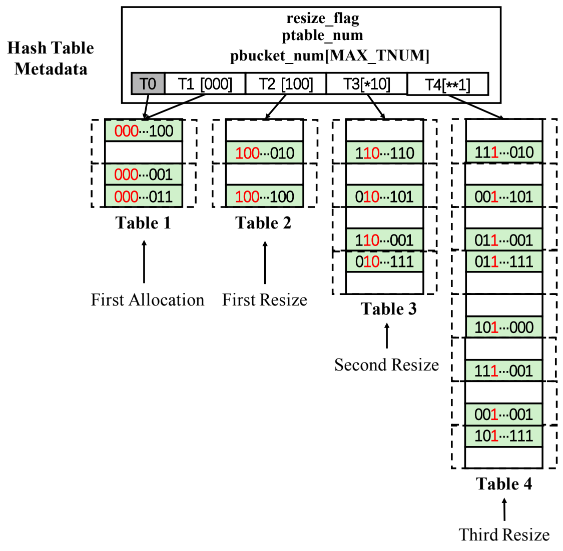


Figure 8: **Lightweight Resizing of Rewo-Hash.** Hash table metadata keeps a directory field, which contains fixed size of entries. Each directory entry contains a bit set for key match and a persistent address that points to a separate data table.

again, it is when *T0* comes into use. A new data table which is equal to 2x size of the current *Persistent Table* will be allocated and *T0* will point to it. Afterwards, all the keys in the old data tables should be migrated to the new data table. When the migration completes, the space of all the old data tables will be recycled and T1 will point to the new data table. In our current design, without counting T0, 16 directory entires are held in the metadata area. Suppose the first allocated space size of Rewo-Hash is 1MB, then 16 directory entries would allow a maximum of 512 GB data to be stored without triggering the condition for full key migration. Compared with CCEH [9] that partitions each data segment as the same size (a few KBs), Rewo-Hash keeps fewer directory entries in metadata and hence has lower metadata management overhead.

The migration of one given key includes two phases. First, Rewo-Hash executes an insert operation for current key-value item to the new data table. Second, Rewo-Hash invalidates the original slot by updating the corresponding *bit_flag* in *bitmap*. The interesting point is that the invalidation phase can be conducted in a bucket-batching method because the 8-byte *bmeta* field for one bucket can be updated atomically. We also add a *resize_count* field in metadata to record the proceeding of resizing, which will be useful for data recovery if a system crash happens during table resizing.

Our resizing mechanism also supports high-performance non-blocking concurrent reads and writes. The execution procedure for different operations during resize are introduced as follows.
Search during Resizing. If data is not found in the *Cached Table* and *cache_overflow* field is TRUE, Rewo-Hash will continue to search in the *Persistent Table*. There are two cases. (1) The request key is matched onto an old data table (as owner table). For this case, Rewo-Hash executes a normal search and returns the corresponding *value* field if the key exists. (2) The request key is matched onto the new data table. Then there are two

Algorithm 6: rewo_recover()

```
1  if meta.magic == MAGIC_DEFINED then
2      meta.magic = MAGIC_RUN;
3      ntstore(meta.magic);sfence();
4  if meta.resize_flag == TRUE then
5      redo_resize(rehash_count);
6  item_count = 0;
7  /* different data table processing can be paralleled with
   multi-threading */
8  foreach dtable in persistent_table do
9      foreach bucket in dtable do
10         foreach slot in bucket do
11             if slot.bit_flag = 1 then
12                 if slot.lock == 1 then
13                     obucket ←
                         check_double_and_clear(slot.key);
14                     if obucket does not exist then
15                         item_count++;
16                 else
17                     item_count++;
18         bucket.meta.lockmap = 0;
19         clwb(&bucket.meta.lockmap); sfence();
20 meta.item_count = item_count;
21 clwb(&meta.item_count); sfence();
22 meta.magic = MAGIC_RUN;
23 clwb(&meta.magic); sfence();
```

other situations. First, the key is found and the request can just return normally. Second, the key does not exist in the new data table. There is a possibility that the requested item is in an old data table but not migrated to the new table yet. As such, an additional lookup in the last-matched data table for the request key is needed.

Insert during Resizing. The writing thread will simply insert the key to the matched data table based on the current bit sets of all valid directory entries.

Update during Resizing. If the key is matched onto the table where the key resides in, the writing thread will execute as algorithm 4 describes. If it is matched onto the new data table and the original slot is in the old data table, then the update operation will be divided into an insert into the new data table and a delete in the old data table.

Delete during Resizing. The writing thread just deletes the key where it resides, either in the new data table or an old data table.

3.7. Crash Recovery

In Rewo-Hash, since each single write is an atomic operation, partial write inconsistency or persistence ordering inconsistency will never occur. However, there are still three cases that will lead to an inconsistent state.

Crash during metadata update. For example, *item_count*, which records the number of key-value pairs in the hash table, may not be updated or persisted after an insert operation. Such metadata can be recovered through counting the number of used slots in the *Persistent Table* (i.e., traversing the *bmeta* field of all the buckets).

Crash during resizing. When a crash happens, resizing may be half-done with only a portion of keys in the old data tables migrated to the newly-generated data tables. Fortunately, resize accords with the idempotence rule and replaying the resizing procedure does not affect the correctness. We utilize the *rehash_count* data structure in the metadata area to speed up resizing after crash recovery. Rewo-Hash can start the resizing procedure from a persistent checkpoint.

Crash during cuckoo eviction. There may exist double records when system crashes, which should be detected and resolved after recovery. If an item has a double record, then it indicates that its original version has not been invalidated yet and it must hold a lock in the *lockmap* field. Therefore, Rewo-Hash only checks if there is a double record for the slot which is valid (i.e., *bit_flag* is 1) and holds a lock (i.e., *lock* bit is 1). When the double record is found, one record will be deleted to maintain data consistency.

Algorithm 6 shows the procedure of crash recovery. For correct data recovery, Rewo-Hash will first check if the *resize_flag* is `TRUE`, and the resizing procedure will be replayed if needed. Afterwards, Rewo-Hash will traverse the data buckets to recover *item_count* while clearing double records and resetting the lock states. Such recovery procedure is implemented with multi-threading in Rewo-Hash. That is, each recovery thread is responsible for one data table and they can operate in parallel. The crash recovery overhead of Rewo-Hash, as we evaluate, is merely 0.67 second for 100 million key-value items (in one data table), which is acceptable for practical use.

After both persistent metadata and bucket data are recovered to a consistent state, Rewo-Hash can rebuild the *Cached Table* by inserting items from the *Persistent Table*. This procedure is not forced in Rewo-Hash because even if Rewo-Hash does not initially load any items to the *Cached Table*, items will still be gradually cached as they are processed.

3.8. Discussion

Support for Variable-Size Keys and Values. Since Rewo-Hash keeps each data slot in a bucket as fixed-sized for efficient management, it is optimized for small keys and values, which is also the target of most existing persistent hash table designs. However, besides constant-sized keys and values, Rewo-Hash can also support both different form of keys/values (i.e., integer type, string type) and variable-sized keys/values. For string-type keys and values, Rewo-Hash internally utilizes *strlen()* function to obtain the real length. The limitation is that a large-size key-value pair may not be able to be stored in a single data slot. For large-sized values, we can store the pointer to the real value object into the value field of Rewo-Hash's data slot. Such an approach is also widely used in many existing KV-Stores (e.g., Redis [50], Memcached [51]). A concern may be that a persistent pointer can be 16 bytes in structure size, which is larger than the available space size of Rewo-Hash's value field (15 bytes). For instance, PMDK's persistent pointer [52], namely *pmemoid*, is composed by an 8-byte *pool_id* and an 8-byte *offset*. However, even with 6-byte offset, it is still enough to search 0B to 256TB address space within one specified persistent pool, much larger than the possible deployed size[2] of NVM. Therefore, we can store a 8-byte *pool_id* and 6-byte *offset* in the value field of Rewo-Hash for large-size values. To enable the use of standard 16-byte *pmemoid* structure, a structure transformation should be conducted during search operations.

Configuration for Cached Table Size. In practice, precious DRAM resource may be applied for other purposes (e.g., long-time and large-scale in-memory computing) or applications (e.g., big data analytics and processing, scientific simulations, virtualization, graph processing and editing, etc). Therefore, it is natural to ask: what can be a proper (or minimum) DRAM Cache size for Rewo-Hash that can also maintain expected performance and latency? To be honest, the answer depends on the workload pattern. In Rewo-Hash, the DRAM Cache Table is used to speed up skewed read requests (read-heavy and high access skewness on the dataset). Table 2 presents the DRAM Cache requirement under different scenarios. Theoretically, the DRAM Cache size should be suitable when it can contain the hot work set (i.e., key-value items that are frequently read). For instance, for a YCSB-C workload with a skewness of 0.99, a DRAM Cache size of 35% Optane should be proper to respond to over 80% accesses; with a higher skewness of 1.5, the DRAM Cache size requirement can be as small as 8%. For a lower skewness such as 0.25, it might not be useful to use a DRAM Cached Table because the access pattern is nearly random and the caching/eviction procedure only adds extra performance overhead in the critical path. Hence, when the workload pattern changes, the proper ratio of the Cached Table also changes. A natural idea is to devise an adaptive technique that dynamically adjusts the size of DRAM Cached Table. That is, initially Rewo-Hash only applies for the minimum DRAM size to avoid occupying other high-priority applications' resources. Only for performance reasons, Rewo-Hash requests for more DRAM resource on demand. However, there are two significant challenges to design such an on-demand mechanism in Rewo-Hash: 1) how to recognize the change precisely and timely in workload pattern? 2) how to maintain a consistent access method without much rehashing overhead due to the Cached Table size update? Although there is a line of research [54, 55, 56] discussing the caching techniques for DRAM-NVM hybrid memory, they cannot be directly applied in Rewo-Hash because they are not optimized for fine-granularity of key-value access in hash table. We consider the research on adaptive caching for persistent hash table in DRAM-NVM hybrid memory both important and intriguing. However, as it diverges from the current focus of our work, we plan to explore it in future research endeavours.

[2]According to the released roadmap of Intel Optane DC PM [53], the maximum size is 6TB per socket.

Table 2: **DRAM Cache Requirement under Different Workloads.**

Workload / Skewness	High	Mild	Low
Read-Only	Small	Large	No
Read-Heavy	Small	Large	No
Write-Heavy	Small	Small	No
Write-Only	No	No	No

4. Threat to Validity

4.1. The Cease of Optane

With the arrival of Intel Optane Persistent Memory in 2019, research on new data management techniques for byte-addressable persistent memory has increased more significantly than ever before. However, in 2022, Intel announced that their Optane product line was discontinued due to the lower-than-expected market acceptance. This can impair the validity and significance of Rewo-Hash since it is specially optimized for the features of Opatne.

Intel's Optane, introduced in 2015 as 3D XPoint memory [3], represented a significant advancement in phase-change memory technology. Its dense Crosspoint structure allowed for stacking bits vertically, promising reduced cost per bit and superior efficiency through multiple layers. Moreover, its compatibility with tighter process geometries offered a potential for cost reduction surpassing DRAM and NAND flash.

However, transitioning Optane to DIMM format necessitated extensive modifications across processor architectures, operating systems, and application software. This process was further complicated by Intel's use of a proprietary DDR-T protocol for Optane persistent memory, which, being fitted into motherboard DIMM slots, hindered the development of a robust ecosystem and broader industry participation. Despite these efforts, Optane's high production costs could not be justified to buyers without a substantial price advantage over DRAM, leaving it appreciated by a few but adopted by too few, thus failing to achieve widespread acclaim.

For about four years, Intel endeavoured to boost wafer demand via DIMMs, but the adoption of this innovative approach lagged, leading to significant financial losses for the company. In retrospect, while Intel's execution of the Optane project was meticulous, it was a strategic misstep rooted in a misjudgment of the critical role economies of scale play in the memory sector.

4.2. CXL and Future NVM

The discontinuation of Intel's 3D XPoint-based Optane product line [43] left many researchers questioning the future of persistent memory technologies. However, beyond the immediate reactions, both Micron [57] and Intel [58] have aligned with the Compute Express Link (CXL) [59] standard, positioning it as the future of persistent and hierarchical memory. This shift represents a dual movement for persistent memory: a step back, with the loss of a key storage technology, and a cautious step forward, embracing an alternative and arguably more versatile interface. The CXL interface stands out not only for its

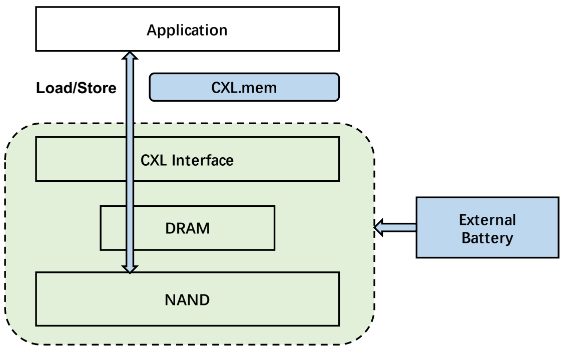


Figure 9: **Persistent Memory Mode of CMM-H.** Load/Store accesses to DRAM persistency via flushes to SSD.

vendor neutrality but also for its versatility, supporting a range of applications such as cache-coherent GPU-to-host access and CXL memory pooling. These applications promise to secure its relevance beyond the specific demands for persistent memory. Prior to CXL, the integration of persistent memory was predominantly a consideration for CPU manufacturers. Now, with CXL, even academic researchers have the opportunity to design and implement FPGA-based persistent memory prototypes [60].

Furthermore, the concept of CXL-based persistent memory has progressed from conception to reality. Samsung has announced a CXL-compatible product called CMM-H [61] that supports persistent memory mode. Figure 9[3] shows the architecture of CMM-H. From the user's perspective, the DRAM+SSD module is exposed as a non-volatile memory. Data persistence is achieved through the NAND Flash Backing Store, complemented by an internal DRAM buffer within the device. Under normal operations, CPU caches regularly flush data to DRAM. In the event of a host power failure, the CPU initially flushes cache data to DRAM. Subsequently, it issues a Global Persistent Flush (GPF) command to the device. This command triggers the transfer of data from DRAM to the NAND Flash Backing Store. Upon restoration of host power, data is seamlessly retrieved from the NAND Flash, repopulated into DRAM and then returned to the CPU caches as required. This architecture is quite similar to Optane, which also utilizes a DRAM buffer and persistent domain to guarantee performance and persistence.

Among other types of NVM, several companies have achieved varying degrees of financial success with their emerging memory technologies. Everspin and Renesas are known for their MRAM offerings, Adesto for ReRAMs, and Infineon boasts a thriving FRAM business [63]. These technologies are particularly preferred in scenarios that require a nonvolatile memory with either extremely low energy consumption or high resistance to radiation exposure. The latter is a significant concern not only in space applications but also for surgical instruments made from wafers. However, their market share is expected to remain relatively small in the near term, limiting the wafer volumes necessary for cost reduction.

[3]This figure is reshaped from the introduction in [62].

Table 3: **Comparison between Optane and CXL-based NVM [24, 64, 65]**

Feature / NVM Type	Optane	CXL-Based
Device Exposition	DAX FS	DAX FS
Access Interface	Load/Store	Load/Store
Access Granularity	256B block	4KB page
Latency	300ns-400ns	850-950ns
Bandwidth	6GB/s per DIMM	4GB/s per lane

4.3. Can Rewo-Hash adapt to CXL-based NVM?

Now the question is for emerging CXL-based NVM device, can Rewo-Hash still hold its performance merits although it is initially designed and optimized for Optane? Our answer is yes. The reason is that there are many common features between CXL-based NVM and Optane. Table 3 lists the main common features and differences between them.

Based on the comparison presented in Table 3, it is evident that Rewo-Hash can be seamlessly integrated into a CXL-based NVM platform without requiring any adjustments. This compatibility arises because both platforms share identical device exposition and access interfaces, operating with block-level granularity. However, the CXL-based NVM distinguishes itself with a larger storage capacity due to its SSD backing. Additionally, both platforms exhibit latencies for read and write operations that exceed those of DRAM, with CXL-based NVM potentially facing further latency increases due to PCIe limitations. When considering bandwidth per DIMM or per lane, they are notably inferior to that of DRAM.

Given these shared characteristics, Rewo-Hash's Cached Table could offer enhanced utility within a CXL-based NVM environment compared to its performance on an Optane platform. The log-free atomic write mechanism further adds to its applicability and effectiveness.

5. Evaluation

5.1. System Configuration

We conduct our experiments on both DRAM-simulated platform and a real commercial NVM platform (i.e., equipped with Intel Optance DC Persistent Memory). The configuration parameters of our experimental system are shown in Table 4. For the DRAM-simulated platform, we follow the guide illustrated in [66] and reserve a persistent region `/dev/pmem0`, which is 64 GB. For the Optane platform, we create a 256 GB persistent region `/dev/pmem1` using Intel's released tools *impctl* and *ndctl*. Similar to Dash [67], the low-level PM inconsistency issue (e.g., persistent memory leak, persistent pointer remapping) of Rewo-Hash is addressed by PMDK [52], which provides primitives for persistent memory allocation/recycling and crash-safe NVM management. By using a DRAM-simulated platform for comparison, we can verify three points of Rewo-Hash. First, since previous works simulate NVM with DRAM (adding write latency or not), we can get an overview of how Rewo-Hash behaves compared with other works based on the previously-assumed NVM features. Second, when compared

Table 4: Non-Volatile Memory Platform Configuration

Processor and Cache	
CPU	Intel Xeon(R) CPU@2.60GHz
Core Number	36
Private L1 Cache	32KB, 8-way, LRU
Private L2 Cache	1MB,16-way,LRU
Shared L3 Cache	24MB,11-way,LRU
DRAM Attributes	
Capacity	32 GB x 6
Persistent Region Namespace	/dev/pmem0
Persistent Memory Attributes	
Capacity	256 GB x 6
Mode	AppDirect
Persistent Region Namespace	/dev/pmem1

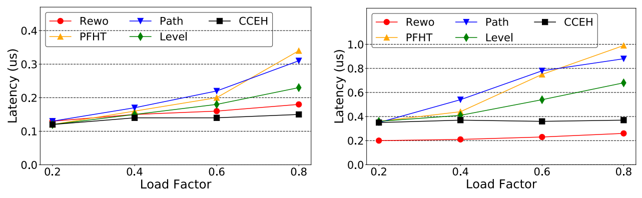
with the real NVM platform, we can tell that how much performance improvement is gained from Rewo-Hash over previous works with its Optane-optimized designs. Third, future NVM hardwares that achieve real DRAM-comparable latency and bandwidth may emerge, we can make sure if Rewo-Hash is still a good general fit when such hardwares are available. For DRAM-simulated platform, we disable the use of *Cached Table* by simply setting `DRAM_CACHE_ENABLE` as `FALSE` in the configuration file.

Since 16-byte key has been broadly adopted in existing key-value stores [68, 5], we limit the key size to 16 bytes and the value is set to be not longer than 15 bytes. With 256-byte bucket size, there are eight data slots and the remaining 8 bytes are used as bucket metadata (8 bits for *bitmap*, 8 bits for *lockmap* and 48 bits for *fingerprints*, with each fingerprint 6 bit in size). We observe that the maximum load factor of Rewo-Hash can reach 0.92, which is comparable to the state-of-the-art NVM-optimized hash tables [8, 67]. By default, we set the maximum capacity of in-DRAM *Cached Table* to be 1/8[4] of in-NVM *Persistent Table* (i.e., the maximum *Cached Table* size is 32 GB).

We compare Rewo-Hash with four representative persistent hash table designs, which include PFHT [23], Path-Hash [7], Level-Hash [8] and CCEH [9]. Since Path-Hash and CCEH do not implement update operation originally, we add necessary codes into their open-source implementations to support update operations. Both of them should rely on the logging techniques to guarantee consistent update. We adopt RandomInt[5] benchmark that is used in prior works [8, 7] to evaluate latency and YCSB [69] benchmark to evaluate throughput. For latency test, we conduct search and write operations for ten million times. For throughput test, we use three types of YCSB workloads: YCSB-C (read-only, 100% reads), YCSB-B (read-heavy, 95% reads and 5% updates) and YCSB-A (write-heavy, 50% reads and 50% updates). For both latency and throughput test, we adopt the zipfian-distributed workload access pattern, in which the skewness parameter is set to 0.99 by default. We run each experiment for five times and calculate the average result for presentation.

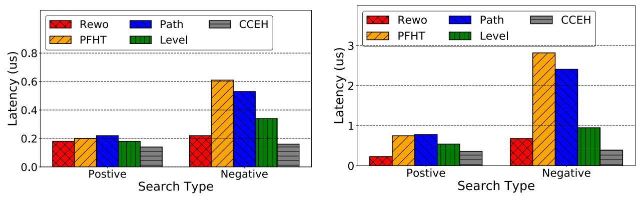
[4]This ratio is also the DRAM/NVM ratio on our Optane-based platform, which should be common in practical use.

[5]Both keys and values are randomly-generated integers.



(a) DRAM-simulated (b) Optane-based

Figure 10: Search Latency Under Different Load Factors.



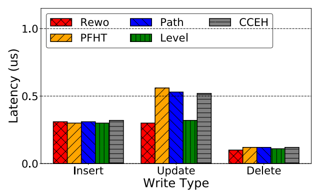
(a) DRAM-simulated (b) Optane-based

Figure 11: Latency for Different Search Types.

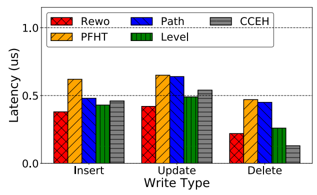
5.2. Search Latency

Figure 10 shows the comparison of search latency between Rewo-Hash and other designs under different load factors. The load factor indicates the occupancy of a hash table. On DRAM-simulated platform (shown in Figure 10(a)), the read latency of the five hash tables for low load factors (0.2 and 0.4) is similar. This is because almost all keys can be probed successfully in the primary hashed entry/bucket. When the load factor becomes higher (0.6 and 0.8), PFHT and Path-Hash has 1.33x-2.05x higher latency than Rewo-Hash. This is caused by their nature of multiple accesses to the storage space (i.e., probing the linear stash [23] or multi-level buckets [7]). CCEH performs the best among all designs since it probes at most one data bucket for any load factors [9]. On Optane-based platform (shown in Figure 10(b)), Rewo-Hash outperforms others by about 1.76x for load factor of 0.2 and 1.57x-3.39x for load factor of 0.8. Thanks to the *Cached Table* and CATI mechanism, Rewo-Hash can process each search request as much as possible in DRAM space. As DRAM is much faster than Optane in read latency, Rewo-Hash has much less overhead in data probing than others. On the contrary, under high load factor (i.e., 0.8), PFHT, Path-Hash and Level-Hash not only require multiple slow NVM block probing but also incur severe NVM access amplification overhead due to small bucket organization.

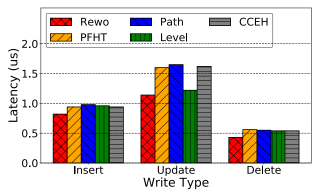
To further investigate the latency effect of multiple NVM accesses, we compare the latency of positive search (i.e., query keys are in the hash table) with negative search (i.e., query keys are not in the hash table) for the hash tables. The latency results are given in Figure 11. Even on DRAM-simulated platform (shown in Figure 11(a)), the latency of PFHT, Path-Hash is increased by 3.1x and 2.41x, respectively, when shifting from positive search to negative search. This implies that they suffer from more probing overhead due to scattered data storage. The negative search latency of Rewo-Hash and Level-Hash is 1.2x and 1.9x higher than the positive search case. CCEH is steady in read latency for negative search because it incurs no



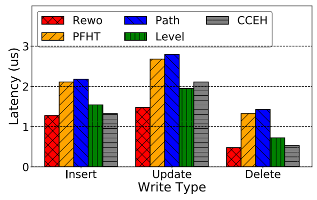
(a) Load Factor = 0.2, DRAM-simulated



(b) Load Factor = 0.8, DRAM-simulated



(c) Load Factor = 0.2, Optane-based



(d) Load Factor = 0.8, Optane-based

Figure 12: Comparison of Random Write Latency.

additional bucket access. On Optane-based platform (shown in Figure 11(b)), the effect of negative search is more remarkable for PFHT and Path-Hash. The read latency of them rises to 3.8x and 3.1x, respectively, when shifting positive search to negative search. Level-Hash incurs 1.8x latency for negative search while Rewo-Hash and CCEH almost keep steady search performance. In conclusion, Rewo-Hash outperforms PFHT, Path-Hash and Level-Hash by 1.4x-4.2x for negative search on Optane-based platform. The reason why Rewo-Hash can avoid high-latency even for negative search is attributed to the fingerprint technique we use for the *Persistent Table*, which reduces many unnecessary key comparisons for non-existing requested keys. In Rewo-Hash, there are four cachelines and two NVM blocks to be probed in total for negative search. As for Level-Hash and CCEH, the upper bound of search request overhead are four NVM block access and one NVM block access, respectively. Although CCEH needs at most one NVM bucket probing for each search request and hence has better negative search latency, it is not flexible for a key-value item to store in the same segment. Hence, it may cause frequent splits for insert collisions and lead to low occupancy for certain segments [9].

In addition to the average latency results for search operations, we also provide the comparison of 99.99 percentile tail latency (shown in Table 5), which is also a significant performance indicator for storage systems and applications [70]. We can observe that Rewo-Hash still outperforms other works in terms of 99.99% tail latency for search, and the performance improvement of Rewo-Hash over others has reached 1.3x-4.9x under high load factor (i.e., 0.8).

5.3. *Write Latency*

The write latency comparison on both DRAM-simulated platform and Optane-based platform is provided in Figure 12. We distinguish low load factor (0.2) from high load factor (0.8) and collect results for all three types of write: insert, update and delete. When the load factor is 0.2, all the five hash tables perform equally well for insert and delete operations on both DRAM-simulated platform (Figure 12(a)) and Optane-based platform (Figure 12(b)). This is because these two operations are log-free for all proposed designs and items are probably inserted to or deleted from the primary hashed entry. For update operation, Rewo-Hash and Level-Hash outperform the other three designs by about 1.5x to 1.6x on Optane-based platform. While PFHT, Path-Hash and CCEH require an undo log to guarantee consistency, Rewo-Hash and Level-Hash can update an existing item with append-only write and atomic metadata modification. Therefore, they incur less write overhead caused by logging or persistence ordering.

When the load factor becomes high (0.8), the latency improvement of Rewo-Hash over PFHT and Path-Hash is more remarkable. On DRAM-simulated platform (shown in Figure 12(c)), Rewo-Hash has similar write latency as Level-Hash and CCEH. All of them have constant probing time overhead for finding a free slot or an existing item. Path-Hash is slightly better than PFHT. On Optane-based platform (shown in Figure 12(d)), Rewo-Hash outperforms its counterparts by 1.04x-1.72x, 1.32x-1.77x and 1.10x-2.98x for insert, update and delete, respectively, showing the design advantages of LOFA mechanism. The high write overhead of PFHT and Path-Hash is mainly caused by multiple NVM block probing. Level-Hash generates 21% and 32% more overhead than Rewo-Hash for insert and update, respectively. We believe that it is caused by the logging overhead in the critical path under high load factor. A consistent result for all designs is that update is the most time-consuming operation while delete is the most time-efficient operation. The reason is that update may incur more NVM writes and *clwb+mfence* instructions while for deletes, only the probing overhead for target item grows when the load factor increases.

Table 6 provides the 99.99% tail latency comparison for different write operations. While the results show that Rewo-Hash consistently outperforms its counterparts in insert, update and delete, the superiority of Rewo-Hash's tail latency is not so remarkable as search operations. This is because unlike a search operation that benefits from the DRAM-resided *Cached Table*, Rewo-Hash must process and commit all writes in NVM space. Especially, we find that when the load factor is high (i.e., 0.8), the tail latency of Rewo-Hash's update is quite similar to Level-Hash. We infer that it is the slot eviction procedure of a few requests in Rewo-Hash that consumes the main portion of the processing time.

5.4. *Single-Thread Throughput*

In throughput test, for optane-based platform, we set the hash table size to be 256 GB. In this situation, the in-DRAM *Cached Table* can only cache a portion of all items that exist in the *Persistent Table*. For DRAM-based platform, we set the hash table size to be 64 GB. For both platforms, we keep inserting new items until the hash table load factor reaches 0.6.

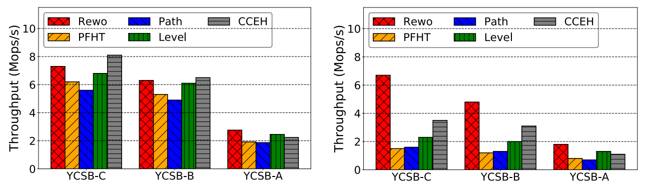
Figure 13 compares the single throughput of the five hash tables for three types of YCSB workloads. On DRAM-simulated platform (shown in Figure 13(a)), there is no obvious throughput gap among all designs. For read-only (YCSB-C) and read-heavy (YCSB-B) workload, CCEH performs the best since it

Table 5: Comparison of 99.99% Search Tail Latency (μs) on Optane-Based Platform

	Rewo-Hash	PFHT [23]	Path-Hash [7]	Level-Hash [8]	CCEH [9]
Load Factor = 0.2	0.25	0.68	0.59	0.51	0.38
Load Factor = 0.4	0.27	0.84	0.76	0.63	0.44
Load Factor = 0.6	0.30	1.25	1.22	0.74	0.46
Load Factor = 0.8	0.38	1.67	1.88	0.89	0.49
Negative Search	0.68	2.89	2.64	1.24	0.52

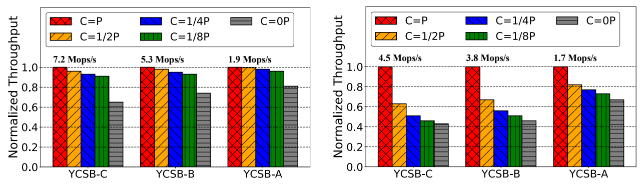
Table 6: Comparison of 99.99% Write Tail Latency (μs) on Optane-Based Platform

	Rewo-Hash	PFHT [23]	Path-Hash [7]	Level-Hash [8]	CCEH [9]
Insert (Load Factor = 0.2)	0.91	1.18	1.08	1.04	0.94
Insert (Load Factor = 0.8)	1.45	2.58	2.46	1.86	1.54
Update (Load Factor = 0.2)	1.28	1.73	1.85	1.34	1.70
Update (Load Factor = 0.8)	2.44	3.45	3.37	2.57	2.86
Delete (Load Factor = 0.2)	0.64	0.75	0.78	0.69	0.67
Delete (Load Factor = 0.8)	0.72	1.83	1.92	1.05	0.76



(a) DRAM-simulated (b) Optane-based

Figure 13: Single-Thread Throughput for YCSB Workloads

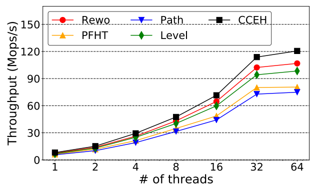


(a) Skewed Workload (b) Uniform Workload

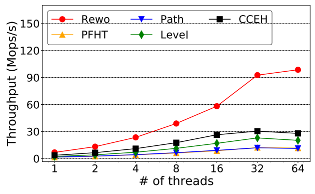
Figure 14: Performance Effect of Cached Table Size

has least probing overhead for search request. Rewo-Hash and Level-Hash have similar performance for these two workloads. When updates become heavy (YCSB-A), Rewo-Hash starts to outperform CCEH, which has extra logging overhead and higher persistence ordering overhead. On Optane-based platform (shown in Figure 13(b)), Rewo-Hash remarkably exceeds the other designs by 1.91x-4.46x, 1.54x-4.02x and 1.38x-2.57x for read-only, read-heavy and write-heavy workload, respectively. The higher performance of read-only and read-heavy workload is mainly gained by the *Cached Table* design and CATI read mechanism. On the other hand, the benefit of block-aligned block design of Rewo-Hash and LOFA write mechanism contribute to higher performance of write-heavy workload. In Rewo-Hash, most of the reads are served in the *Cached Table* directly and there is no need to probe the *Persistent Table*, due to the skewed access of YCSB workloads. The LOFA write mechanism ensures that no extra write overhead is introduced for write requests. We believe that the relatively lower throughput for write-heavy workload is mainly caused by the limited Optane buffer size and low write bandwidth to Optane DIMM.

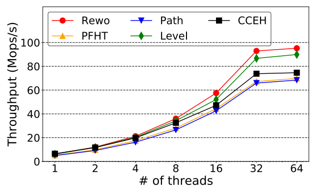
In practical use, DRAM size should be smaller than NVM size. For instance, on our Optane-based platform, the DRAM to NVM capacity ratio is 1/8. Therefore, we also need to observe the performance effect of the *Cached Table* size (i.e., available DRAM size) in Rewo-Hash. We use two types of workloads: skewed workload (default YCSB) and uniform workload (each item in the hash table can be accessed equally in probability). Figure 14 shows the throughput results. The size of *Cached Table* varies from full *Persistent Table* size (i.e., C = P) to 1/8 of *Persistent Table* size (i.e., C = 1/8P). The no-cache counterpart of Rewo-Hash (i.e., C = 0P) is regarded as a baseline. For skewed workload (shown in Figure 14(a)), only 4%-9% throughput drop is incurred with C = 1/8P, compared with the case of C = P. Compared with the baseline C = 0P, Rewo-Hash is 1.25x-1.54x better in throughput. Required items in skewed workload are mostly probed in the *Cached Table* due to data locality. Notice that data locality can also be exploited in LLC and CPU cache layer, which account for the high throughput of no-cache counterpart. For uniform workload (shown in Figure 14(b)), the throughput drops sharply. In particular, the throughput of YCSB-C is decreased by 37% with C = 1/2P and 54% with C = 1/8P, respectively. Compared with the baseline of no-cache counterpart, a small *Cached Table* is only 1.07x better. However, even for random-distributed workload, Rewo-Hash can still achieve the comparable throughput to existing state-of-the-art designs. For instance, CCEH obtains 2.7 Mops/s (1.3x Rewo-Hash), 2.1 Mops/s (1.1x Rewo-Hash) and 0.8 Mops/s (0.7x Rewo-Hash) for read-only, read-heavy, and write-heavy workloads, respectively. From this experiment, we can conclude two takeaways. First, for skewed workload, keeping a relatively small *Cached Table* in DRAM is sufficient to offer good performance. Second, for uniform workload (which is uncommon), keeping as large a *Cached Table* as the Persistent Table is desirable.



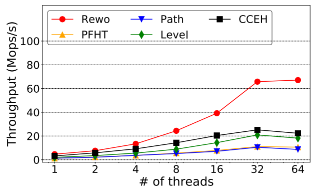
(a) YSCB-C, DRAM-simulated



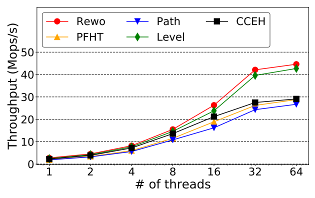
(b) YSCB-C, Optane-based



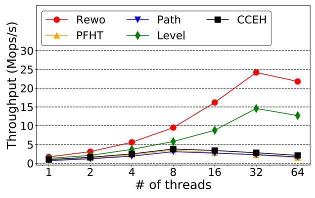
(c) YSCB-B, DRAM-simulated



(d) YSCB-B, Optane-based

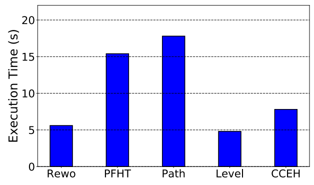


(e) YSCB-A, DRAM-simulated

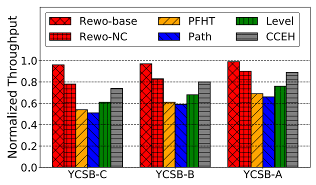


(f) YSCB-A, Optane-based

Figure 15: Comparison of Concurrent Throughput



(a) Time Overhead of Resize



(b) Throughput during Resize

Figure 16: Performance Effect of Resize.

5.5. *Concurrent Throughput*

Figure 15 shows the concurrent throughput of the compared hash tables. The number of (frontend) threads is set from 1 to 64 to evaluate the throughput scalability. We make two main observations to the results. First, on DRAM-simulated platform, the performance gap among the five NVM-optimized hash tables is small. For example, with 8 threads, Rewo-Hash obtains only up to 1.35x, 1.34x, 1.42x throughput for read-only, read-heavy and write-heavy workload compared with the others. With 64 threads, Rewo-Hash can only obtain 87.4% throughput of CCEH for read-only workload. This is because CCEH only needs to probe one bucket for each search request while Rewo-Hash requires to probe two buckets. As for read-heavy and write-heavy workload, Rewo-Hash gains the throughput similar to Level-Hash and outperforms the others by 1.24x-1.35x and 1.51x-1.64x. The reason is that both Rewo-Hash and Level-Hash enable fine-grained lock by atomically updating 8-byte bucket metadata for write request. Therefore, the throughput for concurrent reads/writes can be performed in non-blocking style for most requests. On the contrary, other designs lock the full bucket and only when the write operation is completed, will the lock be released and reads/writes to this bucket posted by other threads can be processed. The overall throughput is bottlenecked by the lock operation of write request. Fortunately, flush and fence in DRAM is efficient and the effect to DRAM is not quite severe.

Second, on Optane-based platform, however, Rewo-Hash remarkably scales better. Concretely, with 64 threads, the concurrent throughout of Rewo-Hash outperforms others by 3.46x-8.57x, 3.01x-7.56x, and 1.70x-13.7x for YCSB-C, YCSB-B and YCSB-A workload, respectively. For read-only workload and read-heavy, the high throughput of Rewo-Hash is mainly gained by CATI mechanism, which exploits the high read bandwidth of DRAM. Other designs suffer from (multiple) slow NVM reads and limited NVM read bandwidth. We observe that the throughput of CCEH, which consumes the least NVM read bandwidth, reaches the peak throughput of only 30 Mops/s with 32 threads. The trend for read-heavy workload on Optane-based platform is similar to that for read-only workload, but the peak throughput of each hash table has lower peak throughput due to the effect of write overhead. For write-heavy workload, the throughput of PFHT, Path-Hash and CCEH peak at 3.8 Mops/s with 8 threads. With more threads, the throughput drops dramatically. We infer that this is caused by their inefficient lock mechanism, which heavily block concurrent reads and writes to the locked bucket. Rewo-Hash and Level-Hash has better scalability for writes because of the design of lightweight atomic write. As such, the NVM write bandwidth consumption is minimized. Compared with Level-Hash, Rewo-Hash does not introduce any logging overhead in the critical path, and hence supports higher concurrency for search requests.

To sum up, it is important to consider the characteristics of the hardware in designing persistent hash tables. Existing hash tables have been designed based on assumptions that do not hold in real hardware. While they still offer good performance, their designs are not optimized. The design of Rewo-Hash can exploit the merits of Optane while avoiding its drawbacks to exert the best performance as a persistent hash table.

5.6. *Resizing Performance*

To evaluate the performance of resizing, we actively call resize function for each hash table, whose load factor reaches 0.8. To promise fairness, for each hash table, we strictly collect the time and throughput results for 2x hash table size extending. The results on Optane-based platform for a single thread are shown in Figure 16. From Figure 16(a), we have three observations. First, PFHT and Path-Hash have much higher time overhead than the other three. This is because they have to migrate each key-value item in the hash table for resizing. Second, Rewo-Hash has slightly lower time overhead than CCEH, although they are both based on dynamic zone allocation and directory matching. The reason is that Rewo-Hash efficiently reduces the space allocation overhead and metadata management overhead by allocating a new zone equal to current hash table size. An extend operation requires to allocate only one new persistent region/zone. However, CCEH allocates a fixed-sized segment each time and incurs more allocation overhead for 2x hash table size extending. Third, Level-Hash has the least time consumption among all hash tables. It takes advantages of the two-layer structure. Upon each resizing, only the top layer buckets, which occupy only 1/3 of total items, should be migrated to the new buckets.

Figure 16(b) shows the normalized throughput for each hash table during resizing. Here, the normalized throughput indicates the relative throughput compared with itself in the no-resizing case. Rewo-NC is a Rewo version without using the *Cached Table*. We can draw two conclusions. First, there is little performance hit for Rewo-Hash during resize, with 1%-4% throughput drop for three YCSB workloads. This is because most search requests are served directly in the *Cached Table* while update requests are not affected by the rehashing procedure. Second, without the *Cached Table*, the performance is decreased by 11%-22% but still better than other designs. There are two reasons: 1) at most four NVM blocks should be probed and the average probing block number is only 2.8 for search request during resizing; 2) write requests are processed in the matching data table without introducing additional cost.

6. Related Work

6.1. NVM-Optimized Data Structures

With the non-volatility, byte-addressability and large capacity NVM brings, building persistent data structures have been widely studied in the past decade, which mainly focused on hash tables [23, 7, 8, 9, 67] and trees [14, 47, 71, 11, 15].

In the area of designing persistent hash tables, the main aims are keeping high load factor, maintaining constant lookup time for random key search while avoiding the drawbacks of NVM (e.g., slow write and wear-leveling). PFHT [23], which is based on bucket cuckoo hashing [20], reduces NVM writes by allowing only one eviction for insert operation and it keeps a linear stash to store the items that cannot be inserted to the normal cuckoo hashing table to maintain high load factor. Path-Hash [7] supports insert and delete operations without any extra NVM writes by logically organizing the buckets in the hash table as an inverted complete binary tree while only the leaf nodes can be addressed by hash function. Each entry in the leaf-to-root path can be used for storing a key-value item if it is hashed to that leaf entry, and thus supporting high load factor. Level-Hash [8] simplifies the structure of Path-Hash to be only two layers: top layer and bottom layer, each of which is composed by buckets that allow multiple slots to be stored. The bottom layer is used as the hidden backup for the addressable top layer. With each key-value item having four alternative buckets to be stored, Level-Hash keeps both high load factor and constant search time. However, these hash tables are all based on previous assumptions on NVM, which is not consistent with the features of the commercial Intel Optane PM product [40, 72]. The performance drop is remarkable when deploying them on Optane-based platform, rather than a DRAM-simulated platform. Two newly NVM-based hash table proposals, CCEH [9] and Dash [67], utilize dynamic hashing scheme to support efficient resizing. However, these two designs still suffer from high read latency to NVM for search requests and may incur high metadata overhead (i.e., space allocation/recycling) caused by frequent resizing procedure. Different from previous work, Rewo-Hash is designed towards Optane, taking advantage of its merits while avoiding its drawbacks to achieve both high performance and high load factor.

For the work on persistent tree, B+-Tree is the mainstream. CDDS [73] is the first to propose using versioning technique to allow atomic updates without requiring logging for NVM-resided B-Tree. wB+ Tree [4] is a write-atomic persistent B+-Tree that employs a small indirect slot array to avoid the write and persistence overhead of index entries. clfBTree [74] is a B+-tree structure whose tree node fits in a single cache line assisted by differential encoding technique. BzTree [10] is a latch-free B+-Tree that utilizes a multi-word compare-and-swap operation (PMwCAS) as the core building block. NV-Tree [14] decouples the data structure of B+-Tree into in-NVM leaf nodes and in-DRAM internal nodes. It only guarantees consistency of only leaf nodes in B+-tree while relaxing that of internal nodes to reduce persistence overhead. FPTree [47], similar to NV-Tree, is a persistent B-Tree for hybrid DRAM-NVM main memory, with only the leaf nodes of B-tree persisted in NVM. It uses the fingerprinting technique to limit the in-leaf probing time overhead and enables hybrid concurrency with HTM. Hwang et al. [71] proposes the log-free failure-atomic shift (FAST) and in-place rebalance (FAIR) algorithms for B+-tree in persistent memory through sustaining transient inconsistency. LB+-Tree [49] is the latest persistent B+-Tree that optimizes NVM writes with three proposed techniques: entry moving, logless split and distributed header/metadata.

In addition to B-Tree, other tree structures such as radix tree [48, 75] and rbtree [11, 15] have also been studied to fit NVM. Unfortunately, most of these proposed trees are either heavily or partially dependent on previous assumptions of NVM, and we infer that they may probably lose the expected performance when using the Optane product. However, we believe that our work can also provides some optimization guidelines for them, such as block-aligned bucket, in-DRAM cached table and CATI mechanism, and log-free atomic write optimization that reduce NVM write bandwidth consumption.

6.2. NVM-Optimized Key-Value Stores

NVM technologies have also inspired a proliferation of key-value store researches [12, 5, 6, 76, 77, 78], in which hash table plays a key role. HiKV [5] develops a B-Tree index structure in DRAM to accelerate the performance of scan operations, but normal read requests are still served directly in NVM. Bullet [6] proposes a cross-referencing logging mechanism to resolve the concurrent write collisions to the same key-value item, but it is at the cost of more writes to NVM in the critical path. uDepot [76] and PapyrusKV [77] are developed for distributed high-performance computing (HPC) architectures that offer potentially massive pools of NVM and can achieve high performance through local NVM access.

6.3. DRAM As Cache in DRAM-NVM Hybrid Architecture

Since DRAM has lower latency and higher bandwidth than NVM, many researches have devoted to exploiting DRAM as the cache layer of NVM to improve the access performance. For instance, the hardware/software cooperative caching (HSCC) [55] mechanism organizes NVM and DRAM in a flat address space while logically supporting a cache/memory hierarchy, and it uses utility-based cache filtering policies to improve the efficiency of DRAM cache. UniBuffer [54] combines NVM with DRAM to reduce the journaling overhead. It puts infrequently updated data in NVM to reduce the journaling overhead and frequently updated data in DRAM to improve the write performance and lifetime of the hybrid buffer cache. Rainbow [79] manages NVM at the superpage granularity and uses DRAM to cache frequently accessed (hot) small pages within each superpage. Salkhordeh et. al [56] proposes an efficient data migration scheme at the operating system level in a hybrid DRAM-NVM memory architecture, where DRAM is used to cache hot data and NVM is used to cache cold data. Both DRAM and NVM have a least recently used (LRU) queue for the sake of data migration. However, all these works utilize NVM as a large-capacity memory device, but takes no advantage of its non-volatility. Therefore, their basis in using DRAM as a cache is totally different from Rewo-Hash. They do not address the persistence-related issues such as data durability guarantee and concurrency consistency guarantee.

7. Conclusion

To overcome the challenges that current NVM-optimized hash tables do not naturally fit for Intel Optane DC Persistent Memory (Optane), we propose a novel read-efficient and write-optimized hash table design for Optane, named Rewo-Hash. We keep a cached table in DRAM to speed up search requests and design a log-free atomic write mechanism to minimize the performance overhead for write requests. To mask the synchronization delay between persistent table and cached table, we devise an efficient shadowing synchronization scheme. We also propose a non-blocking lightweight resizing scheme to enable both time-efficient resize and high-performance request processing during data migration. The experimental results show that Rewo-Hash remarkably outperforms its counterparts.

References

[1] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, K. E. Goodson, Phase change memory, Proceedings of the IEEE 98 (12) (2010) 2201–2227.

[2] K. Wang, J. Alzate, P. K. Amiri, Low-power non-volatile spintronic memory: Stt-ram and beyond, Journal of Physics D: Applied Physics 46 (7) (2013) 074003.

[3] T. Morgan, Intel shows off 3d xpoint memory performance (2015).

[4] S. Chen, Q. Jin, Persistent b+-trees in non-volatile main memory, Proceedings of the VLDB Endowment 8 (7) (2015) 786–797.

[5] F. Xia, D. Jiang, J. Xiong, N. Sun, Hikv: a hybrid index key-value store for dram-nvm memory systems, in: 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17), 2017, pp. 349–362.

[6] Y. Huang, M. Pavlovic, V. Marathe, M. Seltzer, T. Harris, S. Byan, Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs, in: 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18), 2018, pp. 967–979.

[7] P. Zuo, Y. Hua, A write-friendly hashing scheme for non-volatile memory systems, in: Proc. MSST, 2017.

[8] P. Zuo, Y. Hua, J. Wu, Write-optimized and high-performance hashing index scheme for persistent memory, in: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), 2018, pp. 461–476.

[9] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, B. Nam, Write-optimized dynamic hashing for persistent memory, in: 17th {USENIX} Conference on File and Storage Technologies ({FAST} 19), 2019, pp. 31–44.

[10] J. Arulraj, J. Levandoski, U. F. Minhas, P.-A. Larson, Bztree: A high-performance latch-free range index for non-volatile memory, Proceedings of the VLDB Endowment 11 (5) (2018) 553–565.

[11] C. Wang, Q. Wei, L. Wu, S. Wang, C. Chen, X. Xiao, J. Yang, M. Xue, Y. Yang, Persisting rb-tree into nvm in a consistency perspective, ACM Transactions on Storage (TOS) 14 (1) (2018) 1–27.

[12] K. Huang, J. Zhou, L. Huang, Y. Shen, Nvht: An efficient key–value storage library for non-volatile memory, Journal of Parallel and Distributed Computing 120 (2018) 339–354.

[13] W.-H. Kim, J. Kim, W. Baek, B. Nam, Y. Won, Nvwal: exploiting nvram in write-ahead logging, in: ACM SIGPLAN Notices, Vol. 51, ACM, 2016, pp. 385–398.

[14] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, B. He, Nv-tree: reducing consistency cost for nvm-based single level systems, in: 13th {USENIX} Conference on File and Storage Technologies ({FAST} 15), 2015, pp. 167–181.

[15] M. Jeong, E. Lee, A swapping red-black tree for wear-leveling of non-volatile memory, The Journal of The Institute of Internet, Broadcasting and Communication 19 (6) (2019) 139–144.

[16] J. Xu, S. Swanson, {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories, in: 14th {USENIX} Conference on File and Storage Technologies ({FAST} 16), 2016, pp. 323–338.

[17] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, J. Jackson, System software for persistent memory, in: Proceedings of the Ninth European Conference on Computer Systems, 2014, pp. 1–15.

[18] S. Zheng, M. Hoseinzadeh, S. Swanson, Ziggurat: a tiered file system for non-volatile main memories and disks, in: 17th {USENIX} Conference on File and Storage Technologies ({FAST} 19), 2019, pp. 207–219.

[19] W. D. Maurer, T. G. Lewis, Hash table methods, ACM Computing Surveys (CSUR) 7 (1) (1975) 5–19.

[20] R. Pagh, F. F. Rodler, Cuckoo hashing, in: European Symposium on Algorithms, Springer, 2001, pp. 121–133.

[21] M. Herlihy, N. Shavit, M. Tzafrir, Hopscotch hashing, in: International Symposium on Distributed Computing, Springer, 2008, pp. 350–364.

[22] P. Flajolet, P. Poblete, A. Viola, On the analysis of linear probing hashing, Algorithmica 22 (4) (1998) 490–515.

[23] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, C. Ungureanu, Revisiting hash table design for phase change memory, ACM SIGOPS Operating Systems Review 49 (2) (2016) 18–26.

[24] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, S. Swanson, An empirical guide to the behavior and use of scalable persistent memory, in: 18th {USENIX} Conference on File and Storage Technologies ({FAST} 20), 2020, pp. 169–182.

[25] F. T. Hady, Intel optane technology delivers new levels of endurance (2023).

[26] K. Huang, Y. Yan, L. Huang, Revisiting persistent hash table design for commercial non-volatile memory, in: 2020 Design, Automation Test in Europe Conference Exhibition (DATE), 2020, pp. 708–713.

[27] D. Schwalb, G. K. BK, M. Dreseler, S. Anusha, M. Faust, A. Hohl, T. Berning, G. Makkar, H. Plattner, P. Deshmukh, Hyrise-nv: Instant recovery for in-memory databases using non-volatile memory, in: International Conference on Database Systems for Advanced Applications, Springer, 2016, pp. 267–282.

[28] A. Chatzistergiou, M. Cintra, S. D. Viglas, Rewind: Recovery write-ahead system for in-memory non-volatile data-structures, Proceedings of the VLDB Endowment 8 (5) (2015) 497–508.

[29] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, O. Mutiu, Thynvm: Enabling

software-transparent crash consistency in persistent memory systems, in: 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 2015, pp. 672–685.

[30] N. Cohen, M. Friedman, J. R. Larus, Efficient logging in non-volatile memory by exploiting coherency protocols, Proceedings of the ACM on Programming Languages 1 (OOPSLA) (2017) 1–24.

[31] Y. Lu, J. Shu, L. Sun, O. Mutlu, Loose-ordering consistency for persistent memory, in: 2014 IEEE 32nd International Conference on Computer Design (ICCD), IEEE, 2014, pp. 216–223.

[32] S. Pelley, P. M. Chen, T. F. Wenisch, Memory persistency, in: ACM SIGARCH Computer Architecture News, Vol. 42, IEEE Press, 2014, pp. 265–276.

[33] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, V. Chidambaram, Recipe: converting concurrent dram indexes to persistent-memory indexes, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019, pp. 462–477.

[34] J. Arulraj, M. Perron, A. Pavlo, Write-behind logging, Proceedings of the VLDB Endowment 10 (4) (2016) 337–348.

[35] H. Wan, Y. Lu, Y. Xu, J. Shu, Empirical study of redo and undo logging in persistent memory, in: 2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA), IEEE, 2016, pp. 1–6.

[36] K. Bhandari, D. R. Chakrabarti, H.-J. Boehm, Makalu: Fast recoverable allocation of non-volatile memory, in: ACM SIGPLAN Notices, Vol. 51, ACM, 2016, pp. 677–694.

[37] H. Volos, A. J. Tack, M. M. Swift, Mnemosyne: Lightweight persistent memory, in: ACM SIGARCH Computer Architecture News, Vol. 39, ACM, 2011, pp. 91–104.

[38] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, S. Swanson, Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories, ACM Sigplan Notices 46 (3) (2011) 105–118.

[39] Intel optane dc persistent memory: Big memory breakthrough for your biggest data challenges (2019).

[40] The challenge of keeping up with data (2019).

[41] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, S. Swanson, An empirical guide to the behavior and use of scalable persistent memory, arXiv preprint arXiv:1908.03583 (2019).

[42] B. Beeler, Intel optane dc persistent memory module (pmm) (2018).

[43] D. E. C. C. f. C. P. G. Gelsinger, P.; Zinsner, C. D. Zinsner, Intel reports first quarter 2022 financial results (2022).
URL `https://www.intel.com/content/www/us/en/newsroom/news/intel-reports-first-quarter-2022-financial-results.html`

[44] T. Szepesi, B. Wong, B. Cassell, T. Brecht, Designing a low-latency cuckoo hash table for write-intensive workloads using rdma, in: First International Workshop on Rack-scale Computing, 2014.

[45] C. Mitchell, Y. Geng, J. Li, Using one-sided {RDMA} reads to build a fast, cpu-efficient key-value store, in: Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13), 2013, pp. 103–114.

[46] E. Porat, B. Shalem, A cuckoo hashing variant with improved memory utilization and insertion time, in: 2012 Data Compression Conference, IEEE, 2012, pp. 347–356.

[47] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, W. Lehner, Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory, in: Proceedings of the 2016 International Conference on Management of Data, ACM, 2016, pp. 371–386.

[48] S. K. Lee, K. H. Lim, H. Song, B. Nam, S. H. Noh, {WORT}: Write optimal radix tree for persistent memory storage systems, in: 15th {USENIX} Conference on File and Storage Technologies ({FAST} 17), 2017, pp. 257–270.

[49] J. Liu, S. Chen, L. Wang, Lb+ trees: optimizing persistent index performance on 3dxpoint memory, Proceedings of the VLDB Endowment 13 (7) (2020) 1078–1090.

[50] J. Zawodny, Redis: Lightweight key/value store that goes the extra mile, Linux Magazine 79 (2009).

[51] S. Chu, Memcachedb: The complete guide (2008).

[52] Persistent Memory Programming, `http://pmem.io/` (2015).

[53] A. Ilkbahar, The future of intel optane persistent memory (2020).

[54] Z. Zhang, Z. Shen, Z. Jia, Z. Shao, Unibuffer: Optimizing journaling overhead with unified dram and nvm hybrid buffer cache, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (2019).

[55] H. Liu, Y. Chen, X. Liao, H. Jin, B. He, L. Zheng, R. Guo, Hardware/software cooperative caching for hybrid dram/nvm memory architectures, in: Proceedings of the International Conference on Supercomputing, 2017, pp. 1–10.

[56] R. Salkhordeh, H. Asadi, An operating system level data migration scheme in hybrid dram-nvm memory architecture, in: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2016, pp. 936–941.

[57] Micron updates data center portfolio strategy to address growing opportunity for memory and storage hierarchy innovation. (2021).

[58] Intel optane persistent memory and intel® xeon® scalable processors offer a practical migration path to memory expansion, tiering, and pooling with compute express link (cxltm)-attached memory devices (2022).

[59] C. E. Link, Compute express link (cxl) specification (2022).
URL `http://www.computeexpresslink.org`

[60] P. Desnoyers, I. Adams, T. Estro, A. Gandhi, G. Kuenning, M. Mesnier, C. Waldspurger, A. Wildani, E. Zadok, Persistent memory research in the post-optane era, in: Proceedings of the 1st Workshop on Disruptive Memory Systems, 2023, pp. 23–30.

[61] D. S. Pei, D. R. Pitchumani, Cmm-h (cxl memory module – hybrid): Rethinking storage for the memory-centric computing era (2023).

[62] Cmm-h (cxl memory module – hybrid): Samsung's cxl-based ssd for the memory-centric computing era (2023).
URL `https://semiconductor.samsung.com/us/news-events/tech-blog/webinar-memory-semantic-ssd/`

[63] J. Handy, T. Coughlin, Optane's dead: Now what?, Computer 56 (3) (2023) 125–130.

[64] Y. Fridman, S. Mutalik Desai, N. Singh, T. Willhalm, G. Oren, Cxl memory as persistent memory for disaggregated hpc: A practical approach, in: Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, 2023, pp. 983–994.

[65] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, P. Chauhan, Tpp: Transparent page placement for cxl-enabled tiered-memory, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, 2023, pp. 742–755.

[66] U. U. Thai Le, How to emulate persistent memory using dynamic random-access memory (dram) (2016).

[67] B. Lu, X. Hao, T. Wang, E. Lo, Dash: Scalable hashing on persistent memory, Proceedings of the VLDB Endowment 13 (8).

[68] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, M. Paleczny, Workload analysis of a large-scale key-value store, in: ACM SIGMETRICS Performance Evaluation Review, Vol. 40, ACM, 2012, pp. 53–64.

[69] M. Barata, J. Bernardino, P. Furtado, Ycsb and tpc-h: Big data and decision support benchmarks, in: Big Data (BigData Congress), 2014 IEEE International Congress on, IEEE, 2014, pp. 800–801.

[70] Z. Cao, S. Dong, S. Vemuri, D. H. Du, Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook, in: 18th {USENIX} Conference on File and Storage Technologies ({FAST} 20), 2020, pp. 209–223.

[71] D. Hwang, W.-H. Kim, Y. Won, B. Nam, Endurable transient inconsistency in byte-addressable persistent b+-tree, in: 16th {USENIX} Conference on File and Storage Technologies ({FAST} 18), 2018, pp. 187–200.

[72] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, et al., Basic performance measurements of the intel optane dc persistent memory module, arXiv preprint arXiv:1903.05714 (2019).

[73] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell, et al., Consistent and durable data structures for non-volatile byte-addressable memory., in: FAST, Vol. 11, 2011, pp. 61–75.

[74] W.-H. Kim, J. Seo, J. Kim, B. Nam, clfb-tree: Cacheline friendly persistent b-tree for nvram, ACM Transactions on Storage (TOS) 14 (1) (2018) 1–17.

[75] W. Pan, T. Xie, X. Song, Hart: A concurrent hash-assisted radix tree for dram-pm hybrid memory systems, in: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2019, pp. 921–931.

[76] K. Kourtis, N. Ioannou, I. Koltsidas, Reaping the performance of fast {NVM} storage with udepot, in: 17th {USENIX} Conference on File and

Storage Technologies ({FAST} 19), 2019, pp. 1–15.

[77] J. Kim, S. Lee, J. S. Vetter, Papyruskv: a high-performance parallel key-value store for distributed nvm architectures, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017, pp. 1–14.

[78] X. Chen, E. H.-M. Sha, A. Abdullah, Q. Zhuge, L. Wu, C. Yang, W. Jiang, Udorn: A design framework of persistent in-memory key-value database for nvm, in: 2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA), IEEE, 2017, pp. 1–6.

[79] X. Wang, H. Liu, X. Liao, J. Chen, H. Jin, Y. Zhang, L. Zheng, B. He, S. Jiang, Supporting superpages and lightweight page migration in hybrid memory systems, ACM Transactions on Architecture and Code Optimization (TACO) 16 (2) (2019) 1–26.