Persistent Memory Hash Indexes: An Experimental Evaluation

Daokun Hu*
Hunan University, China
daokunhu@hnu.edu.cn

Zhiwen Chen*
Hunan University, China
zhiwenchen@hnu.edu.cn

Jianbing Wu
Hunan University, China
kimbing@hnu.edu.cn

Jianhua Sun
Hunan University, China
jhsun@hnu.edu.cn

Hao Chen
Hunan University, China
haochen@hnu.edu.cn

ABSTRACT

Persistent memory (PM) is increasingly being leveraged to build hash-based indexing structures featuring cheap persistence, high performance, and instant recovery, especially with the recent release of Intel Optane DC Persistent Memory Modules. However, most of them are evaluated on DRAM-based emulators with unreal assumptions, or focus on the evaluation of specific metrics with important properties sidestepped. Thus, it is essential to understand how well the proposed hash indexes perform on real PM and how they differentiate from each other if a wider range of performance metrics are considered.

To this end, this paper provides a comprehensive evaluation of persistent hash tables. In particular, we focus on the evaluation of six state-of-the-art hash tables including Level hashing, CCEH, Dash, PCLHT, Clevel, and SOFT, with real PM hardware. Our evaluation was conducted using a unified benchmarking framework and representative workloads. Besides characterizing common performance properties, we also explore how hardware configurations (such as PM bandwidth, CPU instructions, and NUMA) affect the performance of PM-based hash tables. With our in-depth analysis, we identify design trade-offs and good paradigms in prior arts, and suggest desirable optimizations and directions for the future development of PM-based hash tables.

PVLDB Reference Format:
Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. Persistent Memory Hash Indexes: An Experimental Evaluation. PVLDB, 14(5): 785 - 798, 2021.
doi:10.14778/3446095.3446101

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/daokunhu/HashEvaluation.

1 INTRODUCTION

Until now, various types of persistent memory based on different physical mediums have been proposed [22, 59, 64]. Despite this difference, persistent memory is commonly featured by byte-addressability, direct persistence, and DRAM-scale latency. 3D XPoint is one of the well known PM technologies, and Intel Optane DC Persistent Memory Modules (DCPMM) based on this technology is the first-to-market PM product [43]. Recently, the performance characteristics of Optane DCPMM were explored and evaluated from both micro and macro perspectives [33, 63, 68]. Some insights are drawn from these studies, and part of them are inconsistent with previous assumptions. J. Yang et al. find that the actual behaviors of Optane DCPMM are more complicated and nuanced than the *slower, persistent DRAM* label would suggest [68]. The performance of applications on PM is highly coupled with the access size, workload, pattern, and degree of concurrency compared to DRAM. Contrary to the estimation made in previous work [61], Lersch et al. [33] show that PM bandwidth is a scarce resource and has a significant impact on performance. The results in [68] show that end-to-end write latency is often lower than read latency. These characteristics about PM hardware are ignored or not comprehensively considered in existing studies like the design of PM data structures. For example, given the incorrect assumption about the asymmetric read/write performance, some prior works [46, 72] use schemes that intentionally impose more PM reads to reduce PM writes.

The advent of scalable persistent memory offers a new way to address the issues that many practical applications or systems have been facing. Among them, the hash indexing structure or hash table, a fundamental component in many software systems [16, 36, 50, 53], can benefit from the new features of PM to achieve high performance and instant recovery. There has been a new breed of hash tables specifically designed for PM [31, 37, 46, 72, 73]. They are either based on DRAM emulation [46, 72, 73], or designed for actual PM [9, 37], or ported from existing hash tables [31]. It is unclear how well these proposed hash tables (evaluated using emulators) perform on real PM, how they differentiate from each other if a wider range of performance metrics are considered, and how they behave under specific hardware configurations such as NUMA that is only available on certain CPU architectures (like Cascade Lake [1]).

Inspired by existing circumstances, in this paper, we conduct a comprehensive evaluation of hash tables on a system equipped with DCPMM. We evaluate six PM-based hash tables including Level hashing [72], CCEH [46], Dash [37], PCLHT [31], Clevel [9], and SOFT [73], using a benchmarking framework that is an extension of PiBench [33] that provides a set of interfaces and a unified testing environment for fair comparison. The six hash tables cover a wide range of techniques in various dimensions, as detailed in Section 3. Our experimental results obtained from running representative workloads reveal important insights that can be adopted

*The first two authors contributed equally to this work.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 14, No. 5 ISSN 2150-8097.
doi:10.14778/3446095.3446101

as guidelines for the future development of PM-based hash tables (summarized in Section 5). Overall, we make the following contributions.

- First, we leverage the PiBench framework in our hash table evaluation with necessary extensions. In this way, we can guarantee that the experimental results are comparable and fair. Our benchmarking framework is publicly available at https://github.com/daokunhu/HashEvaluation.
- Second, we evaluate PM-based concurrent hash tables from both macro and micro perspectives. In addition to the commonly used metrics such as throughput, scalability, latency, load factor, LLC misses, resizing overhead, memory usage, and recovery overhead, we also consider the impact of PM hardware such as DCPMM bandwidth, various combinations of instructions, and NUMA architecture. The experiments are all conducted on real PM hardware.
- Third, implications about design trade-offs, good paradigms, and desirable optimizations are summarized, which can serve as guidelines for future research and practical development of PM-based hash tables.

The rest of this paper is organized as follows. Section 2 provides background on Intel Optane DCPMM and Persistent Memory Development Kit (PMDK) [25]. We introduce the design and implementation of six state-of-the-art hash tables in Section 3. The evaluation results are discussed in Section 4. We summarize important observations in Section 5. Related work and conclusion are presented in Section 6 and Section 7 respectively.

2 BACKGROUND

This section first presents the background of Optane DC Persistent Memory. Then, we briefly introduce PMDK that is the programming framework commonly used to implement PM applications.

2.1 Optane DC Persistent Memory

Optane DC Persistent Memory (hereinafter referred to as Optane DCPMM) is the first commercially available PM product that modifies the traditional computer memory hierarchy by creating a new non-volatile tier between the volatile DRAM and block-based storage.

Hardware Architecture. Optane DCPMM was introduced with the Cascade Lake architecture [1], which supports multiple sockets (2/4/8), each consisting of one or two processor dies that comprise separate NUMA nodes [68]. The integrated memory controller (iMC) on Cascade Lake is capable of interfacing with both DDR4 and Optane DCPMM DIMMs. To maintain data persistence, the iMCs are located in the asynchronous DRAM refresh (ADR) domain that can ensure CPU stores that arrive here will survive a power failure [27]. The iMC maintains both read and write pending queues (RPQs and WPQs) for each Optane DCPMM, but only the WPQs are in the ADR domain. Therefore, once data reaches the WPQs, it will be flushed to PM by the iMC on a system crash. As the 3D-XPoint physical medium access granularity is 256 bytes, the on-DIMM controller translates smaller requests into larger 256-byte ones, causing write amplification as small stores become read-modify-write operations. The on-DIMM controller has a small write-combining buffer (referred to as XPBuffer [68]) to merge adjacent writes. Similar to WPQs, the XPBuffer resides in the ADR domain. Hence, all updates are already persistent when they arrive at the XPBuffer.

Operating modes. Optane DCPMM can be configured to run in *Memory* mode or *App Direct* mode [23]. Memory can be interleaved across channels and DIMMs in each mode. Both modes allow direct access to PM via `load` and `store` instructions. In *Memory* mode, the DRAM acts as a cache for the most frequently-accessed data, while the DCPMM provides large memory capacity *without* persistence. In *App Direct* mode, applications and operating system are explicitly aware there are two types of directly accessible memory using load/store instructions. It is worth noting that although PM is persistent, CPU caches and registers are still volatile. Data will be persisted in PM when a cacheline flush instruction, such as `clflush`, `clflushopt`, or `clwb`, is executed or other events that implicitly cause cacheline flush occur [37]. In order to ensure that applications can recover from system crashes correctly, we must use memory fences to avoid undesirable reorderings of instructions. Since our main target is to evaluate the performance of PM-based hash tables, we configure Optane DCPMM in the *App Direct* mode.

Performance. Although Optane DCPMM is designed to work with direct and byte-addressable load/store access, it has lower bandwidth and higher read/write latency than DRAM. As reported in previous work [33], the read latency of Optane DCPMM is ~300 ns, which is 4x of that of DRAM (~75 ns). Moreover, it exhibits asymmetric read/write latency. According to a more recent study [68], end-to-end write latency *as seen by the application* is often lower than read latency, because writes return once data reaches the ADR domain at the memory controller. But reads have a high probability of accessing the physical DCPMM medium if the data is not cached in RPQs. The maximum sequential read and write bandwidth of DCPMM is about 40 GB/s and 13 GB/s [68], which are about 3x and 11x lower than that of DDR4 DRAM, respectively. The asymmetric bandwidth is more prominent for random reads and writes.

2.2 Programming Persistent Memory

On the one hand, PM allows us to build data structures that are much faster than those requiring serialization or flushing to block-based storage [58]. On the other hand, PM breaks the volatile-persistent boundary between the traditional volatile memory and external memory, creating a new boundary between processor cache and PM. These changes make the programming of PM systems more complicated, challenging, and error-prone [54]. For example, algorithms must be carefully tailored to properly persist data by flushing the CPU cache or using non-temporal store and memory barrier to ensure data consistency. However, data consistency requires the proper ordering of stores and making sure data is stored persistently. To make a store (that writes more than 8 bytes) atomic, we typically rely on certain mechanisms, such as redo/undo logging [4, 26], copy-on-write [2, 39, 46], versioning [37], and hybrid memory [65]. These mechanisms are complicated and error-prone.

These challenges can be addressed by employing a sound programming model enforced by PM programming libraries [20, 25, 30, 40, 60]. In order to be consistent with the original implementation of hash tables as most as possible, we use the libraries in PMDK to map memory files, allocate PM memory, and persist data. Furthermore, we use xfs [10], a file system that has native DAX support

(direct access to files backed by persistent memory) to organize and manage PM data in our experiments.

3 PERSISTENT HASH TABLES

In this section, we briefly introduce six PM-based hash tables, including Level hashing [72], Clevel hashing [9], CCEH [46], Dash [37], PCLHT [31], and SOFT [73]. These hash tables can be classified into the following categories: 1) converted from DRAM counterparts or specifically designed for real PM hardware; 2) based on various structures like separate chaining, open addressing, or extendible hashing; 3) using different synchronization mechanisms (lock-based or lock-free); 4) using distinct resizing strategy (dynamic or static); and 5) employing different crash-consistency mechanisms. We believe they are sufficiently representative.

3.1 Level Hashing and Clevel

Level hashing [72] is a write-optimized and scalable hash table designed for persistent memory, featuring low overhead consistency guarantee and cost-efficient resizing. Level hashing adopts a sharing-based two-level structure to achieve constant-scale time complexity. As shown in Figure 1(a), only the top-level buckets are addressable, and buckets in the bottom-level are used to store conflicting items evicted from the top-level buckets. Similar to the PCM-friendly hash table (PFHT) [12], it has multiple slots in each bucket and allows evicting at most one item when inserting an item to a full bucket. It enables each key to have two target locations to choose from (via two hash functions), like Cuckoo hashing [51], to improve load factor.

In Level hashing, a four times larger hash table is created during resizing, and the key-value pairs in bottom-level buckets are first rehashed into the new table, then the buckets of the old bottom-level are deleted. At this time, the new table becomes the top level, while the previous top level becomes the bottom level. To achieve low overhead consistency, it adopts log-free schemes in the operations such as delete, insert, and resize, through a bitmap in each bucket's head. Level hashing relies on a slot-grained lock for concurrency control, and no pointers are maintained in its data structure. Although Level hashing shows improvement over previous work [12, 71], it has two shortcomings. First, it is a static hashing scheme, and its rehashing overhead is still high. Second, it was not evaluated on real PM.

Clevel is a lock-free concurrent hash table based on Level hashing, and it proposes a dynamic multi-level structure to support concurrency. Resizing is performed by background threads without blocking concurrent queries.

3.2 CCEH

CCEH inherits the design of extendible hashing [15] to address the issues of Level hashing, and is featured with low-overhead dynamic memory management, constant lookup time, and failure-atomicity guarantee without explicit logging. In traditional extendible hashing, a directory is maintained to include pointers that store the addresses of buckets. When the number of buckets increases, the directory may consume a large amount of memory, making it unable to reside in CPU cache. As a result, CCEH proposes an intermediate layer (referred to as *segment*) between the directory and buckets to reduce the number of pointers used to address buckets. This structure is illustrated in Figure 1(b), where a directory entry points to a segment that consists of a fixed number of buckets indexed by the L least significant bits (LSBs) of hash values, and the segments are indexed by the G most significant bits (MSBs) of hash values. By combining multiple buckets into a segment, the directory remains significantly smaller as fewer bits are required to address the segments.

When a collision happens upon inserting an item and the target bucket is full, instead of splitting the bucket (as in traditional extendible hashing), CCEH splits the segment to expand capacity. However, segment splitting would result in low load factors and more PM accesses, because other buckets in the segment may still have free slots. Thus, CCEH uses linear probing (before splitting) in attempt to improve space utilization. After the segment splitting is completed, some items need to be migrated to the newly created segment. CCEH uses a *lazy deletion* strategy to make sure the migrated items in the old segment do not need to be cleaned immediately, since they will be ignored by *search* operations and overwritten by *insert* operations. In this way, extra writes are avoided.

3.3 Dynamic and Scalable Hashing (Dash)

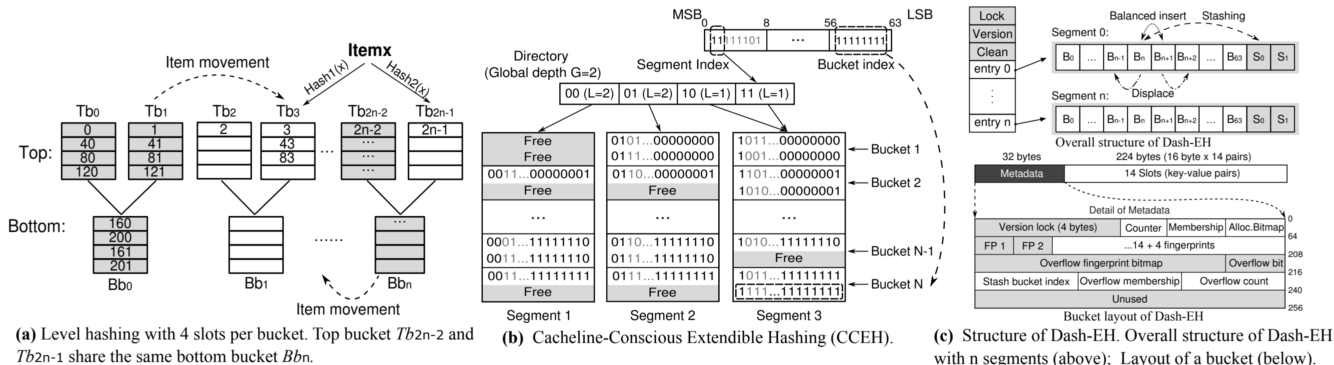
Dash [37] is a holistic approach to building dynamic and scalable hash tables. It is adapted to implement extendible hashing and linear hashing denoted as Dash-EH and DASH-LH. Similar to CCEH [46], Dash leverages a three-level structure consisting of a directory and a number of segments that are composed of numerous buckets. Figure 1(c) shows the structure of Dash (Dash-EH) based on extendible hashing. A directory entry points to a segment which consists of 64 normal buckets and two stash buckets used to store overflowed key-value pairs. Both normal and stash bucket share the same layout, as shown in Figure 1(c) (below). Each bucket is 256 bytes in size, and has 32-byte metadata in its head, which is followed by 14 key-value pairs.

Dash employs techniques such as *balanced insert*, *displacement*, and *stashing* to address the issue of low load factor in CCEH. To insert a key, it first calculates the segment index s and bucket index b by the key's hash value, then it finds an empty slot in the less full bucket b or b+1 (as shown in Figure 1(c) above) according to the allocation bitmap in the metadata. If both of the buckets b and b+1 are full, a *displacement* operation is triggered to make room for the new key being inserted. A membership bitmap is maintained in the metadata to accelerate displacement. The key will be inserted into a stash bucket if both of the two operations (balanced insert and displacement) fail. If the insert operation is successful, the original bucket's overflow flag is updated; otherwise, a segment splitting happens. In the worst case, Dash has to probe bucket b, b+1, and stash buckets for a query, incurring more cache misses and PM reads. The fingerprints, which are one-byte hash values of keys, are maintained in the metadata to accelerate negative queries.

Dash uses optimistic concurrency control, and employs a bucket-level CAS lock for the insert operation. The search operation in Dash is lock-free, but we need to verify the keys returned.

3.4 PCLHT

Persistent Cache-Line Hash Table (PCLHT) is a chaining-based concurrent crash-consistent hash table discussed in RECIPE [31]. It is a variant of the DRAM-based hash table CLHT [11].



(a) Level hashing with 4 slots per bucket. Top bucket *Tb*2n-2 and *Tb*2n-1 share the same bottom bucket *Bb*n.

(b) Cacheline-Conscious Extendible Hashing (CCEH).

(c) Structure of Dash-EH. Overall structure of Dash-EH with n segments (above); Layout of a bucket (below).

Figure 1: Structure of three PM-based hash tables. (a) Level hashing; (b) CCEH; (c) Dynamic and Scalable Hashing (Dash).

CLHT is cache-friendly because its bucket is 64-byte in size (a common cacheline size). Each bucket contains an 8-byte word for concurrency control, a next pointer, and at most three 16-byte key-value pairs. This design aims at addressing the cache coherence problem by ensuring that each update to the hash table requires only one cacheline access in ordinary cases. To ensure that a non-blocking read finds the correct result, CLHT uses atomic snapshot of bucket for write operations. The insert and delete operation both use a single atomic commit point that is ordered by memory fences: writing the correct value first before updating the 8-byte key (for insert) or writing 0 to the key (for delete). If an insert fails because the table is full, then CLHT resizes the table using a copy-on-write method.

Because the operations like insert, delete, and resize are implemented via a single atomic store, they can be converted to persistent counterparts by simply inserting cacheline flushes and memory fences after corresponding `store` instructions.

3.5 SOFT

SOFT [73] is a chaining-based lock-free hash table proposed to avoid persisting any pointers in the data structure. SOFT has two main components, i.e., persistent node (PNode) and volatile node (VNode). The PNode contains a key-value pair and three validity bits. The VNode consists of a key-value pair and two pointers: one pointing to the PNode with the same key-value pair, and the other pointing to the next VNode. Only PNodes are persisted in PM, and all VNodes are stored in DRAM. The resizing operation is not implemented in SOFT. When inserting a key, a new PNode and a new VNode are allocated, and the VNode will be linked to an existing VNode atomically (via CAS). The delete operation shares a similar process with the insert operation except that nodes are removed instead. SOFT can recover from a system crash by scanning the PNodes in PM and rebuilding the structure in DRAM.

3.6 Summary

We summarize the characteristics of hash tables discussed above in Table 1, including data structures, concurrency control mechanisms, memory architectures, and resizing strategies. These techniques are commonly used in the design of PM hash tables.

Structure. CCEH and Dash both use a directory-segment based structure. This three-layer structure is more space-efficient than traditional extendible hashing. Level hashing uses a sharing-based two-level structure, which offers constant-scale time complexity for writes. PCLHT and SOFT are both chaining-based design using pointers to link buckets (or nodes).

Concurrency. Except SOFT and Clevel, all other hash tables are lock-based. Level hashing uses slot-grained locking, CCEH adapts two-level reader/writer locking, and PCLHT uses bucket-grained locking. Dash employs a optimistic locking scheme, and writes are protected by bucket-grained lock, while reads are lock-free.

Architecture. SOFT stores volatile nodes and persistent nodes in DRAM and PM respectively. This hybrid architecture make it possible to achieve near DRAM performance, but it suffers from long recovery time, depending on the size of the structure that needs to be rebuilt in DRAM. The other four hash tables store all data in PM, which may suffer from longer read/write latency.

Resizing. Both CCEH and Dash split segment to expand space. Segment splitting incurs much fewer PM accesses than whole table rehashing. In addition, CCEH employs a strategy called *lazy deletion* to further reduce the overhead of segment splitting. In PCLHT and Level hashing, the resizing operation doubles the size of the hash table. But the bucket-sharing structure of Level hashing enables it to reuses 2/3 of buckets during resizing. The resizing operation of Clevel is lock-free. SOFT does not support resizing.

4 EXPERIMENTAL EVALUATION

4.1 Environment and Setup

Hardware Configuration. Our experiments were performed on a Linux server (kernel version 5.0.0) that is equipped with two Intel Xeon Gold 5218 CPUs that both are clocked at 2.3GHz and have 16 cores, 32 hyperthreads, 32KB L1 instruction cache, 32KB L1 data cache, and 1024KB L2 cache. The last level cache is 22MB in size. The memory system includes 192GB of DDR4 DRAM and 768GB of Optane DCPMM (6 x 128 GB DCPMMs) configured in the App Direct mode. In the evaluation of multi-threaded performance and NUMA, we use two CPUs. In other cases, only one CPU is used and the DCPMMs are all installed with this CPU.

Parameters. To perform a fair comparison among all hash tables, we use the configuration as presented in their original papers except Level hashing. The default size of keys and values in Level hashing is 16 bytes and 15 bytes respectively. Thus, a bucket with 3 slots and a 3-byte token can be aligned to 128 bytes (two cachelines). We choose to store keys and values using 8 bytes, because other

Table 1: Comparison of persistent hash tables.

	Structure	Concurrency	Architecture	Resizing
Level hashing	Open addressing (two-level)	Locking	PM-only	Doubling
CCEH	Extendible hashing	Selective (locking & lock-free)	PM-only	Splitting
Dash	Extendible hashing	Optimistic locking	PM-only	Splitting
PCLHT	Separate chaining	Locking	PM-only	Doubling
SOFT	Separate chaining	Lock-free	DRAM (VNode) + PM (PNode)	–
Clevel	Open addressing (multi-level)	Lock-free	PM-only	Doubling

hash tables only support this configuration. Thus, we set the bucket size of Level hashing to 64 bytes. CCEH uses 16KB segments and 64-byte buckets (4 slots), and probes at most 4 cachelines (4 buckets, 16 slots, 256 bytes). In Dash, a bucket has 14 slots that consume 256 bytes memory, and a segment contains 64 normal buckets and 2 stash buckets. PCLHT is chaining-based, and its bucket is aligned to 64 bytes. SOFT is also chaining-based with one key-value pair in each node, and it creates and destroys node on demand.

Implementation. For CCEH, Level, PCLHT, and SOFT, we use the library `libvmem` [24] of PMDK [25] to manage PM memory. For Dash, we directly adopt its original implementation, which uses the interfaces provided by `libpmem` and `libpmemobj` in PMDK for crash-safe PM management. For Clevel, we also use its original implementation using PMDK with C++ bindings. For cacheline flush, we use the `clwb` instruction for better performance [31]. As for the hash function, we employ GCC's std::Hash_bytes that is also used in previous studies [33, 46, 72], because it is known to be fast and provides high-quality hashes [37]. All hash tables do not allow duplicate keys.

Benchmark Framework. Due to its easy adoption and well-designed architecture, we leverage the PiBench [33] framework in our evaluation with necessary extensions. It collects a wide range of metrics, including throughput, latency, and hardware counters using the Processor Counter Monitor (PCM) library [13] through well-defined interfaces that can invoke common operations of indexing structures, such as insert, search, delete, and update etc. We extend PiBench with functionalities that are specific to evaluating hash tables, such as load factor, utilization, resize latency, recovery time, and the number of instructions targeting PM (cache flushing and memory fence). To evaluate a specific hash table, we simply need to implement the interfaces defined by PiBench and encapsulate them in a shared library, which can be loaded by PiBench at runtime to invoke the corresponding interfaces. Moreover, we use the three random distributions offered by PiBench in workload generation, i.e., uniform, self similar, and zipfian.

Workloads. We stress test each hash table with individual operations (insert, positive and negative search, and delete) and mixed workloads. Negative search means searching keys that do not exist. Unless otherwise stated, we use the following method to generate workloads for corresponding experiments. We initialize hash tables with a capacity that can accommodate 16M key-value pairs except SOFT. Because SOFT creates and destroys nodes on demand, and we initialize it with 16M head nodes (the head node of a list is not used for data storage). To measure insert-only performance, we insert 200M records into an empty hash table directly. To measure the performance of the search and delete operation and the mixed workloads, we first initialize the hash table with 200M items (loading phase), then execute 200M operations to perform the measurements (measuring phase). Similar to prior studies [33, 37], we run the experiments with workloads using uniform distribution and skewed distribution (self similar with a factor of 0.2, which means 80% of accesses focus on 20% of keys). Since CCEH, Level hashing, and PCLHT have no support for variable-length keys and values, we only consider fixed-length (8 bytes) keys and values.

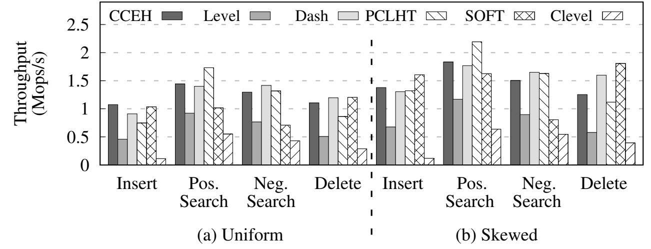


Figure 2: Single-threaded throughput with (a) uniform distribution (b) skewed (self similar) distribution (80% access to 20% of data)

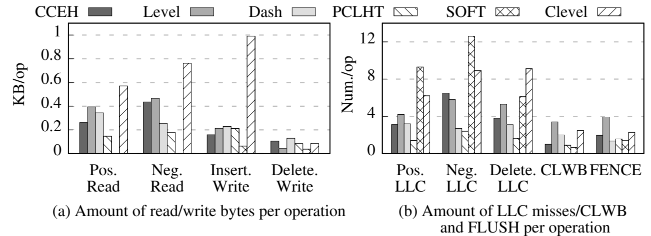


Figure 3: Low-level metrics demonstrating the amount of (a) read/write bytes per operation (b) LLC misses/instructions for different operations.

4.2 Single-threaded Performance

We start our evaluation with the analysis of single-threaded performance. The results are shown in Figure 2.

Search. Search is the most basic operation of hash tables since updates to hash tables (such as insert and delete) need to confirm whether a certain key exists at first. PCLHT has the highest performance in positive search, which benefits from its design philosophy [11]: *the search operation should not involve any stores, waiting, or retries.* In addition, PCLHT performs a full-table rehashing to expand capacity, which ensures its chaining-based bucket is short enough to obtain constant time search performance when the hash

table is nearly full. To illustrate more clearly, we collect the last level cache (LLC) misses when running the search-only workload. As shown in Figure 3(b), the number of LLC misses of PCLHT is the lowest, and PCLHT only needs 1.4 cacheline accesses on average per search. Besides, PCLHT reads less data as shown in Figure 3(a). In comparison, SOFT needs to traverse its long chained buckets, which incurs the largest number LLC misses (see Figure 3(b)). Fortunately, SOFT reads keys from VNodes stored in DRAM (zero reads from PM shown in Figure 3(a)). We also notice that SOFT achieves higher throughput in positive search than in negative (1.4x and 2.0x under uniform and skewed distribution respectively). Because SOFT needs to traverse all nodes in negative search, causing more LLC misses (see Figure 3(b)).

Dash maintains a fingerprint array in the metadata, which can help avoid unnecessary PM accesses. With this optimization, Dash achieves the highest throughput in negative search. To find an existing item, Dash needs to probe 4 buckets (target bucket, neighbor bucket, and two stash buckets) in the worst case, while CCEH needs to probe 5 consecutive buckets. It means that Dash needs to access 4 XPlines in the XPBuffer in the worst case, while CCEH only needs to access 2 XPlines at most. Consequently, Dash reads more data (about 1.3x of CCEH) from PM (see Figure 3(a)), leading to slightly lower throughput in positive search compared to CCEH. Clevel reads much more data from PM than all other hash tables, since it needs to probe all levels in the hash table and dereference pointers to get the final key-value pairs. As a result, Clevel has the lowest throughput.

Insert. To insert a record, we first search the hash table to make sure no record with the same key exists. Thus, the insert performance relies on how the search operation behaves.

As shown in Figure 2(a), for the insert-only workload with uniform distribution, the throughput of SOFT and CCEH is close to each other. As the insert-only workload would inevitably trigger table resizing if the initial capacity of hash tables is less than the total size of items to be inserted. Thus, the resizing overhead has a non-trivial impact on insert throughput. Level hashing and PCLHT exhibit lower insert throughput because of their time-consuming resizing operation, which incurs a large amount of extra PM writes. In addition, as Level hashing exhibits a lower search performance, its throughput falls behind PCLHT. As for CCEH and Dash, they have similar resizing overhead and search performance, but CCEH shows higher throughput because it needs fewer cacheline flushes per insert (see Figure 3(b)) and writes less data to PM (see Figure 3(a)). Because the skewed distribution generates duplicate keys, the throughput in this case is slightly different. PCLHT's throughput is improved greatly since its resizing overhead is decreased (fewer keys to be rehashed). Clevel has the lowest throughput, as it needs to dynamically allocate memory for the items being inserted and record a log entry for each item to guarantee crash consistency.

Delete. The performance of delete operation is also related to search performance, because before deleting a key, we need to check whether the key exists. The delete operation involves multiple reads but only one write. Hence, the delete performance shows a similar trend with positive search (keys to be deleted exist). SOFT frees the node after the key is deleted. This means the size of its linked list shrinks as more keys are deleted. With fewer nodes to probe, SOFT can perform the search operation faster. Thus, SOFT achieves the highest throughput for workloads under both uniform and skewed distribution. Comparatively, the delete throughput of PCLHT is lower than that of SOFT, even it exhibits outstanding search performance. This is due to the fact that PCLHT just marks a key as invalid, but the nodes containing the deleted keys remain, resulting in extra checks during searching. Under uniform distribution, Dash, CCEH, and SOFT exhibit similar performance. Due to its poor search performance, Clevel's delete throughput is also quite low.

4.3 Multi-threaded Performance

We use the same configurations as in Section 4.2 to evaluate multi-threaded performance. The results are shown in Figure 4, and Figure 5. Among the mixed workloads, the write heavy workload consists of 80% inserts and 20% searches, the balanced workload consists of 50% inserts and 50% searches, and the read heavy workload comprises 20% inserts and 80% searches.

Search. For the search operation, all hash tables scale well when the number of threads is less than 32, as shown in Figure 4(b)(c)(f)(g). PCLHT performs best, and this is consistent with its single-threaded performance. In particular, PCLHT shows excellent performance for negative search under skewed distribution, achieving 60Mops/s of throughput at 64 threads, which is 1.2x/2.2x/3.1x/4.3/7.1x of Dash/SOFT/Level/CCEH/Clevel. SOFT is a dynamic lock-free hash scheme and its search operation only accesses DRAM, which guarantees its good search performance. However, SOFT uses pointers to link nodes like PCLHT and does not resize the hash table. Hence, the length of the linked list would limit its overall search throughput. For Dash, threads can proceed without holding any locks for reads, which makes it achieve good scalability. The fingerprinting technique employed by Dash can indeed improve its negative search throughput, since it can filter negative lookups to PM. As a result, Dash outperforms PCLHT in negative search under uniform distribution (see Figure 4(c)). For the skewed workload, PCLHT is slightly better than Dash in negative search because much less of data (20%) is accessed compared to the uniform distribution. CCEH underperforms Dash due to its usage of pessimistic locking even for reads. In general, hash tables with non-blocking read can obtain higher search performance.

Insert. SOFT achieves significantly better insert performance than others under both workload distributions. As depicted in Figure 4(a)(e), its throughput is up to 14.7x and 13.7x of Level hashing under uniform and skewed distribution respectively, when the number of threads reaches 32. SOFT's outstanding insert performance can be attributed to four aspects: (1) It is lock-free. (2) It completely avoid persisting any pointers in the data structure. (3) It uses the least amount of flush instructions (Figure 3(b)). (4) The number of fence instructions per operation is close to the theoretical minimum (Figure 3(b)). Given that Dash needs to move items among buckets to deal with collisions, which incurs more flushes and writes, its throughput is slightly lower than that of CCEH. Level hashing exhibits the lowest throughput because the whole table is locked during resizing. In addition, Level hashing needs to delete items in buckets during resizing. This causes a lot of extra cache flushes and PM writes. Similar to Level hashing, PCLHT's surprisingly inferior insert performance is due to its resizing cost (the second-largest overhead detailed in Section 4.6). Clevel inherits the design of Level

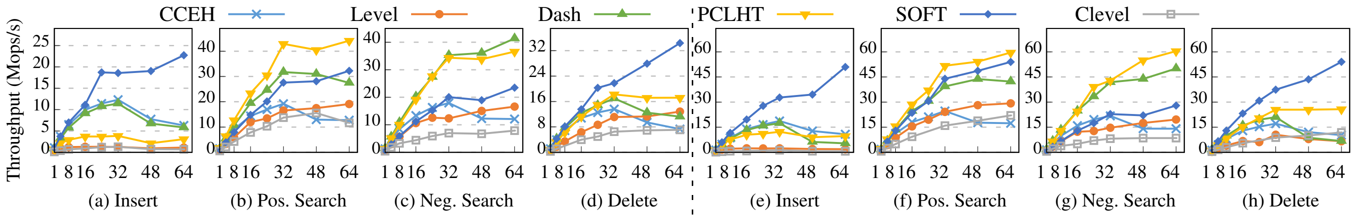


Figure 4: Multi-threaded throughput of uniform (left) and skewed (right) distribution workloads.

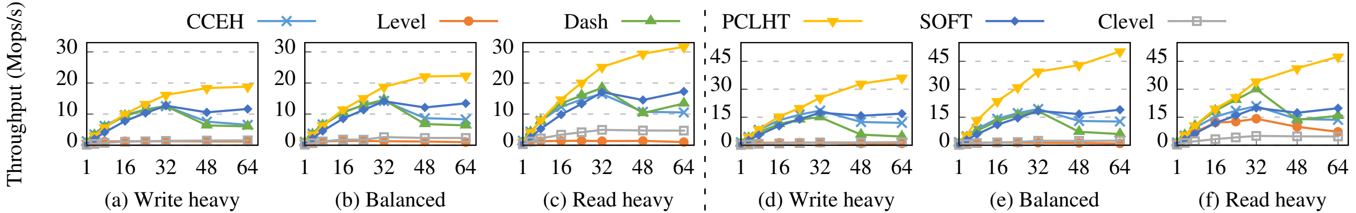


Figure 5: Multi-threaded throughput of mixed workloads under uniform (left) and skewed (right) distribution workloads.

hashing, so their insert performance is close to each other even Clevel has a lock-free optimization for resizing.

Delete. Similar to the insert operation, SOFT outperforms others significantly under the delete-only workload, as illustrated in Figure 4(d)(h). PCLHT has lower single-threaded performance for delete compared to CCEH and Dash (see Figure 2). But PCLHT performs better in the multi-threaded case. Because the delete operation involves both reads and writes, and PCLHT causes fewer reads but comparable writes with CCEH and Dash, resulting in more effective bandwidth utilization.

Mixed. To measure the performance of mixed workloads, we initialize hash tables with 200M records, then execute 200M operations with different read/write ratios. PCLHT achieves the best performance in all situations. Compared to the insert-only workload, PCLHT reaches a higher throughput in the write heavy scenario. The reason is that we insert 200M items before measuring the performance, which indicates free buckets are adequate and no resizing is triggered in this case. With the increasing of the percentage of search operations (comparing the write-heavy, balanced, and read-heavy workload), the throughput of PCLHT increases from 19Mops/s to 32Mops/s steadily under uniform distribution, and reaches approximately 47Mops/s under skewed distribution. CCEH, Dash, and SOFT exhibit identical trend for both write heavy and balanced workload.

Most hash tables achieve their maximum throughput at 32 threads. However, there are two exceptions. (1) SOFT achieves a continuous growth in throughput for the delete-only workload. This is because SOFT looks for keys in DRAM, so read accesses to PM are avoided. In addition, SOFT incurs less writes to PM as indicated in Figure 3(a). (2) PCLHT does not level off until 64 threads under mixed workload. This is because PCLHT has fewer reads/writes per operation than others, and more operations can be processed under the same bandwidth constrains.

4.4 DCPMM Scalability

To observe how the throughput scales with the number of DCPMMs, we rerun the multi-threaded benchmarks in Section 4.3 with only one DCPMM enabled. We define a metric *throughput ratio*, which

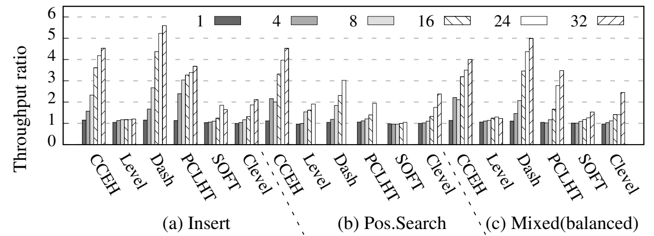


Figure 6: Scalability in throughput by comparing a single DCPMM with six interleaved DCPMMs (uniform).

is computed by dividing the throughput of using all six DCPMMs (interleaved) by that of using one DCPMM, to illustrate DCPMM scalability. A larger value of ratio indicates better DCPMM scalability. The maximum ratio is 6 if the throughput scales ideally. Figure 6 shows the results obtained by running different number of threads. CCEH and Dash have a larger throughput ratio especially with more threads stressing the PM subsystem, which means that their throughput can be improved by adding more DCPMMs. However, more DCPMMs does not necessarily lead to noticeable improvement in throughput even with a large amount of threads. This is the case for Level hashing and SOFT. Level hashing has a throughput ratio of about 1 for insert-only and mixed workload, while its maximum ratio is below 2 in the case of positive search. SOFT is a hybrid memory design that imposes more pressure on DRAM, so its throughput is not sensitive to the number of DCPMMs. For the search workload, SOFT's throughput ratio is approximately 1 no matter how many threads are used.

In summary, for PM bandwidth hungry hash tables, allocating more DCPMMs could effectively increase the performance. For others, we observe only marginal returns with additional DCPMMs installed. In particular, for hybrid memory designs such as SOFT, systems should be configured with an appropriate amount of DCPMMs, leaving more DIMM slots for DRAM, as the capacity and bandwidth of DRAM may contribute more to the overall performance. However, determining the optimal number of DCPMMs and threads is out of the scope of this paper.

4.5 Latency

In general, large scale systems suffer from unpredictable high-percentile (tail) latency variations [44]. Tail latency reflects the response time of most operations, but some designs sacrifice tail latency for higher throughput [19]. We collect experimental results under uniform distribution to exclude caching effects and achieve more stable access patterns. We measured single-threaded tail latency, but did not obtain additional insights. Thus, we only present the results of multi-threaded experiments in Figure 7.

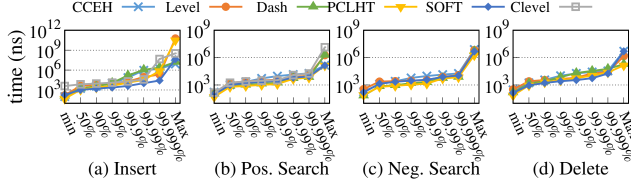


Figure 7: Tail latency at different percentiles under uniform distribution with 32 threads.

Insert. All hash tables suffer from high latency except for SOFT, which employs a lock-free concurrency control mechanism and a hybrid memory architecture. Concretely, the tail latencies of CCEH, Level, Dash, Clevel, PCLHT and SOFT increase to milliseconds (about 1.6ms, 0.2ms, 2.3ms, 46ms, 0.7ms and 0.02ms respectively) at 99.999 percentile. In the worst case, their latencies are about 8ms, 58751ms (resizing triggered), 16ms, 325ms, 31134ms (resizing triggered), and 42ms respectively. Meanwhile, the tail latencies of CCEH and Dash surpass that of others from 99.9 percentile, because of structural modifications (e.g., segment splitting).

Search. The tail latency of search depends more on concurrency control mechanisms. Thus, CCEH has the highest tail latency because it uses pessimistic locking. But for negative search, it remains almost fixed, since negative search needs to traverse all the target buckets to make sure a specific item does not exist. Concretely, the tail latencies for positive/negative search of CCEH, Level, Dash, Clevel, PCLHT and SOFT are in the scale of several microseconds (about 18/18µs, 10/12µs, 7/7µs, 18/60µs, 6/7µs, and 9/10µs respectively) at 99.999 percentile.

Delete. The delete operation may result in multiple reads. Therefore, the latency for the delete operation is tightly bound to search overhead. Notably, SOFT exhibits the lowest latency. This is because SOFT destroys the target node after deleting the records, making its chaining-based buckets shorter. The delete latencies of CCEH, Level, Dash, Clevel, PCLHT and SOFT are tens of microseconds (about 34µs, 26µs, 58µs, 67µs, 23µs and 18µs respectively) at 99.999 percentile.

4.6 Resizing

Resizing is indispensable for hash tables that demand dynamic capacity expanding. We perform experiments on resizing using the insert workload described in Section 4.1. The results are shown in Figure 8. We can see that the resizing overhead should not be overlooked. CCEH consists of two operations to expand its capacity. One is segment splitting that has a constant cost of approximately 37.9μs and does not depend on the size of the hash table. The other is directory doubling, whose overhead scales with the table size.

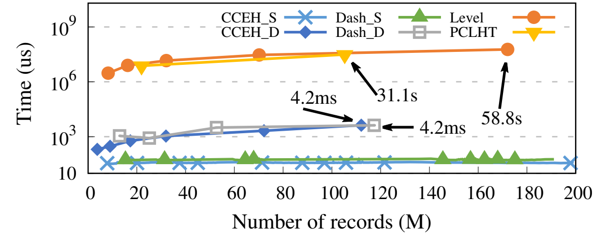


Figure 8: Resizing time (μs). The suffixes "_S" and "_D" indicate segment splitting and directory doubling.

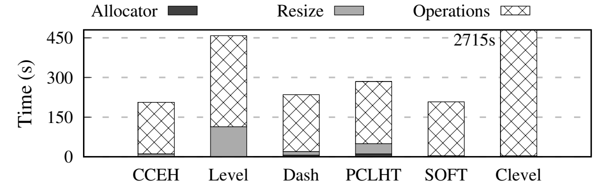


Figure 9: Resizing time and total execution time.

It takes 4.2ms to double the size of the directory when we insert 112M items to trigger this operation. The resizing overhead of Level hashing and PCLHT is much higher, and also increases with the table size. The resizing cost of Dash is sightly higher than CCEH's for both segment splitting and directory doubling (see the curves marked with suffixes "_S" and "_D").

CCEH achieves the smallest resizing overhead. The reason is that CCEH only splits the segment one at a time, and the items copied to the new segment remain intact in the old segment (*lazy deletion*). The structure of Dash is analogous to CCEH. The difference is that Dash maintains 32 bytes metadata in each bucket and items moved to the new segment are deleted in the old segment, which implies that it needs to write more data to PM during resizing. Hence, the segment splitting in Dash is more time consuming (55.6μs on average) than that in CCEH, while the cost of doubling the directory is similar between them. Dash takes 4.2ms to expand the directory at the point when we insert 117M items. Although the structure of Level hashing allows it to reduce the number of rehashes during resizing, its overhead is still higher than that of dynamic hash tables. The resizing operation in Level hashing can be divided into 3 steps: (1) Create a new table; (2) Lock the whole table and move the items of the bottom level to the new table; (3) Free the bottom level table. The buckets of the bottom level are used to store conflicting items from the top level. After resizing, the new bottom level may be too full to store any items. Therefore, when a collision happens in the new top level, it has to resize again. Worse, in the process of resizing, Level hashing needs to delete the items of the original bottom level to maintain consistency, causing extra writes to PM [46, 72]. The resizing strategy in Level hashing leads to high overhead. Concretely, its resizing operation takes over 58.8s at the point when we insert 172M items. Similarly, PCLHT needs to rehash the whole table in resizing. It locks the entire hash table and creates a new table with doubled capacity, then rehashes all items to the new one. At last, the old table is freed, instead of deleting the items in the old table as Level hashing does. As a result, PCLHT achieves lower resizing cost than Level hashing, but it is

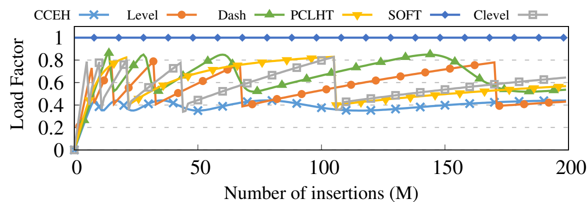


Figure 10: Load factor of different hash tables with respect to the number of items inserted.

still comparatively high. For example, it takes 31.1s to resize the table when 105M items are inserted.

In addition, we compare the resizing time and the total execution time in Figure 9. Level hashing spends a significant amount of time (24.8%) on resizing, and this ratio is 13.6% for PCLHT. CCEH and Dash have similar resizing cost (5.0% vs 5.8%). Since Clevel performs resizing using background threads that run in parallel with query processing threads, its cost of resizing is effectively hidden.

4.7 Load Factor

Load factor is a key metric for hash tables, and it is defined as the number of occupied slots in the hash table divided by the total number of slots. We measure the load factor using the insert workload introduced in Section 4.1, and Figure 10 depicts the results. CCEH has the lowest load factor since it employs a short probing distance by default to solve hash collisions until a segment splitting is triggered. Because its default probing distance is 4 cachelines that can contain 4 buckets or 16 slots, a hash collision may cause premature segment splitting even other buckets in the segment have free slots. This is the reason why CCEH's load factor can only reach 44% under the default probing distance. When the probing distance is set to 64 (256 slots), the load factor can be increased up to 92% [46]. However, a long probing distance would deteriorate its performance.

Dash, Level hashing, Clevel, and PCLHT achieves higher load factor through dedicated optimizations, and the maximum load factors for them are 86%, 79%, 83%, and 83% respectively. Dash solves the issue of premature segment splitting in CCEH using techniques such as stashing, balanced insert, and displacement. Level hashing and Clevel enable each key to have two locations to choose from (similar to cuckoo hashing) before resizing, which improves load factor. The chaining-based PCLHT makes it more flexible to accommodate conflicting items than the open addressing design. Notably, the load factor of CCEH and Dash do not drop significantly (halved) during resizing like PCLHT and Level hashing, since they only split one segment to expand the table. The load factor of SOFT is 100%, because it creates and destroys nodes on demand, and no free space like free slots or buckets is maintained.

Overall, the essential strategy to optimize the load factor in all hash tables but SOFT is to probe more standby buckets. However, this contradicts with the goal of achieving high throughput. On the contrary, we cannot blindly try to improve performance at the cost of maintaining a large amount of unutilized memory.

4.8 Memory Utilization

Hashing indexes may intend to sacrifice memory space for achieving certain design goals. For example, it is common to maintain extra metadata to improve query performance, and to make buckets cacheline-aligned (with unused bits) to optimize cache access. Here, we measure the memory utilization of hash tables, and the results are shown in Table 2. The utilization is defined as the ratio between the actual memory consumption of key-value pairs and the total memory allocated. The memory utilization of SOFT is quite low (about 22%) as it maintains an extra copy of data in DRAM. But if we consider PM storage solely, SOFT reaches nearly 100% utilization because only the actual data plus 3 validity bits are stored in PM. Level hashing's maximum memory utilization is about 52.6% because it includes 13 unused bytes in each bucket to align the buckets to a cacheline. The original implementation of Level hashing has a higher memory utilization because its bucket is more space efficient (4 key-value pairs each consisting of a 16-byte key and a 15-byte value, and 1-byte token), and the bucket is aligned to 2 cachelines. For CCEH, the pointers stored in the directory are the only extra space consumption, but its memory utilization is quite low (only 44.0%) because of its linear probing design that result in low load factor.

Table 2: Comparison of the memory utilization.

	CCEH	Level	Dash	PCLHT	SOFT	Clevel
Utilization(%)	43.9	52.6	74.6	62.2	22.2	55.3

Although each bucket (256 bytes) of Dash has 32 bytes metadata, the memory utilization reaches 74.6% due to its special optimizations to improve load factor. The memory utilization of PCLHT is 62.2%. Each bucket (64 bytes) in PCLHT contains an 8-byte word for concurrency control and a `next` pointer in addition to three key-value pairs. Clevel needs to store pointers that point to key-value pairs, and its memory utilization is 55.3%.

4.9 Recovery

Recovery is necessary for table rebuilding and inconsistency cleaning when a system crash or a power failure occurs. The original PCLHT lacks recovery implementation, so we implement its recovery procedure according to the mechanisms adopted by static hashing schemes. The recovery time is measured as follows. First, a number of key-value pairs are loaded. Second, the process is killed manually. Third, we restart the process, and record the time duration between the point at which the process starts and the point the process is ready to handle incoming requests. Thus, the recovery time consists of two components. One is the time needed to start a process, and it is system specific and normally constant (around 30ms on our test machine). The other is the time required to restore hash specific structures.

Table 3: Comparison of the recovery time (milliseconds).

Scale	CCEH	Level	Dash	PCLHT	SOFT	Clevel
50M	44.2	30.0	30.1	30.0	12981.6	30.0
100M	60.7	30.0	30.1	30.0	31549.4	30.0
150M	72.6	30.0	30.1	30.0	55779.9	30.0
200M	94.9	30.0	30.1	30.0	84786.2	30.0

The results are shown in Table 3. Level hashing, Clevel, and PCLHT exhibit constant recovery time (almost negligible if the process starting time is excluded), because they only need to check

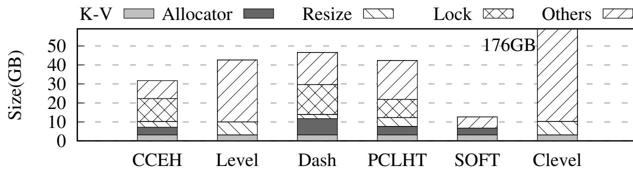


Figure 11: Breakdown of data written to PM.

if there is a unfinished resizing operation. In particular, PCLHT simply concerns if a new table was created but failed to be properly initialized. If it is the case, the new table is deallocated and the resizing operation is rerun. Otherwise, the process can serve requests immediately. Level hashing examines if the pointer that points to the new top level is `null` (indicating an incomplete resizing). It deals with inconsistency similar to PCLHT. For Clevel, besides checking the consistency of rehashing, it also needs to check memory leaks in order to release unused memory.

CCEH's recovery time scales with data size, since it needs to traverse the directory for inconsistency checking. Dash behaves similarly, but its overhead is amortized over segment accesses after recovering. Therefore, the recovery time of Dash is lower than that of CCEH. SOFT maintains structural information in DRAM. In addition to a full traversal of data on PM, it is required for SOFT to reconstruct the structure in DRAM. As a result, its recovery time is 3 orders of magnitude larger than others.

4.10 Breakdown of Data Written to PM

Recall that when inserting items, the average size of data written to PM is much larger than the actual size of key-value pairs as shown in Figure 3(a). In order to understand this phenomenon, we conduct the following experiments. We breakdown the total amount of data written to PM (denoted as `total data`) when inserting 2M key-value pairs (3.2GB), into five parts (except Clevel) that include: (1) `allocator`, data written due to dynamic memory allocation/deallocation; (2) `resize`, data written during the resizing operation; (3) `K-V`, the actual size of key-value pairs; (4) `lock`, extra data incurred by `lock`/`unlock` primitives; and (5) `others`. We use the PCM tool to measure the size of `allocator` and `resize` directly, and collect the results for `lock` through source code instrumentation because it is too time consuming to measure 2M invocations of locks using the API provided by PCM. Results are shown in Figure 11.

Allocator. Both Dash and CCEH need to allocate memory for a new segment when segment splitting happens. Dash causes ~8.5GB (18%) extra data in memory allocation, while it is only ~3.9GB (13%) for CCEH. The reason for this discrepancy is that they use different memory allocators. Dash depends on `libpmemobj` that needs to create logs for crash-safety, while CCEH uses `libvmem` that has not crash-safety guarantee. PCLHT and SOFT produce ~4.4GB (11%) and ~3.5GB (28%) of data in allocators. For Level hashing, the data volume is ~41 MB, because it triggers memory allocations only when the resizing is required. Clevel uses the API `make_persistent_object` of PMDK to allocate memory, which is coupled with data copy. Hence, we cannot accurately measure how much data is generated from `allocator` without modifying the source code of PMDK, so we classify it to the category of `others`.

Resize. In this experiment, Level hashing causes five full-table resizing operations, resulting in ~6.8GB (16%) of data written to the PM. While PCLHT triggers two full-table resizing operations, producing ~4.6GB (11%) of data. Both CCEH and Dash expand capacity at the granularity of segment. Therefore, they induce less resizing data than PCLHT. CCEH triggers structural changes more frequently, so it has a larger percentage of resizing data than DASH (i.e., 9% vs 5% or 3.0GB vs 2.2GB).

Lock. From Figure 11, we can observe that for the three lock-based PM hash tables (i.e., CCEH, Dash, and PCLHT), the extra data induced by locking is tremendous in size because of frequent invoking of locks and contention among threads, reaching 12.5GB (32%) on average.

Others. Except Clevel, other hash tables are log-free. Dash uses log only when doubling the directory, and the data generated by logging can be ignored. For CCEH, Dash, PCLHT, and SOFT, the data volume belonging to `others` (accounting for 40% on average) is approximately equal to that incurred by write amplification, and it is 4x of that in `K-V`. This is consistent with the fact that 16-byte key-value pairs are flushed to PM at the granularity of 64 bytes. For Level hashing, `others` consists of data produced by write amplification and moving items between buckets. For Clevel, 92% of data is in this category. Reasons are three-fold. (1) It needs to allocate space for each item to be inserted; (2) It stores pointers in the hash table to support variable-length items; (3) The function `make_persistent_object` will create a log entry for each memory allocation and data copy.

The above analysis demonstrates that key-value pairs only accounts for a small part of all data written to PM, and writes associated with memory allocation, synchronization primitives, and hash table specific features (like resizing) could be unexpectedly high.

4.11 Issues Related to PM Hardware

In this subsection, we explore three performance issues related to PM hardware. First, **when operating on the elements of hash tables, will there be contention for DIMMs?** Previous work [68] found that *accesses across interleaved DIMMs may cause contention for particular DIMMs*. We use the PCM tool [13] to observe the bandwidth of each channel every 15ms when running the workload. We found that the bandwidth of each channel is always approximately the same during execution. Therefore, we conclude that the contention for DIMMs does not happen for hash tables.

Note that, none of the hash tables evaluated in this paper can approach the upper limit of DCPMM write bandwidth (13GB/s [68]). Thus, the second question is **how small random PM writes restrict the performance of hash tables.** We designed a simplified model to explore why hash tables have such a low PM bandwidth utilization. In order to eliminate the interference from operations like resizing, synchronization, and hash value computation, we use a large array to simulate a hash table that has no such operations. Items of size 16 bytes are inserted into a random position of the array using 32 threads. We also evaluate the effect of certain instructions on the overall performance by enabling only one or both of flush and fence instructions, or disabling both, or using non-temporal stores (*NTstore*). Figure 12 displays the throughput of (the bars in gray) and bandwidth (the curves marked with squares). We can see that the throughput cannot exceed 31 Mops/s when inserting items with size of 16 bytes, even when flush and fence

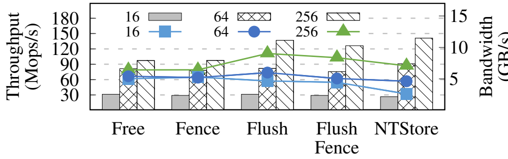


Figure 12: The impact of different instructions on the throughput of the simplified hash table under 32 threads.

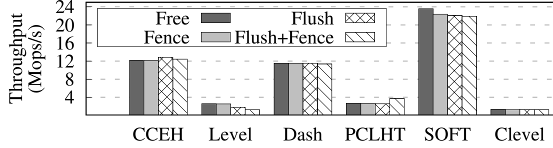


Figure 13: Throughput with different instructions under 32 threads.

instructions are both disabled. Thus, we can conclude that the low throughput and bandwidth utilization are mainly due to the small random writes. Unfortunately, none of the existing PM hash tables, even those with specific optimizations like cacheline alignment, can solve this problem properly. A feasible solution is to convert small random writes into large sequential ones. For example, we can group random writes into a DRAM buffer that can then be written to PM sequentially. We use the simplified model to evaluate the efficiency of this method. When items are packed into 64-byte buffers, under the same bandwidth, the throughput increases to 91Mops/s, and reaches 141Mops/s when we use 256-byte buffers. From Figure 12, we can also observe that the performance is more sensitive to flush and fence instructions for large write, and *NTstore* achieves the highest throughput for the 256-byte write.

The last question is that **how much effect flush and fence instructions have on the performance of hash tables.** To answer this question, we conduct an experiment, in which we intentionally disable one or both of the flush and fence instruction for all hash tables. The results are shown in Figure 13. We can observe that these instructions have little impact on the throughput for all hash tables, which is consistent with the 16-byte case in Figure 12. This is because write operations in PM hash tables (using customized designs to ensure consistency) merely involve a few cacheline flushes and memory fences, as compared to the data structures that rely on software transactional memory [20].

4.12 Impact of NUMA

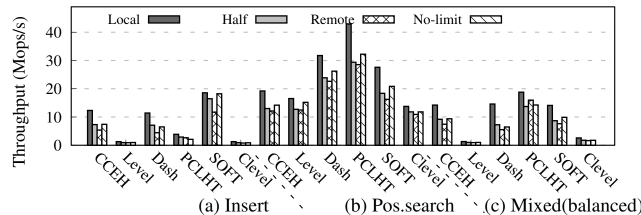


Figure 14: The impact of NUMA architecture on throughput.

In this section, we investigate the impact of NUMA on throughput using workloads with uniform distribution. We use both CPUs (each with 16 cores) on our test machine, one denoted as local CPU and the other as remote CPU. All DRAMs and DCPMMs are installed on one CPU (local). We launch 32 threads and pin them to CPU cores with different configurations: (1) *Local* indicates all the 32 threads are pinned to the local CPU. (2) *Half* means 16 threads are pinned to the local CPU and the remaining 16 to the remote CPU. (3) *Remote* means all the 32 threads are pinned to the remote CPU. (4) *No-limit* indicates threads are managed by the default OS scheduler.

The results shown in Figure 14 reveal that accessing the remote node hurts performance severely in most cases. For the insert workload, the throughput of *Remote* drops significantly compared to *Local*. For example, Dash's throughput is decreased by 2.6x, followed by CCEH (2.3x). The reduction is 1.4x for Level hashing and PCLHT. The throughput of Level hashing and Clevel are less affected by NUMA. From Figure 14(b), we can observe that NUMA has less impact on performance for reads than writes (comparing *Half* with *Remote* for search and insert). The reason is as follows. The ratio of read latency between local and remote Optane DCPMM is 1.20x for random access. For write, the remote access latency of Optane DCPMM is 1.68x higher compared to local [68]. And, hash tables exhibit fairly random access patterns. Mixed workloads shown in Figure 14(c) exhibit similar trends with search, because the delete operation produces multiple reads and one write on PM. In all cases, the throughput under *No-limit* is close to (even better than) *Half*, because the OS may be able to schedule threads optimally.

Although our evaluation demonstrates the negative impact of NUMA on performance, running more threads on remote node can still be properly leveraged to improve performance, depending on the utilization of PM bandwidth. As discussed in Section 4.3, for some workloads, the throughput of SOFT and PCLHT scales with more remote threads, while CCEH and Dash can exhaust PM bandwidth with only local threads.

5 DISCUSSIONS

In this section, we summarize the observations and insights made from our evaluation and analysis.

1. Random small write is the primary factor that fundamentally restricts the performance of PM hash tables. Random small write is the inherent access pattern in hash tables. Thus, even with specific optimizations to address this issue, such as reducing the number of random writes, the performance of existing PM hash tables is fundamentally constrained. We argue that new hash table designs are desired to mitigate the detrimental effect of random small write and achieve higher performance.

2. The impact of cache flush and memory fence instructions on performance is marginal. Contrary to common observations made in PM data structure designs that employ software transactional memory, the performance of PM hash tables is not constrained by cache flushing and memory fence instructions, due to the limited number of these instructions in individual operations. Therefore, optimizations should focus on other aspects in designing PM hash tables.

3. Fingerprinting can accelerate negative queries significantly. Cache-resident fingerprints can speedup negative search considerably, since unnecessary accesses to PM can be filtered. This is especially meaningful for PM given its asymmetric read/write

performance (i.e., end-to-end write latency is often lower than reads).

4. Resizing should not discourage normal operations. Full-table rehashing is harmful to concurrency, leading to low throughput and unexpected latency. Dynamic extendible schemes (Dash and CCEH) and lock-free resizing (Clevel) are good paradigms to alleviate the overhead of resizing.

5. Beware of the significance of write amplifications caused by memory allocators and synchronization primitives. Besides write amplifications incurred by small random write that is inherent for hash tables, memory allocators and synchronization primitives also cause severe write amplifications, which may be far beyond our expectations as quantified in this paper. Write amplification not only degrades performance, but also hurts the endurance of DCPMM. In the future, we plan to explore the design of wear-aware memory allocator for PM systems and PM-friendly locks.

6. Concerns about hybrid memory designs. Maintaining structural metadata in DRAM and key-value pairs in PM can substantially reduce PM reads/writes and access latency. However, new techniques should be considered in the future to reduce the *recovery cost* for hybrid memory designs.

7. PM bandwidth is a scarce resource, but the performance of hash tables does not necessarily scale with more DCPMMs. For some hash tables, when more DCPMMs are installed in a system, their performance scales almost linearly. While for others, the increase in throughput is small or barely noticeable even if the system is fully populated with DCPMMs. More efforts are needed to figure out this scalability issue for PM hash tables. Furthermore, system designers who want to use an existing hash table would be aware the importance of striking a balance between DCPMM and DRAM resources, in order to achieve better performance.

8. Micro-architectural characteristics of PM matter. We have shown that designers of hash tables should be aware of the existence and importance of XPBuffer. Moreover, we expect to see more characteristics study to reveal performance-critical micro-architectural components, such as the WPQ size, read-modify-write buffer, and address indirection translation buffer as identified in [62] to optimize the high-level design of PM data structures like hash table.

6 RELATED WORK

DRAM-based hash tables have been studied for decades. A wide range of high performance hash schemes for single-processor or multi-processor systems have been proposed [3, 7, 11, 21, 34, 35, 42, 51]. Our previous work [8] conducted extensive evaluations of five DRAM-based concurrent hash tables on four representative multi-core platforms under a unified framework.

PM-based index structures. In addition to the hash tables discussed in Section 3, several PM hash tables were recently proposed. NVC-Hashmap [56] is a lock-free hashmap that supports unordered dictionaries and delta indices for in-memory databases. PFHT [12] is a write-optimized hash table using a stash bucket to improve load factor. To avoid extra PM writes from update operations, Path hashing [71] organizes its buckets as an inverted complete binary tree logically. However, this structure causes low lookup performance as the lookup operation depends on the height of the tree. Level hashing [72] solves this problem via a sharing-based two-level structure. PM-based indexing tree is also a well-studied topic, and several proposals have been made [2, 5, 49, 69, 70]. Lersch et al. developed a benchmarking framework PiBench [33], and performed comprehensive evaluations on four B^+-Trees using Intel Optane DC Persistent Memory.

Efforts to reduce consistency and durability overhead and programming difficulty. Transaction memory (TM) interacting with PM is commonly used to guarantee failure-atomicity and durability [4, 25, 29, 60]. However, PM indexes with TM still suffer from the overhead of flushing and logging [20, 57], and several attempts to alleviate this have been proposed. TIMESTONE [30] uses a hybrid logging technique, *TOC logging*, to guarantee crash consistency with low write amplification and minimal memory footprint. Minimally Ordered Durable (MOD) library [20] provides STL-like persistent data structure interfaces to reduce the efforts required to develop PM applications. Similar to MOD, Pronto [40] can reduce the programming efforts required to add persistence to volatile data structures, using asynchronous semantic logging.

PM indexes in data-intensive systems. Concurrent data structures are essential components of modern data-intensive systems, such as database systems [45, 48], key-value stores [14, 16, 18, 28, 32, 36, 41, 50, 52, 53, 67], and file systems [47, 55]. However, the performance of these systems is limited by the size of workloads, especially when a systems need to recover and warm up their state after a restart [17]. Thus, several storage systems [38, 65, 66], which take full advantage of the features of hybrid memory systems consisting of DRAM and PM, were proposed. HiKV [65] is a persistent key-value store with a hybrid index (hash table in PM and B^+-Tree in DRAM). It inherits the efficiency of point query from hash table and the range query from the B^+-Tree. Another problem of PM-based key-value store is that the small-sized access pattern in key-value stores does not match with the persistence granularity in PM, leaving the PM bandwidth underutilized [6, 33]. FlatStore [6] is a PM-based key-value storage engine designed to alleviate this issue, using CCEH [46] as its index.

7 CONCLUSION

This paper performs a comprehensive evaluation of hash tables designed for PM based on the recently released Intel Optane DC Persistent Memory. We extend the framework *PiBench* with more hash table specific functionalities, and evaluate six state-of-the-art hash indexes elaborately, considering not only the common metrics, but also hardware-related properties such as PM bandwidth, flush/fence instructions, and NUMA architecture. We identify key insights for the design PM-based hash tables. We hope our experimental results would be valuable for researchers and practitioners to develop more efficient and scalable PM-based hash tables.

ACKNOWLEDGMENTS

This research was supported by the National Key Research and Development Program of China (No. 2018YFB1003502), and by the National Science Foundation of China under Grants 61772183 and 61972137. We thank the anonymous reviewers for their valuable suggestions in improving this paper.

REFERENCES

[1] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman, et al. 2019. Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro* 39, 2 (2019), 29–36.

[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565. https://doi.org/10.1145/3164135.3164147

[3] Alex D. Breslow, Dong Ping Zhang, Joseph L. Greathouse, Nuwan Jayasena, and Dean M. Tullsen. 2016. Horton Tables: Fast Hash Tables for in-Memory Data-Intensive Computing. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX, Denver, CO, USA, 281–294.

[4] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '14)*. ACM, Portland, Oregon, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[5] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797. https://doi.org/10.14778/2752939.2752947

[6] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. ACM, Lausanne, Switzerland, 1077–1091. https://doi.org/10.1145/3373376.3378515

[7] Zhiwen Chen, Xin He, Jianhua Sun, and Hao Chen. 2018. Have Your Cake and Eat It (Too): A Concurrent Hash Table with Hardware Transactions. *Int. J. Parallel Program.* 46, 4 (2018), 699–709. https://doi.org/10.1007/s10766-017-0529-7

[8] Zhiwen Chen, Xin He, Jianhua Sun, Hao Chen, and Ligang He. 2018. Concurrent hash tables on multicore machines: Comparison, evaluation and implications. *Future Generation Computer Systems* 82 (2018), 127 – 141.

[9] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 799–812. https://www.usenix.org/conference/atc20/presentation/chen

[10] Dave Chinner. 2015. xfs: DAX support. Retrieved January 14, 2021 from https://lwn.net/Articles/635514/

[11] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, Istanbul, Turkey, 631–644. https://doi.org/10.1145/2694344.2694359

[12] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G Khatib, and Cristian Ungureanu. 2016. Revisiting hash table design for phase change memory. *ACM SIGOPS Operating Systems Review* 49, 2 (2016), 18–26.

[13] Intel Corporation et al. 2019. Processor Counter Monitor. Retrieved January 14, 2021 from https://github.com/opcm/pcm/

[14] Facebook. 2020. RocksDB. Retrieved January 14, 2021 from https://rocksdb.org

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

[16] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. USENIX, Lombard, IL, 371–384.

[17] Aakash Goel, Bhuwan Chopra, Ciprian Gerea, Dhruv Mátáni, Josh Metzler, Fahim Ul Haq, and Janet Wiener. 2014. Fast Database Restarts at Facebook. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. ACM, Snowbird, Utah, USA, 541–549. https://doi.org/10.1145/2588555.2595642

[18] Google. 2020. LevelDB. Retrieved January 14, 2021 from https://leveldb.org

[19] Xiangpeng Hao, Tianzheng Wang, Lucas Lersch, and Ismail Oukid. 2020. Interactive Benchmarking of Persistent Memory Indexes. Retrieved January 14, 2021 from http://pibench.org/

[20] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2020. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. ACM, Lausanne, Switzerland, 775–788. https://doi.org/10.1145/3373376.3378472

[21] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing (DISC '08)*. Springer-Verlag, Arcachon, France, 350–364. https://doi.org/10.1007/978-3-540-87779-0_24

[22] M Hosomi, H Yamagishi, T Yamamoto, K Bessho, Y Higo, K Yamane, H Yamada, M Shoji, H Hachino, C Fukumoto, et al. 2005. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. In *IEEE International Electron Devices Meeting, 2005*. IEEE, 459–462.

[23] Intel. 2018. Intel Optane DC Persistent Memory Operating Modes Explained. Retrieved January 14, 2021 from https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes/

[24] Intel. 2020. libvmem of Persistent Memory Development Kit. Retrieved January 14, 2021 from https://pmem.io/vmem/libvmem/

[25] Intel. 2020. Persistent Memory Development Kit. Retrieved January 14, 2021 from https://pmem.io/

[26] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, Atlanta, Georgia, USA, 427–442. https://doi.org/10.1145/2872362.2872410

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

[28] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-Ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX, Boston, MA, USA, 191–204.

[29] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, Atlanta, Georgia, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[30] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. ACM, Lausanne, Switzerland, 335–349. https://doi.org/10.1145/3373376.3378483

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

[32] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: The Design and Implementation of a Fast Persistent Key-Value Store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, Huntsville, Ontario, Canada, 447–461. https://doi.org/10.1145/3341301.3359628

[33] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proceedings of the VLDB Endowment* 13, 4 (2019), 574–587. https://doi.org/10.14778/3372716.3372728

[34] Dagang Li, Rong Du, Ziheng Liu, Tong Yang, and Bin Cui. 2019. Multi-copy Cuckoo Hashing. In *In Proceding of IEEE 35th International Conference on Data Engineering (ICDE'19)*. 1226–1237.

[35] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, Amsterdam, The Netherlands, Article 27, 14 pages. https://doi.org/10.1145/2592798.2592820

[36] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast in-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX, Seattle, WA, 429–444.

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

[38] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'17)*. USENIX, Santa Clara, CA, 4.

[39] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, Belgrade, Serbia, 499–512. https://doi.org/10.1145/3064176.3064215

[40] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. ACM, Lausanne, Switzerland, 789–806. https://doi.org/10.1145/3373376.3378456

[41] Memcached. 2019. Memcached. Retrieved January 14, 2021 from https://memcached.org
[42] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. CPHASH: A Cache-Partitioned Hash Table. *SIGPLAN Not.* 47, 8 (2012), 319–320. https://doi.org/10.1145/2370036.2145874
[43] Micron. 2015. 3D-X-Point Technology. Retrieved January 14, 2021 from https://www.micron.com/products/advanced-solutions/3d-xpoint-technology
[44] Pulkit A. Misra, María F. Borge, Íñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. 2019. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, Dresden, Germany, Article 17, 15 pages. https://doi.org/10.1145/3302424.3303973
[45] MongoDB. 2020. MongoDB. Retrieved January 14, 2021 from https://www.mongodb.com
[46] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies*. USENIX, Boston, MA, 31–44. https://www.usenix.org/conference/fast19/presentation/nam
[47] Oracle. 2019. Architectural Overview of the Oracle ZFS Storage Appliance. Retrieved January 14, 2021 from https://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/o14-001-architecture-overview-zfsa-2099942.pdf
[48] Oracle. 2020. MySQL. Retrieved January 14, 2021 from https://www.mysql.com
[49] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, San Francisco, California, USA, 371–386. https://doi.org/10.1145/2882903.2915251
[50] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2010. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 92–105. https://doi.org/10.1145/1713254.1713276
[51] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *J. Algorithms* 51, 2 (2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002
[52] Swapnil Patil and Garth Gibson. 2011. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX, San Jose, California, 177–190.
[53] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. 2017. Fast scans on key-value stores. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1526–1537.
[54] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. 2017. Programming for Non-Volatile Main Memory Is Hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys '17)*. ACM, Mumbai, India, Article 13, 8 pages. https://doi.org/10.1145/3124680.3124729
[55] Frank Schmuck and Roger Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. USENIX, Monterey, CA, 19–es.
[56] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics (IMDM'15)*. ACM, Kohala Coast, HI, USA, Article 4, 8 pages. https://doi.org/10.1145/2803140.2803144
[57] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, Toronto, ON, Canada, 175–186. https://doi.org/10.1145/3079856.3080240
[58] Steve Scargall. 2020. *Programming Persistent Memory.* Apress. 186–189 pages.
[59] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. 2008. The missing memristor found. *Nature* 453, 7191 (2008), 80–83.
[60] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, Newport Beach, California, USA, 91–104. https://doi.org/10.1145/1950365.1950379
[61] Tianzheng Wang and Ryan Johnson. 2014. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment* 7, 10 (2014), 865–876. https://doi.org/10.14778/2732951.2732960
[62] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*.
[63] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An Early Evaluation of Intel's Optane DC Persistent Memory Module and Its Impact on High-Performance Scientific Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*. Association for Computing Machinery, Denver, Colorado, Article 76, 19 pages. https://doi.org/10.1145/3295500.3356159
[64] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. 2010. Phase change memory. *Proc. IEEE* 98, 12 (2010), 2201–2227.
[65] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX, Santa Clara, CA, USA, 349–362.
[66] Jian Xu and Steven Swanson. 2016. NOVA: A Log-Structured File System for Hybrid Volatile/Non-Volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*. USENIX, Santa Clara, CA, 323–338.
[67] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. Bluecache: A Scalable Distributed Flash-Based Key-Value Store. *Proceedings of the VLDB Endowment* 10, 4 (2016), 301–312. https://doi.org/10.14778/3025111.3025113
[68] Jianhua Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*.
[69] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX, Santa Clara, CA, 167–181.
[70] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *Proceedings of the VLDB Endowment* 13, 4 (2019), 421–434. https://doi.org/10.14778/3372716.3372717
[71] Pengfei Zuo and Yu Hua. 2017. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 5 (2017), 985–998.
[72] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX, Carlsbad, CA, USA, 461–476.
[73] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 128 (2019), 26 pages. https://doi.org/10.1145/3360554