Revisiting Persistent Hash Table Design for Commercial Non-Volatile Memory

Kaixin Huang
Shanghai Jiao Tong University
Shanghai, China
kaixinhuang@sjtu.edu.cn

Yan Yan
Shanghai Jiao Tong University
Shanghai, China
118033910141@sjtu.edu.cn

Linpeng Huang*
Shanghai Jiao Tong University
Shanghai, China
lphuang@sjtu.edu.cn

***Abstract*—Emerging non-volatile memory technologies bring evolution to storage systems and durable data structures. Among them, a proliferation of researches on persistent hash table employ NVM as the storage layer for both fast access and efficient persistence. Most of them are based on the assumptions that NVM has cacheline access granularity, poor write endurance, DRAM-comparable read latency and much higher write latency. However, a commercial non-volatile memory product, named Intel Optane DC Persistent Memory (AEP), has a few interesting features that are different from previous assumptions, such as 1) block access granularity 2) hardware-layer wear-leveling and 3) much higher read latency than DRAM and DRAM-comparable write latency. Confronted with the new challenges brought by AEP, we propose Rewo-Hash, a novel read-efficient and write-optimized hash table for commercial non-volatile memory. Our design can be summarized into three key points. First, we keep a hash table copy in DRAM as a cached table to speed up search requests. Second, we design a log-free atomic mechanism to support fast writes. Third, we devise an efficient synchronization scheme between persistent table and cached table to mask the data synchronization overhead. We conduct extensive experiments using real NVM platform and the results show that compared with state-of-the-art NVM-Optimized hash tables, Rewo-Hash gains improvement of 1.73x-2.70x and 1.46x-3.11x in read latency and write latency, respectively. Rewo-Hash also outperforms its counterparts by 1.65x-4.24x in throughput for various YCSB workloads.**

***Index Terms*—hash table, non-volatile memory, Intel Optane DC Persistent Memory, data consistency**

I. 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. Academia assumes that NVM has non-volatility, cachline access granularity, low read latency, high write latency, and poor write endurance [4]–[6].

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

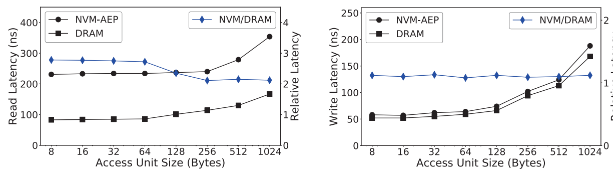
Concretely, PFHT is a write-friendly hash table for PCM that adopts bucket cuckoo hashing as its indexing scheme and only allows one-time eviction for write collision. In order to improve the capacity utilization, PFHT uses a stash (i.e., a linear array) to store the items failing to be inserted into the hash table [7]. Path-Hash 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 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 [8]. Level-Hash makes two major modifications based on Path-Hash. First, it contains only two layers for storing items: one addressable bottom layer and one hidden upper layer. Second, each bucket in Level-Hash is allowed to hold multiple items for space locality [9].

However, these works are heavily dependent on previous assumptions of NVM, which do not accord with the new characteristics of a commercial NVM product, Intel Optane DC Persistent Memory (AEP). As a recently released technique report [10] suggests, AEP has three highlighted features. First, AEP is accessed by block granularity (256 bytes). Second, AEP has relatively higher read latency (2x or 3x DRAM) but lower write latency (1x DRAM). Third, the write endurance may not be an urgent issue for software developers since hardware-layer wear-leveling is provided for AEP.

Existing designs on persistent hash table are no longer useful and in fact counterproductive, due to the lack of consideration for AEP's unique features. Therefore, we choose to revisit persistent hash table design for commercial NVM and propose Rewo-Hash. To bypass the high read latency of AEP, we keep a cached table in DRAM space to serve search requests. To reduce the fence/flush overhead and bandwidth consumption for write requests, we utilize the atomicity rule of CPU's 8-byte write to fulfill log-free writes to persistent table. Specifically, this paper makes the following contributions.

• We analyze the limitations of existing hash tables when using commercial non-volatile memory and propose a novel read-efficient and write-optimized persistent hash table, Rewo-Hash, to exert the full potential of AEP.

• We design a cached table-inclined read (CATI) mechanism to support fast search operations and a log-free atomic write (LOFA) mechanism to enable low-overhead write operations. We also propose an efficient synchronization scheme between



(a) Read Latency (b) Write Latency

Fig. 1. Access Latency of AEP

persistent table and cached table.

• We conduct extensive experiments to evaluate the performance of Rewo-Hash using real non-volatile memory platform. The results show that Rewo-Hash outperforms its counterparts by 1.73x-2.70x for search and 1.46x-3.11x for write in terms of latency. The speedup of throughput reaches 3.69x and 3.09x for read-heavy and write-heavy workload, respectively.

Organization. The remainder of this paper is organized as follows. Section II introduces the background and motivation of our work. Section III describes the design details of Rewo-Hash. Section IV presents experimental results. We review related work in Section V and conclude this paper in Section VI.

II. Background and Motivation

A. 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]–[3]. For NVM, data consistency is a significant issue, since partial writes and reordering writes to NVM may lead to inconsistent data storage across system crash. Most of existing works utilize *clflush/clwb+mfence* to guarantee persistence ordering [9], [11] and adopt logging techniques to support atomic update [5], [6] in NVM.

B. 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 (AEP). As a recently published technique report [10] describes, this new non-volatile memory technology, however, has subverted our knowledge about the characteristics of NVM. It is distinguished from previous assumptions in three aspects. First, despite of byte-addressability, AEP is accessed in block granularity (256 bytes), which leads to better performance for sequential reads than random reads. Second, AEP has relatively higher read latency (2x-3x DRAM) but lower write latency (1x DRAM). Third, AEP is equipped with a suit of hardware-layer wear-leveling techniques provided by Intel.

To verify the latency nature of our own AEP platform, we have conducted a series of experiments which randomly read from and write to AEP (the system configuration is given in Section IV) with different granularities. The results are illustrated in Figure 1, from which we have two key observations. First, the read latency of AEP is about 2.8x of DRAM when the access granularity is small (e.g., smaller than 64 bytes) while keeping steady with 2.1x when the access granularity is large (e.g., larger than 256 bytes). It implies that the large block access granularity (256 bytes) of AEP causes higher latency for random access of small-grained data. Second, the write latency of AEP is comparable to DRAM, with about merely 1.08x-1.13x DRAM latency among all different access unit sizes. The reason is that writes to NVM are durable when data enters the ADR domain, which guarantees to flush data into real NVM device during system crash [10]. The bottleneck of AEP, as Intel's releases numbers shows, is the limited write bandwidth [12].

TABLE I
AEP Feature Effects for Existing NVM-Optimized Hash Tables

Features	PFHT	Path-Hash	Level-Hash
block access granularity	×	×	×
high read latency	×	×	×
low write latency	✓	✓	✓
hardware-layer wear-leveling	×	×	-

C. Limitations of Existing NVM-Optimized Hash Tables

Conventional NVM-optimized hash tables, such as PFHT [7], Path-Hash [8] and Level-Hash [9], are 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 [8] or cache-aligned in a bucket [7], [9]. Second, they serve read requests directly in NVM because of the assumption that NVM read latency is comparable with DRAM. Third, they design wear-leveling mechanisms by reducing writes in the NVM bucket. PFHT and Path-Hash introduce extra lookup overhead due to the existence of large linear stash and multiple separate path buckets, respectively.

The new features brought by AEP have inspired us to rethink the bottlenecks of previous NVM-optimized hash table designs. We conclude the effects of the AEP features on existing works in Table I (✓represents positive effect; × indicates negative effect; – means neutral effect). A detailed explanation is provided as follows.

Block Access Granularity. Since existing works employ much smaller bucket size than AEP's access granularity, there will be read amplification problem for each bucket access.

High Read Latency. As all of three hash tables serve search requests directly in NVM, they are at the cost of at least 2x DRAM latency. Lower-than-expected performance will be observed especially for read-heavy workloads.

Low Write Latency. All the proposed hash tables can benefit from this feature because the write access will be faster than in the simulated environment. But note that the write bandwidth bottleneck should be taken into consideration.

Hardware-layer Wear-Leveling. This feature will indirectly decrease the performance of PFHT and Path-Hash

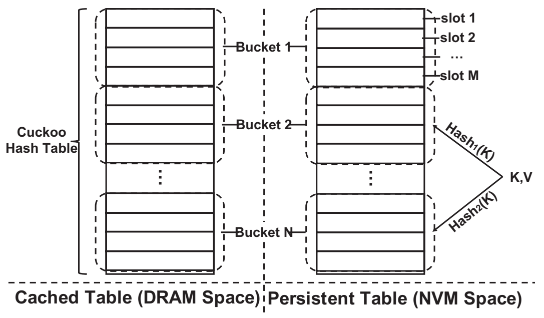


Fig. 2. Architecture Overview of Rewo-Hash.

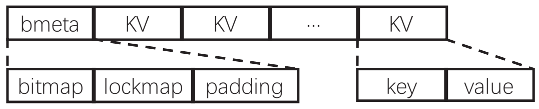


Fig. 3. Data Structure of Rewo-Hash Bucket.

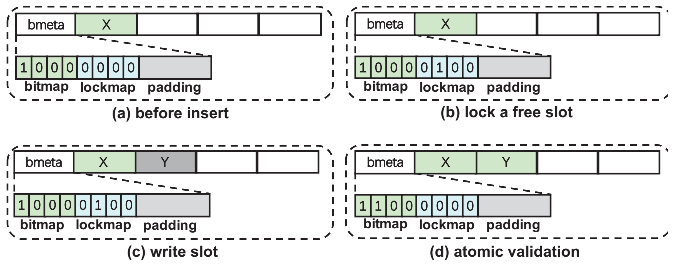


Fig. 4. **Procedure of Log-Free Atomic Insert.** This example shows the flow of inserting a new item Y in current bucket.

since they introduce extra lookup overhead for read requests due to their wear-leveling mechanisms, which in fact can be deprecated if developed on AEP.

All these limitations of existing NVM-optimized hash tables motivate us to propose a new design to fully exert the potential of AEP while avoiding its drawbacks.

III. Rewo-Hash

A. Architecture

Aiming at the characteristics of AEP, we have two main design considerations: 1) reduce the read latency by keeping the read path in DRAM space and 2) minimize data consistency overhead by designing a log-free write mechanism.

Figure 2 shows the design architecture of Rewo-Hash. We keep two tables in Rewo-Hash, with one in NVM space (i.e., persistent table) and the other in DRAM space (i.e., cached table). 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, cached table is totally used to serve search requests since it keeps a copy of data items in persistent table. If there is a size limitation for cached table due to strained DRAM space resource, then cached table will only store hot items with LRU technique, when the total item size is larger than its capacity.

Both of these two tables employ k-cuckoo hashing as the hashing method and we set k as 2, which is similar to Path-Hash [8] and Level-Hash [9]. 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.

B. Hash Table Bucket

The detailed data structure of each bucket is given in Figure 3. Except multiple slots to store key-value items, we also add a metadata zone for each bucket (i.e., *bmeta*), which contains *bitmap*, *lockmap* and *padding* bits. Notice that *bmeta* is an 8-byte data structure, and hence it can be modified atomically. *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 there is a pending write in corresponding slot.

Both persistent table and cached table adopt this bucket data structure. The only difference is the bucket size. To best utilize the space locality of AEP, the bucket size of persistent table is set as 256 bytes (block-aligned). While for cached table, the bucket size is set as 64 bytes (cacheline-aligned).

C. CATI: Cached Table-Inclined Read

Since cached table keeps the data copy of persistent table, all search requests are served first in DRAM space. There are two situations for search operations. If cached table is not overloaded, it can store all the inserted elements, and thus each value for a request key can be found in cached table. Otherwise, cached table only stores the hot dataset of persistent table. When a requested key can not be found in cached table, an additional persistent table lookup is required.

The search procedure in cached table is described as follows: (1) hash the requested key to a candidate bucket; (2) only check the valid slots whose corresponding bits in *bitmap* are 1s, and then find the matched key or restart from (1). (3) check the lock status of that slot in case of a pending update; (4) return if the slot is not locked or retry from step (3).

The search procedure in persistent table is less complicated. There is no need to check the lock status because in-place update will not occur. Once a matched key is found in step (2), the search request can return with corresponding value.

D. LOFA: Log-Free Atomic Write

In order to take advantage of the natue of low write latency while overcoming the obstacle of low write bandwidth of AEP, we design log-free atomic write mechanism to avoid the overhead caused by extra writes, flushes and fences.

Figure 4 illustrates the procedure of log-free atomic insert. Suppose that before inserting a new item Y, the bucket state is shown as Figure 4 (a), with one valid key-value item X. During the execution of inserting Y, Rewo-Hash first locks a free slot by flipping corresponding bit in *lockmap* from 1 to 0 to reject concurrent writes to this slot, as shown in Figure 4 (b). Then it writes the new item Y to the locked slot, which is shown in Figure 4 (c). Finally, Rewo-Hash validates the inserted item by performing an atomic operation, which contains two bit

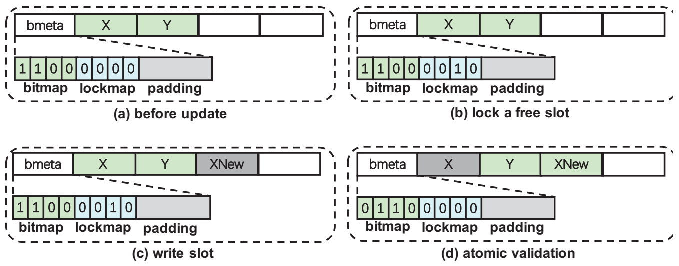


Fig. 5. **Procedure of Log-Free Atomic Update.** This example shows the flow of updating an existing item X to XNew in current bucket.

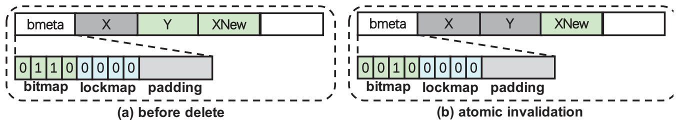


Fig. 6. **Procedure of Log-Free Atomic Delete.** This example shows the flow of deleting an existing item Y in current bucket.

flips in *bmeta*, one $0 \rightarrow 1$ flip in *bitmap* to make Y valid and one $1 \rightarrow 0$ flip in *lockmap* to unlock the concerned slot. The result after a successful insertion is given in Figure 4 (d).

Figure 5 shows the procedure of log-free atomic update, which is similar to the insert procedure. An updated item is considered as a new inserted item, except that in step (4), there are three bit flips in one atomic operation: one $0 \rightarrow 1$ flip in *bitmap* to make XNew valid, one $1 \rightarrow 0$ flip in *bitmap* to invalidate X, and one $1 \rightarrow 0$ flip in *lockmap* to unlock the used slot. The result of an update is presented in Figure 5 (d).

Figure 6 gives the procedure of log-free atomic delete. Delete operation is quite simple in Rewo-Hash since only an invalidation is required, with merely one bit flip in *bitmap*.

E. Data Synchronization

Since we need to guarantee the data in cached table consistent with that in persistent table, data synchronization between these two tables is required. A naive idea is to sequentially modify the data in persistent table and cached table. However, such a procedure will add nontrivial write delay.

We propose an efficient and lightweight synchronization scheme to minimize the write overhead to cached table. Figure 7 gives the work flow of such synchronization. The main idea is that we employ two types of threads during the write execution to mask the write latency of synchronization.

The writing thread is responsible for data modifications in persistent table while the synchronization thread is used to modify items in 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 `incomplete` and then delivers it to the synchronization thread. The synchronization thread executes the same write request in cached table and only when the execution succeeds, will the *sync signal* be updated to `complete`. After successfully executing the write operation in persistent table, the writing thread will check the status of *sync signal*. This

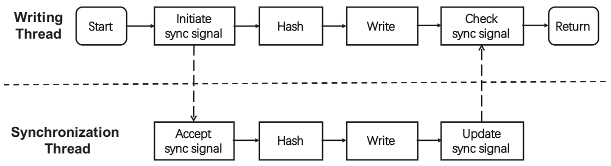


Fig. 7. **Procedure of Synchronization.** The writing thread writes data in persistent table while the synchronization thread writes data in cached table.

writing thread will not return until the *sync signal* is found to be `complete`. We use the thread pool mechanism to avoid the context switch overhead of initiating and destroying a synchronization thread.

F. Optimization

Unlike previous NVM-optimized hash tables that utilize logging techniques to purse both data consistency and wear-leveling [7]–[9], our proposed LOFA mechanism eliminates log writes by executing atomic metadata operation and released eviction limit. However, there still exists extra write overhead in situations when all the slots in a bucket are saturated. Active eviction may occur in the critical path for insert or update operations.

To mitigate such overhead, we make an optimization in Rewo-Hash, which is named background pre-eviction (BPE). Unlike the conventional cuckoo hashing technique that only evicts data to another bucket when needed. BPE allows the eviction process to occur earlier when a bucket is full of valid items. When one bucket is full-loaded, BPE will check from its last item and find its candidate bucket, which should be light-loaded, by using the other hash function. Afterwards, BPE will first lock a free slot in the candidate bucket to reject concurrent write, and then execute a log-free atomic insert operation in the candidate bucket, and finally make a log-free atomic delete operation in the original bucket. Such procedure is conducted by background threads which periodically iterate over the entire persistent table. Since the whole procedure is running in the background, the frontend search and write performance will be little affected. With BPE, Rewo-Hash is able to minimize the extra write overhead in the critical path.

IV. Evaluation

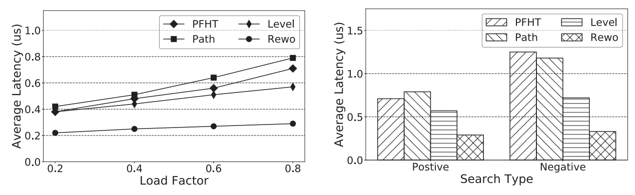
A. System Configuration

Different from previous works that use DRAM to emulate NVM [7]–[9], we conduct all of our experiments on a real commercial NVM platform, which is equipped with Intel Optance DC Persistent Memory. The configuration parameters of the system are shown in Table II. We create a 120 GB persistent memory device `/dev/pmem7` using Intel's released tools *impctl* and *ndctl*.

Since 16-byte key has been broadly adopted in existing key-value stores [5], [13], we limit the key size to 16 bytes and the value is set to be not longer than 15 bytes. Eight slots align a bucket in persistent table via padding several unused bytes. We set the total size of cached table equivalent to persistent

TABLE II
NON-VOLATILE MEMORY PLATFORM CONFIGURATION

Processor and Cache	
CPU	Intel Xeon(R) 6240M 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
Persistent Memory Attributes	
Capacity	256 GB x 6
Mode	AppDirect



(a) Latency for Various Load Factors (b) Latency for Two Search Types

Fig. 8. Comparison of Random Search Latency.

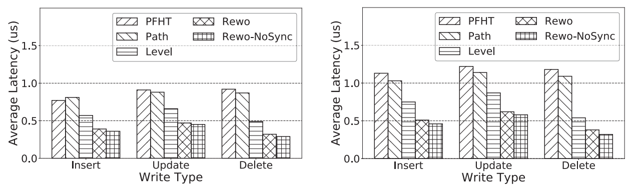
table to verify the performance gain produced by cached table. The BPE optimization is disabled by default.

We compare our proposed hash table with state-of-the-art designs such as PFHT [7], Path-Hash [8] and Level-Hash [9]. Each hash table is sized for a billion key-value items thus about 32GB of `/dev/pmem7` is used in total. We mainly adopt RandomInt benchmark that is used in [8], [9] to test latency and YCSB [14] benchmark to test throughput.

B. Search Latency

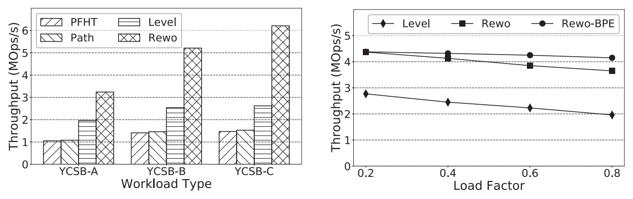
Figure 8 shows the comparison of search latency among Rewo-Hash and existing NVM-optimized hash tables. Load factor indicates the capacity utilization of a hash table. We observe from Figure 8(a) that Rewo-Hash has the lowest search latencies for all different load factors. It outperforms its counterparts by 1.73x-2.04x when the load factor is low (0.2 and 0.4). This is because Rewo-Hash can serve search requests with CATI mechanism, which directly search items in DRAM space. But other hash tables have to search in block-grained NVM, leading to higher access latency and read amplification. While for high load factors (0.6 and 0.8), the improvement of Rewo-Hash over others even reaches 1.97x-2.70x. Especially, both PFHT and Path-Hash have particular high latency in load-heavy situations. We believe this is caused by multiple NVM accesses when they search for given keys.

Figure 8(b) illustrates the latency of both positive search (i.e., query keys are in the hash table) and negative search (i.e., query keys are not in the hash table) for load factor 0.8. It further supports our suggestion by two aspects. First, the latencies of PFHT and Path-Hash are increased by 1.76x and 1.51x, respectively, when shifting from positive search to negative search. This implies that they suffer from more NVM accesses, due to full linear scan in stash structure (for PFHT) and full-path searching (for Path-Hash). Second, both of Level-Hash and Rewo-Hash keep more steady average latency. This is because the upper bound of access overhead for Rewo-Hash is only 2 DRAM cachelines. As for Level-Hash, it need to access at most 4 NVM blocks.



(a) Load Factor = 0.6 (b) Load Factor = 0.8

Fig. 9. Comparison of Random Write Latency.



(a) Throughput for YCSB Workloads (b) Effect of BPE for YCSB-A

Fig. 10. Comparison of Throughput.

C. Write Latency

The write latency comparison is provided in Figure 9. We can observe that Rewo-Hash outperforms others by 1.46x-2.88x when the load factor is 0.6, as given in Figure 9(a). This can attribute to the block-aligned organization for buckets in persistent table and efficient LOFA mechanism that eliminate log writes. We also notice that the synchronization overhead of Rewo-Hash is only 4.21%-8.33%, which indicates the synchronization scheme of Rewo-Hash is efficient.

When the load factor comes to 0.8, the latency improvement of Rewo-Hash over other three are about 1.47x-3.11x. The latency of Rewo-Hash itself is increased by 31%, 32% and 18% for insert, update and delete, respectively. The reason is that there are more active evictions in the critical path for inserts and updates when the load is heavier. While for deletes, only the search overhead for target key grows.

D. Throughput

Figure 10 compares the throughput of different hash tables for YCSB workload, which includes YCSB-A (write-heavy, 50% update + 50% search), YCSB-B (read-heavy, 5% update + 95% search) and YCSB-C (read-only, 100% search). It is observed that Rewo-Hash outperforms existing designs by 1.65x-3.09x, 2.05x-3.69x and 2.47x-4.24x for write-heavy, read-heavy and read-only workload, respectively, when the load factor is 0.8. The high throughput for read-skewed operations (6.21MOps/s and 5.21MOps/s) in Rewo-Hash, mainly benefits from the design of CATI read mechanism. As for the write-heavy workload, Rewo-Hash takes advantage of block-aligned bucket organization, LOFA write mechanism and efficient synchronization. We consider that the relatively lower throughput for write-heavy workload is mainly caused by active evictions and persistence ordering (*clwb+mfence*), since the write bandwidth of AEP is limited.

We also verify the throughput gain of Rewo-Hash from background-pre eviction. From figure 10(b), we can observe that Rewo-Hash with BPE produces more steady performance when the load factor changes. With BPE, the throughput of Rewo-Hash is improved by 5%, 10% and even 15% for load factor of 0.4, 0.6 and 0.8, respectively. The reason is that with background pre-eviction, load balancing will be executed in the backend and the number of overloaded buckets is reduced. Therefore, more key-value evictions will be removed from the critical path, which decreases the bandwidth squandering.

V. Related Work

A. NVM-Optimized Index Structures

Recent NVM-optimized index structure designs mainly focus on hash table [7]–[9] and B-Tree [15]–[17]. For hash tables, PFHT [7] reduces NVM writes by only allowing one eviction for insert operation and it keeps a linear stash to store the items failed to be inserted. Path-Hash [8] supports insertion and deletion 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 node can be addressed by hash function. Level-Hash [9] simplifies the structure of Path-Hash to be only two layers and the upper layer is used as the backup for the addressable bottom layer. For B-Tree related works, NV-Tree [15] guarantees consistency of only leaf nodes in B+-tree while relaxing that of internal nodes. FPTree [16] is a persistent B-Tree for hybrid DRAM-NVM main memory, in which only the leaf nodes of B+-tree are persisted in NVM while the internal nodes are stored in DRAM. Hwang et al. [17] propose the log-free failure-atomic shift (FAST) and in-place rebalance (FAIR) algorithms for B+-tree in persistent memory through sustaining transient inconsistency. Unfortunately, all these works are heavily dependent on previous assumptions of NVM, probably losing the expected performance when using the latest AEP product.

B. NVM-Optimized Key-Value Stores

NVM technologies have also inspired a proliferation of key-value store researches, where hash table data structure plays a key role. NVHT [18] introduces a wear-aware key-value store library using undo logging, which may incur high write latency and heavy bandwidth consumption for write-heavy workload. 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.

VI. Conclusion

To overcome the challenges that current NVM-optimized hash tables do not naturally fit for Intel Optane DC Persistent Memory (AEP), we propose a novel read-efficient and write-optimized hash table design for AEP, named Rewo-Hash. We keep a cached table in DRAM to speed up read requests and design log-free atomic write mechanism to reduce write overhead. We also devise an efficient and lightweight synchronization scheme between persistent table and cached table to mask the synchronization delay. The experimental results show that Rewo-Hash remarkably outperforms its counterparts.

Acknowledgments

This work is supported by National Key Research & Development Program of China (No. 2018YFB1003302), the China Scholarship Council (No. 201906230180) and the National Natural Science Foundation of China (No. 61472241). Linpeng Huang is the corresponding author of this paper.

References

[1] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.
[2] K. Wang, J. Alzate, and P. K. Amiri, "Low-power non-volatile spintronic memory: Stt-ram and beyond," *Journal of Physics D: Applied Physics*, vol. 46, no. 7, p. 074003, 2013.
[3] T. Morgan, "Intel shows off 3d xpoint memory performance," 2015.
[4] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.
[5] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: a hybrid index key-value store for dram-nvm memory systems," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 349–362.
[6] Y. Huang, M. Pavlovic, V. Marathe, M. Seltzer, T. Harris, and 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] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting hash table design for phase change memory," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, pp. 18–26, 2016.
[8] P. Zuo and Y. Hua, "A write-friendly hashing scheme for non-volatile memory systems," in *Proc. MSST*, 2017.
[9] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 461–476.
[10] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," *arXiv preprint arXiv:1908.03583*, 2019.
[11] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 265–276.
[12] "The challenge of keeping up with data," 2019. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf
[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.
[14] M. Barata, J. Bernardino, and 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.
[15] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and 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.
[16] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 371–386.
[17] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, 2018, pp. 187–200.
[18] K. Huang, J. Zhou, L. Huang, and Y. Shen, "Nvht: An efficient key–value storage library for non-volatile memory," *Journal of Parallel and Distributed Computing*, vol. 120, pp. 339–354, 2018.