REH: Redesigning Extendible Hashing for Commercial Non-Volatile Memory

Zhengtao Li, Zhipeng Tan*, Jianxi Chen
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, China
{lizhengtao10, tanzhipeng, chenjx}@hust.edu.cn *Corresponding author

***Abstract*—Emerging Non-volatile Memory (NVM) is attractive because of its byte-addressability, durability, and DRAM-scale latency. Hashing indexes have been extensively used to provide fast query services in the storage system. Recent research proposes crash-consistent and write-optimized hashing indexes for NVM. However, existing NVM-based hashing indexes suffer from limited scalability when running on a Commercial Non-Volatile Memory product, named Intel Optane DC Persistent Memory Module (DCPMM), due to the limited bandwidth of Optane DCPMM. To achieve a high load factor, existing NVM-based hashing indexes often evict an existing item to its alternative position, which incurs extra write and will consume the limited bandwidth. Moreover, the lock operations and metadata updates further saturate the limited bandwidth and prevent the hash table from scaling. In order to achieve scalability performance as well as a high load factor for the NVM-based hashing index, we design a new persistent hashing index, called REH, based on extendible hashing. REH (1) proposes a selective persistence scheme that stores buckets in NVM and places directory and metadata in DRAM to reduce both unnecessary NVM reads and writes, (2) uses 256B sized-buckets, as 256B is the internal data access size in Optane DCPMM, and the buckets are directly pointed to by directory entries, (3) leverages fingerprinting to further reduce unnecessary NVM reads, (4) employs failure-atomic bucket split to reduce bucket split overhead. Evaluations show that REH outperforms the state-of-the-art NVM-based hashing indexes by up to 1.68∼7.78×. In the meantime, REH can achieve a high load factor.**

I. INTRODUCTION

Non-volatile memory (NVM) is an emerging class of memory technology. Intel Optane DC Persistent Memory Module is the first commercially available NVM product. As Optane DCPMM is the only available NVM product, we use NVM and Optane DCPMM interchangeably in the rest of this paper. NVM will be directly attached to the processor's memory bus and accessed like DRAM via load/store instructions [6]. NVM has a unique performance profile: compared with DRAM, loads have 3× higher latency and $1/3^{rd}$ bandwidth, while stores have similar latency but $1/6^{th}$ bandwidth [2], [6], [11]. A single machine can be equipped with up to 6 TB of NVM. The low latency, durability, and large capacity of NVM make it an attractive medium for building storage systems.

Hashing indexes are key to achieving fast query service and are thus a crucial component of several storage systems. Researchers have designed several NVM-based hashing indexes such as Level hashing [14], CCEH [9], and DASH [8]. However, existing NVM-based hashing indexes suffer from limited scalability when running on Optane DC Persistent Memory. This is because they overlook the scarcity of NVM bandwidth and often incur extra NVM reads and writes.

Level hashing [14] is a two-level write-optimized NVM-based hashing index. To achieve a high load factor (the number of stored items in the hash table divided by the total capacity), Level hashing often evicts an existing item to its alternative bucket. However, eviction incurs extra NVM reads and writes. Besides, search operations need to access two buckets in both its top and bottle level, which incur extra NVM reads. CCEH [9] and Dash [8] are both based on extendible hashing, and they use an intermediate layer (referred to as segment) between the directory and buckets to reduce the directory size. This design results in a low load factor. CCEH uses linear probing to improve load factor, which means CCEH needs to access four cache lines to find an empty slot for insertion, incurring more NVM reads. Dash uses techniques such as balanced insert, displacement, and stashing to improve load factor, which incur extra NVM reads and writes. Moreover, the lock operation and metadata updates for insertion further saturate the limited bandwidth. Thus, existing NVM-based hashing indexes cannot achieve scalable performance when running on Optane DCPMM.

In order to achieve scalability performance as well as a high load factor for the NVM-based hashing index, we design a new persistent hashing index, called REH, based on extendible hashing. REH (1) proposes a selective persistence scheme that stores buckets in NVM and places directory and metadata in DRAM to reduce both unnecessary NVM reads and writes, (2) uses 256B sized-buckets, as 256B is the internal data access size in Optane DCPMM, and the buckets are directly pointed to by directory entries, (3) leverages fingerprinting to further reduce unnecessary NVM reads, (4) employs failure-atomic bucket split to reduce bucket split overhead. Evaluations show that REH outperforms the state-of-the-art NVM-based hashing indexes by up to 1.68∼7.78×. In the meantime, REH can achieve a high load factor.

The rest of this paper is organized as follows. Section II introduces the background. Section III describes the design of REH. Experimental results are presented in Section IV. Section V reviews related work, and we conclude the paper in Section VI.

II. BACKGROUND

A. Intel Optane DC Persistent Memory

Intel Optane DC persistent memory module (Optane DCPMM) based on 3DXPoint technology is the first commercially available product of NVM. NVM exhibits higher

978-3-9819263-6-1/DATE22/©2022 EDAA

latency than DRAM and has limited bandwidth. Compared with DRAM, loads have 3× higher latency and $1/3^{rd}$ bandwidth, while stores have similar latency but $1/6^{th}$ bandwidth [2], [11]. As the access granularity of 3DXPoint media in Optane DCPMM is 256 bytes, the XPController translates smaller requests into larger 256-byte accesses, incurring write amplification because small stores become read-modify-write operations. The XPController has a small write-combining buffer to merge adjacent writes [11]. As a result, the performance of small random stores is severely limited and non-scalable, which, however, is the inherent access pattern in hash tables [8]. Thus, it is critical to reducing both NVM reads and writes for higher performance.

It is worth noting that although NVM provides persistence, the multiple levels of CPU caches are still volatile. Data is not guaranteed to be persisted in NVM until a cache line flush instruction is executed or other events that implicitly cause cache line flush to occur [5], [8]. The memory controller may reorder writes to NVM, while memory fence instruction can avoid undesirable reordering. To ensure that persistent hashing indexes can recover from system crashes correctly, we need to leverage cache line flush instruction and memory fence instruction. However, persistent operations (e.g., clwb and sfence) have a high overhead and can drastically slow down write performance [7]. Thus, we should avoid unnecessary persistent operations. The failure atomicity unit of NVM is 8 bytes. If the size of the updated data exceeds 8 bytes and a system failure occurs before completing the update, the data may be inconsistent.

B. Extendible Hashing

Extendible hashing [4] can grow and shrink on demand at run-time without time-consuming full-table rehashing and has been used in numerous real systems. Figure 1 shows the structure of extendible hashing.

Extendible hashing includes many buckets and there is a directory to index these buckets. For the 64-bit hash key, K bits are used by the directory to locate the bucket. As shown in figure 1, we employ the two most significant bits (MSB) to select a bucket. The number of prefix bits currently used by the directory is called global depth (GD) ($GD \leq K$). Each bucket is associated with a local depth (LD) ($LD \leq GD$) that indicates the number of prefix bits used by the bucket [15]. If a bucket is pointed to by n directory entries, the local depth of the bucket is $LD = GD - log_2 n$. For example, bucket B is pointed to by only one directory entry, and GD is 2, thus, LD of bucket B is 2.

When a hash collision happens in a bucket, extendible hashing compares its local depth with the global depth. If the local depth is smaller than the global depth, indicating multiple directory entries are pointing to the bucket, it can split the bucket without doubling the directory. For example, bucket A can be split without doubling the directory, and then we can update directory entry 01_2 to point to the new split bucket. Finally, the local depths of bucket A and the new bucket are increased to 2. If the local depth is equal to the global depth, which means the bucket is pointed to by only one directory entry, it needs to double the directory before splitting the bucket (e.g., bucket B and bucket C).

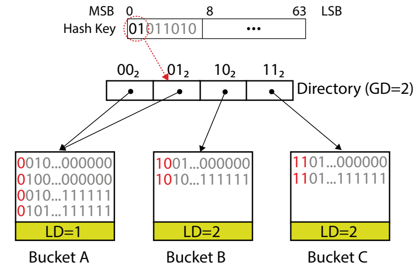


Fig. 1. Extendible hashing structure.

III. The Design of REH

A. Design Principles

- **Avoid Unnecessary NVM Reads.** Due to the high read latency and limited read bandwidth of NVM, it is imperative to reduce unnecessary NVM reads.
- **Avoid Unnecessary NVM Writes.** Although NVM exhibits similar write latency with DRAM, the write bandwidth of NVM is only $1/6^{th}$ that of DRAM. Excessive NVM writes can consume the limited bandwidth and prevent the hash table from scaling. Thus it is also imperative to reduce unnecessary NVM writes.
- **High Load Factor.** Indexes could consume more than 55% of the total memory in main-memory database management systems [12], so it is crucial to achieving a high load factor.

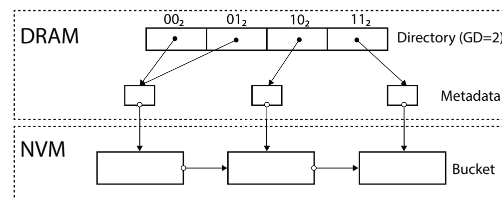


Fig. 2. Selective persistence applied to an extendible hashing: directory and metadata are kept in DRAM while buckets are kept in NVM.

B. Selective Persistence

Inspired by FPTree [10], we propose a selective persistence scheme for extendible hashing. Selective persistence means that the minimal set of primary data are kept in NVM and consistency will be enforced, while non-primary data are placed in DRAM and can be rebuilt from primary data during failure recovery. Applied to an extendible hashing, as illustrated in Figure 2, buckets are placed in NVM and chained using sibling pointers while directory and metadata are placed in DRAM and can be rebuilt as long as the buckets are in a consistent state. Thereby reducing both unnecessary NVM reads and writes.

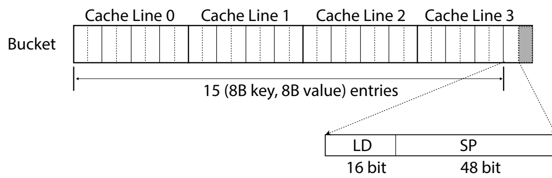


Fig. 3. 256-byte bucket.

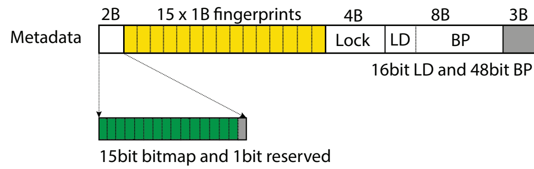


Fig. 4. Metadata of the bucket.

C. 256-Byte Bucket and its Corresponding Metadata

CCEH [9] and Dash [8] are both based on extendible hashing, and they introduce an intermediate layer, called segments, between the directory and buckets to reduce directory size. However, this design results in low load factors because a segment will be split once any bucket in it is full, even if the other buckets in the segment still have free space. To improve the load factor, CCEH and Dash often incur extra NVM reads and writes, which is undesirable for Optane DCPMM because it has limited bandwidth. Unlike CCEH and Dash, REH does not use the intermediate layer, and each directory entry directly points to a bucket. This design can achieve a high load factor because a bucket can be split only if it is full.

As 256 bytes is the internal data access size in Optane DCPMM, we set the bucket size to 256 bytes (bucket size smaller than 256 bytes will induce write amplification when bucket split, while the bucket size larger than 256 bytes will make the search performance degrade). All the buckets are aligned at 256-byte boundaries. Figure 3 shows the layout of a 256-byte bucket. A key-value pair (8-byte key, 8-byte value) takes 16 bytes. There are four 16-byte units in every cache line and 16 units in total in the entire bucket. We use 15 of the 16 units to store key-value pairs. The last unit in cache line 3 contains 16 bytes, and the first 8 bytes is employed to store local depth (LD) and sibling pointer (SP), and the remaining 8 bytes is reserved. Sibling pointer consumes 48 bits like the x86_64 systems [1], [15], and local depth takes up the remaining 16 bits. 16 bits should be enough to store local depth because the maximum local depth is 64. The choice of using 8 bytes to store local depth and sibling pointer is critical, as it allows us to update local depth and sibling pointer in a failure-atomic 8-byte write (details in subsection E).

Each bucket has its own metadata and the size of the metadata is 32 bytes. Figure 4 shows the layout of the metadata. Its starts with a 15-bit bitmap, and the following 1 bit is reserved. The bitmap is used to quickly identify whether the corresponding key-value pair in the bucket is valid. In fact, we can employ the local depth to verify the validity of the key-value pair, but it is much slower. The 15 1-byte fingerprints is used to accelerate search computation inside the bucket (details in subsection D). The 4-byte lock is for concurrency control (details in subsection F). The following 8 bytes contains local depth (LD) and bucket pointer (BP), and the bucket pointer points to its corresponding bucket. The last 3 bytes is reserved.

One might want to store the metadata inside the bucket. Suppose the 32-byte metadata occupies the first 32-byte of cache line 0. When we insert a key-value pair into cache line 2, we need to use special instructions (clwb and sfence) to persist the key-value pair from the CPU cache to NVM. After that, we need to update the metadata. As the metadata can be reconstructed from key-value pairs in the bucket. We can choose not to persist the metadata from CPU cache to NVM, and let the cache naturally evict cache line 0 to NVM. However, letting the cache naturally evict cache line 0 adds nondeterminism [11]. That is, cache line 2 is written to the 3DXpoint media, incurring 256-bytes write as 256 bytes is the internal data access size in Optane DCPMM. While cache line 0 is still in the CPU cache. After that, cache line 0 is evicted from CPU cache and reaches 3DXpoint media, incurring another 256-byte write. We call this phenomenon double 256-byte writes. Another option is to persist cache 0 from CPU cache to NVM after having persisted cache line 2 and let the XPController merging adjacent writes [11], thus we only need to write 256 bytes to the 3DXpoint media. However, persistent operations (clwb and sfence) have a high overhead and can drastically slow down write performance [7]. Thus, we choose to place the metadata of the bucket in DRAM to avoid double 256-byte writes and the overhead of persistent operations.

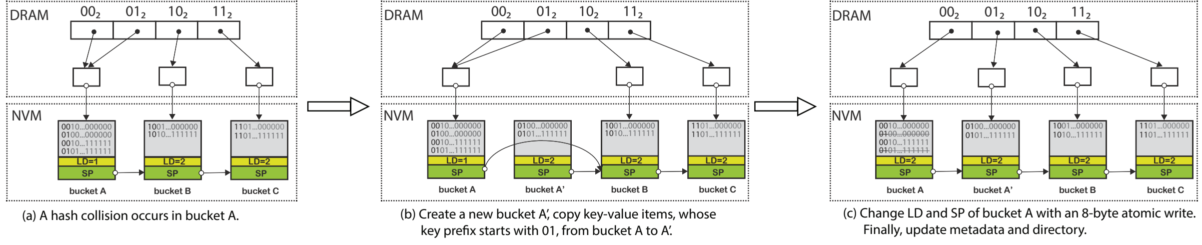
D. Fingerprinting

Fingerprinting was used by FPTree [10] and Dash [8] to improve probing (i.e., search in the leaf node or the bucket) performance. However, both FPTree and Dash place the fingerprints inside the leaf node or the bucket of a segment, which means search operations and insert operations need to read and update the fingerprints, incurring extra NVM reads and writes. Our design of REH moves a further step and places the fingerprints in DRAM to reduce unnecessary NVM reads and writes. As shown in figure 4, the 15 1-byte fingerprints correspond to the 15 key-value pairs in the bucket. Thus, a single 128-bit SIMD instruction can compare the fingerprint of a search key with the 15 fingerprints to quickly locate the key-value pair [7], greatly improve the probing performance.

E. Failure-Atomic Bucket Split

To ensure data consistency and to reduce NVM write during the bucket split, we propose a failure-atomic bucket split design for REH, as illustrated in figure 5. The basic idea is to update the local depth and the sibling pointer in a failure-atomic 8-byte write.

we will elaborate on the process of the failure-atomic bucket split using the example shown in figure 5. when a hash collision occurs in bucket A, we create a new bucket A' and copy key-value pairs, whose key prefix starts with 01, from bucket A



(a) A hash collision occurs in bucket A.

(b) Create a new bucket A', copy key-value items, whose key prefix starts with 01, from bucket A to A'.

(c) Change LD and SP of bucket A with an 8-byte atomic write. Finally, update metadata and directory.

Fig. 5. failure-atomic bucket split.

to bucket A'. To reduce NVM writes, we do not delete those migrated key-value pairs in bucket A, because they will become invalid with the update of local depth (LD) of bucket A.

Next, we update the local depth (LD) and sibling pointer (SP) of bucket A with an 8-byte atomic write, which adds bucket A' into the linked list and delete the migrated key-value pairs (those crossed-out keys in figure 5(c)) from bucket A. Finally, we update the metadata and directory.

REH also supports failure-atomic bucket merge, which is an inverse process of failure-atomic bucket split.

F. Concurrent Control

We follow Dash [8] to use optimistic concurrency, and the 32-bit lock in the metadata consists of a single bit that serves the role of "the lock" and a 31-bit version number for detecting conflicts. The insert operation sets the lock bit before making any changes to the bucket. After making its changes, it reset the lock bit and increases the version number in one step using an atomic write. While search operation first snapshots the 32-bit lock. After finishing its operations, it checks whether the lock has been set or the version number has changed. If so, it must retry.

We employ reader-writer lock to protect the directory and treat directory doubling/halving as a write operation. And treat directory updating (update a directory entry to point to a new bucket) as a read operation. Other operations (e.g., insert, search) do not hold any directory locks. However, they need to verify whether they entered the correct bucket after holding/snapshotting the bucket's lock by rereading the directory. If not, they must abort and retry the entire operation.

G. Recovery

In case of a system failure such as power loss, the directory and metadata can be reconstructed from consistent buckets in NVM. The pseudo code of the recovery function is shown in Algorithm 1. It starts by getting the first bucket in NVM. Then, it reconstructs the directory and metadata by scanning all the buckets using sibling pointers. Code line 10-19 shows how to rebuild the bitmap and fingerprints for the metadata using key-value pairs in the bucket. Since REH initializes buckets when first allocated, thus, an invalid key can't have a valid MSB bucket index by pure luck [9]. Code line 21-25 illustrates how to reconstruct directory entries. The recovery process is complete after all the buckets in NVM have been scanned.

Algorithm 1: Directory and Metadata Recovery

```
1  bucket_p = first_bucket;
2  i = 0;
3  while (bucket_p != NULL) do
4      local_depth = bucket_p→LD;
5      metadata_p = AllocDramMetadata();
6      metadata_p→Lock = 0;
7      metadata_p→LD = local_depth;
8      metadata_p→BP = bucket_p;
9      MSB = i;
10     for (j = 0; j <15; j++) do
11         key = bucket_p→entries[j].key;
12         key_hash = hash(key);
13         if (key_hash ≫ (64 − LD) == MSB) then
14             metadata_p→bitmap[j] = 1;
15             metadata_p→fingerprints[j] = key_hash & 0xff;
16         else
17             metadata_p→bitmap[j] = 0;
18         end
19     end
20     stride = 2^(global_depth − local_depth);
21     while (stride != 0) do
22         direcotry[i].metadata_p = metadata_p;
23         stride = stride − 1;
24         i = i + 1;
25     end
26     bucket_p = bucket_p→SP;
27 end
```

IV. Evaluation

A. Experimental Setup

Our experiments run on a server equipped with two Intel Xeon Gold 6240 CPUs, 512GB of Optane DCPMM (4x128GB DIMMs) configured in App Direct mode, and 128GB DRAM. The CPU has 18 cores (36 hyperthreads) and a shared 24.75MB L3 cache. The server runs Linux with kernel version 5.8.0. To avoid the NUMA effect, we conduct experiments on CPU 0 and attach all the Optane DCPMMs to this CPU.

We compare our REH with state-of-the-art NVM-based hashing indexes such as Level hashing [14], CCEH [9], and Dash [8]. We evaluate each hash table with individual operations (insert, positive and negative search, and delete) and mixed workloads. We insert 500 million key-value pairs into an empty hash table to measure insert-only performance. For other operations, we pre-load the hash table with 500 million key-value pairs, then execute 500 million operations to conduct

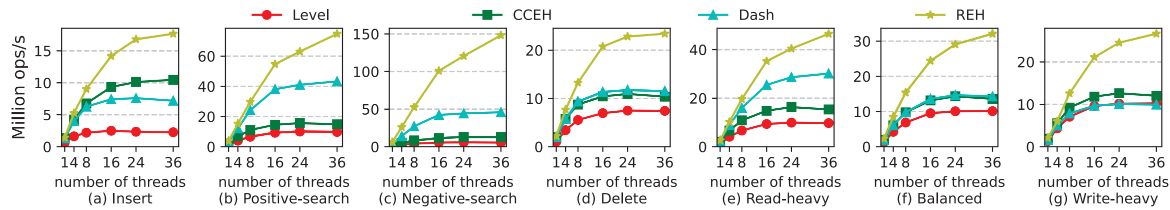


Fig. 6. Throughput under different workloads with a varying number of threads.

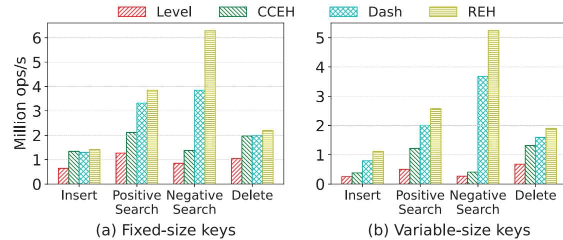


Fig. 7. Single-thread throughput.

the measurements. For fixed-size key experiments, both keys and values are 8-byte integers, and for variable-size keys experiments, we use 16-byte keys (the keys are pointed to by 8-byte pointers) and 8-byte values as previous work [8].

B. Single-thread Performance

We begin our evaluation of each hash table with a single thread. We first analyze search operations with fixed-size keys. Search is the most basic operation of the hash table since modifications to the hash table (such as insert and delete) need to verify whether the key exists at first.

As figure 7(a) shows, REH can outperform Level hashing/CCEH/Dash by 3.03×/1.81×/1.16× for positive search. This is because REH places directory and metadata in fast DRAM and the fingerprints in metadata can further reduce unnecessary NVM reads. For negative search, REH achieves more significant improvement, obtaining up to 7.40×/4.59×/1.63× speedup than Level hashing/CCEH/Dash. This is because fingerprints in metadata can completely avoid NVM access.

For insert operations, REH exhibits slightly higher throughput than CCEH and Dash due to its superior search performance, and only one cacheline flush per insert operation is needed. For delete operations, REH outperforms Level hashing/CCEH/Dash by 2.12×/1.12×/1.10× for the same reason.

REH still shows excellent performance for variable-size keys. As shown in figure 7(b), REH obtains up to 5.14×/2.11×/1.27× speedup than Level hashing/CCEH/Dash for positive search. The benefits of negative search are even more pronounced for REH (19.41×, 12.78×, 1.42×). This is because fingerprints can avoid unnecessary NVM access, especially for negative search and variable-size keys. For the same reason, REH can outperform Level hashing/CCEH/Dash by 4.44×/2.92×/1.41× for insert operations, and 2.79×/1.45×/1.19× for delete operations.

C. Scalability

We evaluate each hash table with individual operations and mixed workloads (write-heavy: 80% inserts and 20% searches, balanced: 50% inserts and 50% searches, read-heavy: 20% inserts and 80% searches). Figure 6 shows the multi-thread scalability of each hash table under workloads with fixed-size keys.

For insert operations, REH exhibits the best scalability because of its reduced writes to NVM, and metadata updates and lock operations only need to write to DRAM. Dash shows worse scalability than CCEH due to its optimizations (such as balanced insert, displacement, and stashing) to improve the load factor, which incurs more NVM writes than CCEH. Level hashing shows the worst scalability because it needs to lock the whole hash table during full-table rehashing, which blocks concurrent operations.

For search operations, figure 6(b-c) shows near-linear scalability for REH mainly due to its reduced reads from NVM, especially for negative searches. Dash exhibits better scalability than CCEH because it employs the optimistic concurrent control, which avoids writes to NVM for search operations. Level hashing shows the worst scalability due to more NVM writes, because it needs to acquire/release read locks even for search operations. Delete operations in REH, Dash, CCEH, Level hashing under 36 threads scale and improve over their single-thread version by 10.64×, 5.76×, 5.26×, 7.14×, respectively. For mixed workloads, REH also exhibits the best scalability as shown in figure 6(e-g).

We observed similar trends for workloads with variable-size keys and did not show them due to the limited space.

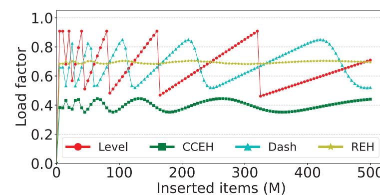


Fig. 8. Load factor.

D. Load factor

To measure the memory utilization of each hash table, we record the load factor after every 5 million insertions. As shown in figure 8, the load factor of Level hashing fluctuates between

0.46∼0.91 due to full-table rehashing. CCEH, Dash, and REH show more smooth curves because they allocate segment/bucket on demand. The load factor of REH is even more stable because REH splits a small bucket instead of a large segment when a hash collision happens. The load factors of REH, Dash, and CCEH fluctuate between 0.68∼0.70, 0.46∼0.85, and 0.35∼0.45, respectively, while inserting 500 million key-value pairs into the hash tables. When the hash tables are big enough, the load factors of REH, Dash, and CCEH will level off to 0.69, 0.65, and 0.40, respectively. Thus, REH can achieve the highest load factor among REH, Dash, and CCEH.

E. Recovery time

To measure the recovery time of REH, we first insert a certain number of key-value pairs into the hash table and then kill the process and record the time needed for the system to reconstruct the directory and the metadata. When we insert 5 million, 50 million, and 500 million key-value pairs into the hash table, our experiments show that the recovery process takes 0.33 seconds, 3.61 seconds, and 37.04 seconds, respectively.

V. Related work

A. Hashing-based Index Structures for NVM

In addition to Level hashing [14], CCEH [9], and Dash [8] mentioned before, several static hash tables for NVM were recently proposed. PFHT [3] is a cuckoo hashing variant tailored to PCM characteristics, and it avoids cascading writes by allowing at most one displacement. Path hashing [13] is a write-optimized hash table, and it leverages the position sharing method to resolve hash collision. P-CLHT is a crash-consistent hash table converted from a cache-line hash table by RECIPE [6], and it supports lock-free search, but it needs to hold the bucket-level lock for insertion and deletion. Clevel [2] is a lock-free concurrent hash table based on Level hashing, and the multi-level structure allows the resizing operations to run concurrently with query operations (e.g., insertion, search).

B. Tree-based Index Structures for NVM

Due to efficient support for range scans, much effort has been put into optimizing tree-based index structures for NVM. FPTree [10] is a hybrid DRAM-NVM persistent and concurrent B^+-Tree. The leaf nodes are persisted in NVM, while inner nodes are placed in DRAM. FPTree uses fingerprinting to improve probing performance in leaf nodes. LB^+-Tree [7] is based on FPTree, and it is specifically optimized for 3DXPoint memory. Our work is mainly inspired by FPTree and LB^+-Tree.

VI. Conclusion

To achieve scalability performance as well as a high load factor for the hashing index when running on Optane DCPMM, we design a new persistent hashing index, called REH, based on extendible hashing. REH (1) stores buckets in NVM and places directory and metadata in DRAM to reduce both unnecessary NVM reads and writes, (2) uses 256B sized-buckets, and those buckets are directly pointed to by directory entries, (3) leverages fingerprinting to further reduce unnecessary NVM reads, (4) employs failure-atomic bucket split to reduce bucket split overhead. Evaluations show that REH significantly outperforms the state-of-the-art NVM-based hashing indexes, and REH can achieve a high load factor.

Acknowledgment

This work was supported in part by National Key R&D Program of China NO.2018YFB10033005, NSFC No.61832020, No.61821003, National Science and Technology Major Project No.2017ZX01032-101. This work was jointly completed with Key Laboratory of Information Storage System and Engineering Research Center for Data Storage Systems and Technology, Ministry of Education, China.

References

[1] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, “Faster: A concurrent key-value store with in-place updates,” in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 275–290.

[2] Z. Chen, Y. Huang, B. Ding, and P. Zuo, “Lock-free concurrent level hashing for persistent memory,” in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 799–812.

[3] 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.

[4] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, “Extendible hashing—a fast access method for dynamic files,” *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 3, pp. 315–344, 1979.

[5] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen, “Persistent memory hash indexes: an experimental evaluation,” *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 785–798, 2021.

[6] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and 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.

[7] J. Liu, S. Chen, and L. Wang, “Lb+ trees: Optimizing persistent index performance on 3dxpoint memory,” *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, 2020.

[8] B. Lu, X. Hao, T. Wang, and E. Lo, “Dash: scalable hashing on persistent memory,” *Proceedings of the VLDB Endowment*, vol. 13, no. 10, pp. 1147–1161, 2020.

[9] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and 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] 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*, 2016, pp. 371–386.

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

[12] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, “Reducing the storage overhead of main-memory oltp databases with hybrid indexes,” in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1567–1581.

[13] P. Zuo and Y. Hua, “A write-friendly and cache-optimized hashing scheme for non-volatile memory systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 985–998, 2017.

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

[15] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua, “One-sided rdma-conscious extendible hashing for disaggregated memory,” in *2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21)*, 2021, pp. 15–29.