Lock-Free High-performance Hashing for Persistent Memory via PM-aware Holistic Optimization

ZHANGYU CHEN, YU HUA, LUOCHANGQI DING, BO DING, and PENGFEI ZUO,
WNLO, School of Computer Science and Technology, Huazhong University of Science and Technology, China
XUE LIU, McGill University, Canada

Persistent memory (PM) provides large-scale non-volatile memory (NVM) with DRAM-comparable performance. The non-volatility and other unique characteristics of PM architecture bring new opportunities and challenges for the efficient storage system design. For example, some recent crash-consistent and write-friendly hashing schemes are proposed to provide fast queries for PM systems. However, existing PM hashing indexes suffer from the concurrency bottleneck due to the blocking resizing and expensive lock-based concurrency control for queries. Moreover, the lack of PM awareness and systematical design further increases the query latency. To address the concurrency bottleneck of lock contention in PM hashing, we propose clevel hashing, a lock-free concurrent level hashing scheme that provides non-blocking resizing via background threads and lock-free search/insertion/update/deletion using atomic primitives to enable high concurrency for PM hashing. By exploiting the PM characteristics, we present a holistic approach to building clevel hashing for high throughput and low tail latency via the PM-aware index/allocator co-design. The proposed *volatile announcement array* with a helping mechanism coordinates lock-free insertions and guarantees a strong consistency model. Our experiments using real-world YCSB workloads on Intel Optane DC PMM show that clevel hashing, respectively, achieves up to 5.7× and 1.6× higher throughput than state-of-the-art P-CLHT and Dash while guaranteeing low tail latency, e.g., 1.9×–7.2× speedup for the p99 latency with the insert-only workload.

CCS Concepts: • **Hardware** → **Non-volatile memory**; • **Information systems** → **Data structures**; • **Software and its engineering** → **Concurrency control;**

Additional Key Words and Phrases: Lock-free index, hashing, persistent memory, correctness

5

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and Key Laboratory of Information Storage System, Ministry of Education of China.

Extension of Conference Paper: The preliminary version was published in the Proceedings of the USENIX Annual Technical Conference (USENIX ATC), 2020, pages: 799–812, as “Lock-free Concurrent Level Hashing for Persistent Memory” [16]. In this manuscript, we address the performance and correctness problems in existing lock-free design. For the performance problem, by exploiting the PM hardware characteristics, we propose a new PM-aware index/allocator co-design and a new metadata structure to substantially improve the throughput while guaranteeing low tail latency. For the correctness problem, we propose the *volatile announcement array* with a helping mechanism to ensure strong consistency, thus improving the applicability. We also discuss and evaluate the impact of the eADR hardware feature on our system design.

Authors’ addresses: Z. Chen, Y. Hua (corresponding author), L. Ding, B. Ding, and P. Zuo, Wuhan National Laboratory for Optoelectronics (WNLO), School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China, 430074; emails: {chenzy, csyhua, dinglcq, boding, pfzuo}@hust.edu.cn; X. Liu, School of Computer Science, McGill University, Montreal, Canada, H3A 0E9; email: xueliu@cs.mcgill.ca.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.

1544-3566/2022/11-ART5 $15.00
https://doi.org/10.1145/3561651

ACM Reference format:
Zhangyu Chen, Yu Hua, Luochangqi Ding, Bo Ding, Pengfei Zuo, and Xue Liu. 2022. Lock-Free High-performance Hashing for Persistent Memory via PM-aware Holistic Optimization. *ACM Trans. Archit. Code Optim.* 20, 1, Article 5 (November 2022), 26 pages.
https://doi.org/10.1145/3561651

1 INTRODUCTION

Persistent memory (PM) is a memory technology that provides TB-scale capacity, durable storage substrate, and DRAM-comparable performance. These salient properties come from attaching **non-volatile memory (NVM)** devices, e.g., Intel Optane DC PMM [4], PCM [50], and ReRAM [49], to the memory bus. Recent studies evaluating Intel Optane DC PMM [4]—a commercial PM product, have shown that PM is promising to improve many storage systems' performance [51, 52]. Moreover, the non-volatility of PM enables instant recovery, providing new opportunities for storage systems requiring high **quality-of-service (QoS)** with low *tail latency* (the high percentiles of latencies) [51]. However, PM data need to be maintained in a consistent state and recoverable after crashes, called *crash-consistency* requirement [36, 56]. Moreover, both research prototypes [49, 50] and industrial products [4] of PM exhibit asymmetric read and write performance. For example, the maximal read bandwidth of Optane DC PMM is about 3× than that of write for a single module [32, 51, 52]. The persist operations for crash consistency, e.g., cache line flushes and memory fences, further decrease the PM write performance [25].

To provide fast query services while supporting instant recovery, high-performance PM-based indexes are critical to storage systems. Some crash-consistent tree-based index structures have been proposed for PM [13, 17, 28, 33, 34]. However, the overheads to traverse through pointers in hierarchical tree structures are non-negligible for queries. Unlike tree-based designs, hashing-based indexes provide constant-scale point query performance by using hash functions to locate data in flat space. Hence, hashing-based indexes have been widely used in many storage systems [7, 11]. As building blocks of memory systems, PM hashing schemes need to offer high throughput and low latency while constraining the tail latency.

For the throughput, existing PM hashing schemes focus on crash-consistency guarantee and write optimizations while paying little attention to the blocking resizing, thus exhibiting low throughput and utilization on the many-core architecture for PM (e.g., a single processor supporting Optane DC PMM often has dozens of cores). When there is no vacant position for new items or the *load factor* (interpreted as dividing the number of inserted items by the index capacity) of a hash table approaches a configurable threshold, the table needs to be resized. For traditional hash table resizing, a global lock is required while migrating all items from the old hash table to the new one. For P-CLHT [35], a crash-consistent cache-efficient hash table with separate chaining, the write accesses to *stale buckets*, in which all items residing have been migrated, are blocked until the full-table resizing is completed. The PM-friendly level hashing [56] leverages in-place resizing to reduce the number of buckets to be rehashed to one-third of the traditional scheme, but still suffers from the global lock for resizing. For PM hashing schemes using extendible hashing [38, 42], coarse-grained locks for shared resources significantly increase the latency. For example, **Cacheline-Conscious Extendible Hashing (CCEH)** [42] splits a segment, an array of 1,024 slots by default, to increase the capacity, which requires the writer lock for the whole segment. Moreover, a global writer lock is acquired for the potential directory doubling. By amortizing rehashing over future queries (lazy rehashing), the **`concurrent_hash_map` (CMAP)** in pmemkv [10] avoids global locks. However, deferred rehashing tasks may aggregate to a recursive execution in the critical path of queries, leading to uncontrolled query performance degradation.

For the latency, existing hashing indexes for PM suffer from the lock-based thread synchronization or the correctness issue on lock-free concurrency control. Coarse-grained exclusive locks [38, 42] block other threads, thus increasing the query latency and decreasing the overall throughput. Persist operations (e.g., logging, cache line flushes, and memory fences) in the critical path exacerbate the contention. Fine-grained locks [35, 56] shorten the critical path for one lock but introduce frequent locking and unlocking for multiple shared resources, which is also prone to bugs (Section 2.4.1). Our preliminary work [16] explores the lock-free design for concurrent queries. However, the lock-free design in our prior work causes reading transient items to be deleted (by future updates/deletions) due to the duplicate items inserted by concurrent insertions, called the *reading uncommitted* problem,[1] thus limiting the applicability.

For the tail latency, the result is affected by many system components, but prior PM hashing schemes lack systematic design and optimizations. For example, existing PM hashing indexes leverage the allocator from **Persistent Memory Development Kit (PMDK)** [9], which provides general-purpose PM allocation but at the cost of expensive logging for crash consistency [54]. In our preliminary work [16], up to 32% of the insertion overhead (reported by the Linux `perf` [8] tool with insert-only workloads) comes from the crash-consistent allocation for key-value items using PMDK's allocator. Moreover, due to the unawareness of hardware characteristics, the inefficient use of PM impedes the building of low-latency persistent hashing indexes.

Overall, to build a PM-efficient hashing scheme, we need to address the following challenges:

(1) Performance Degradation during Resizing. Coarse-grained locks during resizing guarantee the thread safety and crash consistency of hash table, but block other threads waiting for the completion of resizing. We need to minimize the performance impact of resizing.

(2) High Contention for Lock-based Concurrency Control. Lock-based techniques are widely used in PM hashing to synchronize concurrent accesses to a bucket but at the cost of poor scalability. A low-contention concurrent and correct hashing scheme for PM is needed for high scalability.

(3) High Tail Latency for Inefficient Consistency Guarantee. For microsecond-scale PM indexes, the latency is sensitive to crash consistency overheads. Building PM hashing schemes with systematic optimizations is important to the tail latency reduction.

We propose *clevel hashing*, a crash-consistent and lock-free persistent hashing scheme that provides high throughput and low latency while guaranteeing strong consistency. Motivated by our level hashing [56], we further explore write-efficient open-addressing techniques to enable write-friendly and memory-efficient properties for PM in the context of concurrency. Different from level hashing, we demonstrate new insights to provide scalable performance and low latency for persistent hashing via lock-free concurrency control and PM-aware systematic designs. In the clevel hashing, we propose a dynamic multi-level structure with asynchronous resizing. A level is dynamically added for table expansion and removed when rehashing of the level is completed. The rehashing of items, from the last level to the first level, is offloaded into background threads, thus never blocking foreground queries. To provide high scalability and low latency, we design lock-free concurrency control for search/insertion/update/deletion. Moreover, our clevel hashing is a holistic index that systematically optimizes tail latency via an index/allocation co-design and enforces a strong consistency model with low overheads for correctness guarantee. Specifically, we have made the following contributions in the clevel hashing scheme:

- **Non-blocking Resizing.** We propose a dynamic multi-level structure that supports asynchronous resizing without blocking other threads. During resizing, background threads

[1] This problem is similar to the *dirty read* problem, which violates the isolation (one of the ACID properties in the database community) of transactions. The difference is that there is no transaction in the clevel hashing design.

migrate the items from the last level to the first level until there are two remaining levels. Therefore, when rehashing is not running, which is prevalent for many workloads, the time complexity for search/insertion/update/deletion is constant-scale.

- **Lock-free Concurrency Control.** We design lock-free algorithms for all queries in the clevel hashing. To ensure the correctness of interleaved resizing and lock-free queries, we define the metadata about levels in use as *context* and propose lightweight context-aware schemes to handle consistency issues, e.g., missing inserted items and update/deletion failures.
- **PM-aware Holistic Optimization.** We exploit PM characteristics and present a holistic approach to reduce tail latency. In particular, a PM-aware index/allocator co-design is proposed for high scalability and PM efficiency. Based on PM characteristics, our proposed index/allocator co-design consists of three novel PM-aware techniques: (1) thread-local lock-free allocation, a PM management mechanism that significantly mitigates the contention for the item allocation and converts random PM writes into faster sequential ones, (2) write-optimized operations, e.g., the insertion of a variable-length item only involves two PM writes in the software stack (accommodating the key-value item and storing the corresponding item pointer), and (3) lightweight crash-consistency guarantee, which leverages the item pointer residing in the multi-level index as the commit flag for the corresponding item allocation/reclamation. The partially allocated key-value items are efficiently identified by selectively searching for unreferenced PM blocks. Hence, our proposed co-design of index/allocator avoids the expensive logging or checksums for PM allocation and reclamation for items. Moreover, we introduce recoverable metadata structures on DRAM, enabling fast metadata accessing, low-overhead consistency guarantee and instant recovery.
- **Durable Linearizability.** We reveal the reading uncommitted problem in the existing lock-free design for PM hashing and propose the ***volatile announcement array (VAA)*** to address the concurrency correctness problem. As a result, our clevel hashing ensures a strong consistency model, i.e., durable linearizability [29].
- **System Implementation.** We have implemented our clevel hashing[2] using PMDK [9] and compared the clevel hashing with state-of-the-art schemes on Intel Optane DC PMM. The results using YCSB workloads show the efficacy and efficiency of the clevel hashing.

2 BACKGROUND

2.1 Persistent Memory (PM)

We introduce the PM-specific hardware characteristics and crash consistency that support the PM-aware optimizations of our clevel hashing.

2.1.1 Hardware Characteristics. PM technologies, e.g., PCM [50], ReRAM [49], and commercially available Intel Optane DC PMM [4], exhibit asymmetric read/write performance. In particular, many NVM media suffer from high write latency and limited endurance due to the high overheads for programming NVM cells [44]. A recent study [52] on Intel Optane DC PMM demonstrates that the maximal concurrent read bandwidth (6.6 GB/s) is 2.9× of that for write (2.3 GB/s) with a single DIMM. In addition, the sequential write of Intel Optane DC PMM shows higher performance than the small random write [12, 30, 52]. As shown in publicly available documentation [2, 12], the endurance of current Intel Optane DC PMM is enough for a few years, e.g., 103 PBW (petabytes written) for a 512-GB Intel Optane PM 200 Series DIMM under 100% 64-byte write.

[2]https://github.com/chenzhangyu/Clevel-Hashing.

2.1.2 Crash Consistency. To enable instant recovery, PM data need to ensure crash consistency, which requires the program data are consistent or can be recovered after crashes. Due to the volatile CPU cache in common computer systems, e.g., Intel ×86 architecture with **Asynchronous DRAM Refresh (ADR)** [45], it is nontrivial to efficiently guarantee crash consistency for byte-addressable PM. Specifically, for store instructions, when data size is larger than 8 bytes (the typical maximal atomic write size), system failures during sequential write instructions lead to potential partial updates. Moreover, the persist order of data in write-back caches may be different from the issue order of store instructions, thus demanding memory barriers to enforce the consistency. Hence, recent CPUs provide instructions for cache line flushes (e.g., `clflush`, `clflushopt`, and `clwb`) and memory barriers (e.g., `sfence` and `mfence`) [3]. For data larger than 8 bytes, the crash consistency can be guaranteed using logging or **Copy-on-Write (CoW)**, causing additional overheads for extra PM writes [42, 56]. Non-temporal stores, which bypass CPU caches, are leveraged to directly write data to PM [52]. Recently, a new hardware feature, called **extended ADR (eADR)** [45], is proposed to guarantee that data in CPU caches are flushed to persistent domain upon crashes using additional batteries (refer to Section 4.10 for more details). In our implementation to persist temporal stores, clevel hashing leverages `pmem_persist` provided by PMDK [9], which issues `clwb` and `sfence` on our machine.

2.2 Lock-free Concurrency Control

Compare-And-Swap (CAS) primitives are building blocks in existing lock-free algorithms. A CAS primitive atomically performs the following operations: comparing the contents in a memory location with expected values and swapping the contents with new values only if the contents are equal to expected values; otherwise, updating the expected values with the actual contents in the location (or just do nothing) [26]. CAS and other atomic primitives are efficient to provide high scalability, but they do not support data sizes larger than the CPU write unit size (e.g., 8 bytes). For large data size (>8 bytes), CoW is used for atomic updates. CoW first copies data to be modified and performs in-place updates in the copied data. Then the pointer is atomically updated with the pointer to new data using a CAS primitive. The drawback of CoW is the extra writes for the copy of unchanged contents, which degrades the PM system performance [42, 56]. In our clevel hashing, we design the lock-free algorithms using CAS primitives without CoW.

2.3 Basic Hash Tables

A hash table leverages hash functions to calculate the index of a key-value item, thus obtaining $O(1)$ point query performance. Different keys may have the same index, called *hash collisions*. There are some techniques to address hash collisions, e.g., linear probing [42], multi-slot buckets [21, 37, 42, 56], linked list [10, 20, 35, 40], and data relocation [21, 37, 46, 56]. When hash collisions cannot be addressed, a resizing operation occurs to increase the capacity. A typical resizing operation in static hashing schemes allocates a new hash table with 2× as many buckets as the old one, migrates all items from the old hash table to the new one by rehashing, and switches to the new hash table.

2.4 Hashing-based Indexes for PM

Different from indexes on DRAM, PM-based hashing schemes need to ensure the crash consistency.

2.4.1 The Level Hashing Scheme. Our clevel hashing is based on the level hashing [56], a write-optimized and crash-consistent hashing scheme for PM. The overview of level hashing is shown in Figure 1. Level hashing is a two-level structure and the top level has twice the buckets of the bottom level. Each *level* is an array of 4-slot buckets. Level hashing has three main design goals:

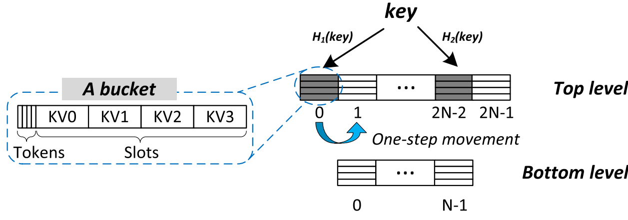


Fig. 1. The two-level structure in level hashing with 4 slots per bucket.

low-overhead crash consistency, write efficiency, and resizing efficiency. Below, we briefly introduce the relevant components in level hashing.

For low-overhead crash consistency, level hashing leverages atomic writes for the tokens in a bucket, thus avoiding logging for insertion/deletion and enabling opportunistic log-free update.

For write efficiency, an insertion operation without resizing introduces one item movement at most while guaranteeing a maximal load factor over 90%. In level hashing, each key-value item has two candidate buckets in one level via two independent hash functions. When the two buckets are full, level hashing tries to perform one-step movement to obtain an empty slot for the new item by trying to move an inserted item to its alternative candidate bucket.

For resizing efficiency, level hashing creates a new level with 2× as many buckets (e.g., 4N buckets in Figure 1) as the top level and migrates stored items in the bottom level (e.g., N buckets) to the new level. Hence, level hashing only migrates one-third buckets for resizing.

The concurrency in level hashing suffers from performance and correctness issues. In terms of resizing, items are rehashed sequentially, blocking concurrent queries of other threads. Level hashing leverages slot-grained locks: A slot lock is acquired before accessing the corresponding slot and released after completing the access. However, there are two correctness issues:

(1) Duplicate Items. A thread holding a single slot lock for insertion cannot prevent other threads from inserting items with the same key into other candidate positions, causing duplicate items with the same key. Duplicate items in hash table violate the update/deletion correctness: A thread updates or deletes an item while future queries may access the duplicate items that are unmodified.

(2) Missing Items. Items in level hashing are movable due to one-step movement and rehashing, while a single slot lock for query cannot prevent the item movements. As a result, one query may miss inserted items due to concurrent moving from other threads.

2.4.2 Concurrent Hashing Indexes for PM. In addition to level hashing, existing crash-consistent PM hashing indexes also explored lock-based concurrency control, as summarized in Table 1. CCEH [42] leverages segment reader-writer locks for queries and segment splitting. To increase the directory size, a global directory lock is required. Dash-EH [38] (or Dash in the context of this article) is also based on extendible hashing and also uses a global directory lock. Dash optimizes search operations using optimistic concurrency control (bucket-grained locks for insertion/update/deletion) and locks all buckets in a segment for splitting. In pmemkv [10], CMAP, a concurrent linked-list based hashing engine for PM, uses bucket-grained reader-writer locks for concurrency control. For resizing, CMAP leverages lazy rehashing to amortize data migration in future queries. P-CLHT [35] is a crash-consistent and cache-efficient hash table with lock-free search. However, during resizing, concurrent insertions to the stale buckets (i.e., buckets whose items have been rehashed) have to wait until the resizing is completed.

As shown in Table 1, we compare our clevel hashing state-of-the-art crash-consistent and concurrency PM hashing schemes. As discussed in Section 2.4.1, level hashing is possible to generate

Table 1. The Comparisons of Our Clevel Hashing with State-of-the-art PM Hashing Indexes in Terms of Performance and Correctness

	Concurrency Control			Correctness Guarantee			Tail Latency
	Search	*Insertion/Update/Deletion*	*Resizing*	**DI**	**MI**	**DL**	
Level [56]	Slot lock	Slot lock	Global metadata lock			Yes	High
CCEH [42]	Segment reader lock	Segment writer lock	Global directory lock		Yes	Yes	Moderate
CMAP [10]	Bucket reader lock	Bucket writer lock	Bucket writer lock + lazy rehashing	Yes	Yes	Yes	High
P-CLHT [35]	Lock-free	Bucket lock	Global metadata lock	Yes	Yes	Yes	Moderate
Dash [38]	Optimistic	Bucket lock	Global directory lock	Yes	Yes	Yes	Moderate
Clevel-Orig [16]	Lock-free	Lock-free	Asynchronous	Yes	Yes		Moderate
Clevel	Lock-free	Lock-free	Asynchronous	Yes	Yes	Yes	Low

For abbreviation, "Level" is the level hashing; "CCEH" is the default CCEH version using the MSB segment index and lazy deletion; "CMAP" is the `concurrent_hash_map` *in pmemkv; "Dash" is the Dash-EH based on extendible hashing; "Clevel" is our clevel hashing; "Clevel-Orig" is our preliminary version of clevel hashing. For correctness guarantee, "DI," "MI," and "DL" indicate duplicate items, missing items, and durable linearizability, respectively.*

duplicate items and miss inserted items in concurrent queries. Since CCEH does not check if a key to be inserted is present in the hash table, CCEH also has the problem of duplicate items. For memory efficiency, CCEH sets a short linear probing distance (16 slots) by default to trade storage utilization for query performance. Due to the frequent locking/unlocking and sequential resizing in level hashing and the recursive amortized rehashing of buckets in CMAP, these two schemes experience high tail latencies. Our preliminary version of clevel hashing (i.e., *Clevel-Orig* in the context of this article) achieves lock-free queries with asynchronous background resizing. However, due to the multiple candidate slots for each key, the absence of locks for insertions introduces reading uncommitted items in search operations, violating the durable linearizability [29] (Section 3.3.2).

Durable linearizability [29] is a strong consistency model as the PM counterpart of linearizability [27]. Essentially, a durable linearizable PM index needs to enforce the operations completed before a crash remain visible and consistent after the system restarts. Durable linearizability is becoming an important standard for PM system designs [20] and bug detection [24]. Note that level hashing and CCEH also suffer from the linearizability violation due to the duplication caused by the buggy implementations. However, if the implementation problems of duplication and missing are addressed, then level hashing and CCEH automatically become durable linearizable. Hence, we classify level hashing and CCEH as durable linearizable in Table 1. Based on our preliminary work, our proposed clevel hashing (i.e., "Clevel") is a holistic scheme that exploits the PM characteristics and explores the index/allocator co-design for high throughput and low tail latency while guaranteeing the concurrency correctness (durable linearizability).

3 THE CLEVEL HASHING DESIGN

3.1 The Clevel Hashing Index Structure

3.1.1 Dynamic Multi-level Structure. The overview of dynamic multi-level structure in clevel hashing is shown in Figure 2. In clevel hashing, there are multiple (≥2) levels and each level is an array of buckets. The *first level* is interpreted as the level with the most buckets, while the *last level* is interpreted as the level with the least buckets. The number of levels in clevel hashing is dynamic: Levels are added for resizing and removed when rehashing is completed. To guarantee high storage utilization, clevel hashing maintains at least two levels [56]. In each level, a key-value item has two candidate buckets via two independent hash functions. Unlike the 4-slot bucket in the level hashing [56], each bucket in clevel hashing has 8 slots by default (discussed in Section 4.2). Each slot occupies 8 bytes and stores a pointer to an item, thus supporting variable-length key-value items via thread-local allocation (Section 3.2).

A ring buffer, called *level ring*, consists of a small fixed number (e.g., 16) of entries to the associated levels. An *active entry* (green entry) in the level ring links to an initialized level that can be

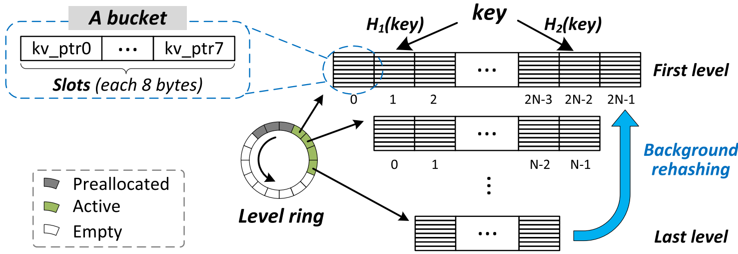


Fig. 2. The dynamic multi-level structure in clevel hashing with 8 slots per bucket.

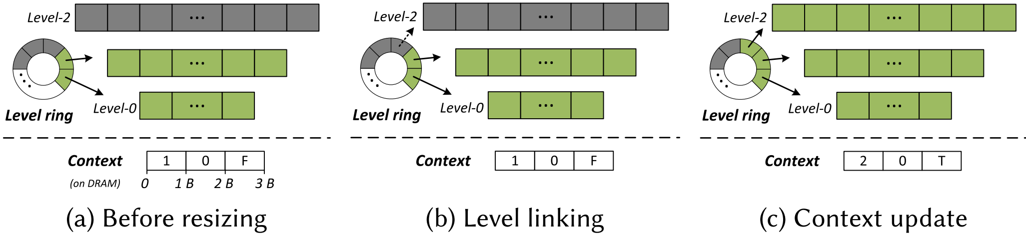


Fig. 3. An example of expanding to three levels in clevel hashing. (a) A two-level structure before resizing occurs. (b) Step-1: linking the preallocated Level-2 via one CAS. (c) Step-2: atomically advancing the `first_level` and setting the `is_resizing` (to be true) of the context via one CAS to activate Level-2. *(Green and gray levels, respectively, indicate active and preallocated levels. The dotted arrow indicates the corresponding level is referenced by the level ring but temporarily invisible for queries before updating the context).*

accessed for queries, i.e., a level in the inclusive range from the last to the first levels. A *preallocated entry* (gray entry) indicates a level is preallocated and ready to be referenced by the entry. We define levels corresponding to active and preallocated entries, respectively, as *active* and *preallocated levels*.

3.1.2 Asynchronous Resizing. If no vacant slots are available for insertions, then clevel hashing adds a new level to the dynamic multi-level index. We propose *context*, a volatile 3-byte metadata consisting of `first_level`, `last_level`, and `is_resizing` (each one occupies one byte) to coordinate resizing and concurrent queries by controlling the concurrency visibility of levels for queries. `first_level` and `last_level`, respectively, denote corresponding entry positions in the level ring for the first and last levels. The corresponding levels in `[last_level, first_level]` are active levels. As the example shown in Figure 3(a), the context $(1, 0, F)$ indicates index status: The first level is Level-1; the last level is Level-0; the index is not being resized. Only Level-1 and Level-0 are active levels.

Clevel hashing performs a resizing operation on two stages: *expansion stage*, a lightweight stage involving only two CASs to expand the index capacity, and *rehashing stage*, a time-consuming stage in which all items of the last level are rehashed and migrated to the first level. We describe the detailed steps in these two stages below.

Expansion Stage: (1) Use one CAS to link the preallocated level next to active levels to the level ring. Specifically, the level entry at `(first_level+1)%LEVELRING_SIZE`, e.g., the entry for Level-2 in the example shown in Figure 3(b), is updated by atomically storing the corresponding preallocated level's address via CAS. `LEVELRING_SIZE` is the level ring size. The CAS may fail because another thread successfully links the preallocated level via the same CAS. In this case, the level entry still needs to be persisted to avoid inconsistencies due to accessing non-persisted levels

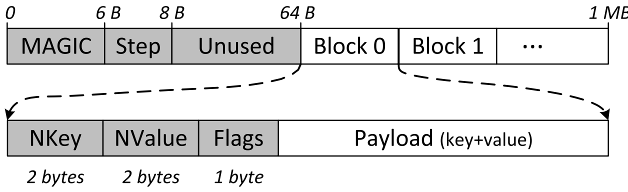


Fig. 4. The one-MB frame layout in clevel hashing. *("Step" indicates the block size).*

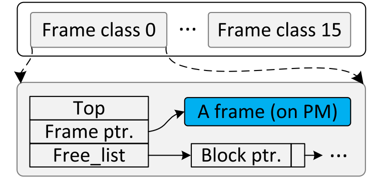


Fig. 5. A thread-local block allocator with 16 frame classes.

in future queries. Since the CAS failure also indicates the success of linking a new level, the expansion progress is guaranteed. (2) Use one CAS to update the context by advancing `first_level` and setting `is_resizing` to be true, e.g., the context in Figure 3(c) is updated to $(2, 0, T)$, to activate the linked preallocated level, e.g., Level-2. Note that increasing `first_level` does not cause the lack of level ring entries, since the maximal number of levels during execution is small (Section 4.1). When the CAS fails, we need to check if `first_level` has been increased: if so, which indicates that another thread has successfully updated the context for expansion, then the expansion stage is completed; otherwise, we need to retry the CAS, since the context (i.e., `last_level` and potential `is_resizing`) is updated in the concurrent rehashing stage. Once the `first_level` is updated, the new linked level (e.g., Level-2 in Figure 3(c)) becomes active and visible for queries.

Rehashing Stage: (1) Rehash each item in the last level. The rehashing operation has two steps: copy the item's pointer to a candidate bucket in the first level via one CAS, and delete the pointer in the last level (without CAS). The concurrency correctness is guaranteed and described in Section 3.3. If the CAS fails, then we find another empty slot in the first level. If no empty slots are found, then we need to go to the expansion stage for expansion and then continue rehashing. (2) When rehashing is completed, one CAS is applied to update `last_level` and optional `is_resizing` (to false only if two levels remain) of the context. If the CAS fails, indicating the context is modified by concurrent expansion or rehashing, then we try again if `last_level` is unmodified.

To address the high latency for blocking resizing, clevel hashing leverages background threads to enable asynchronous resizing. The heavy rehashing stage is delegated to background threads, called *rehashing threads*, which rehash items until there are two active levels left. The threads processing queries are called *worker threads*. By offloading rehashing operations to background threads, resizing no longer blocks queries in worker threads.

Multiple rehashing threads are supported. Specifically, the buckets in the last level are divided into several groups and each group of buckets are processed by one rehashing thread. For example, given two rehashing threads for N buckets, one thread rehashes the buckets at $[0, N/2)$ while the other rehashes the buckets at $[N/2, N)$.

3.2 PM-aware Holistic Optimization

Clevel hashing is a holistic scheme that minimizes the tail latency of concurrent queries by co-designing data indexing and memory management. In particular, based on PM characteristics and properties (Section 2.1), we propose a PM-aware index/allocator co-design for high performance including the following three main components:

3.2.1 Thread-local Lock-free Allocation. Unlike existing general-purpose PM allocators, clevel hashing explores the design space of memory management for PM hashing schemes. We tailor the allocator to clevel hashing for high scalability and low latency. In the memory mapped PM pool, there are two PM data types requiring dynamic allocation: blocks for key-value items and

levels consisting of buckets. In particular, clevel hashing performs thread-local lock-free memory management for the key-value items, enabling low-contention allocation and PM-efficient sequential writes. The levels of buckets are preallocated by rehashing threads, thus avoiding the blocking due to level allocation for worker threads.

Inspired by Memcached [7], our clevel hashing leverages fixed-size frames to support various item sizes. A *frame* is a one-megabyte PM region consisting of metadata and multiple blocks with the same size, as shown in Figure 4. Different frames may have different block sizes. In the clevel hashing, there are 16 predefined block sizes identified by different classes, starting from 64 bytes (class 0) and increasing at a growth rate of 1.25 [7]. The maximal predefined block size (class 15) is 1,856 bytes (aligned to cache line size) and large enough for common allocation demands, since most key-value items in real-world scenarios are smaller than a few hundreds of bytes [14, 21]. Items larger than the maximal block size can be stored by concatenating several blocks of class 15.

We propose thread-local lock-free allocators for blocks to reduce the latency of item allocation in concurrent queries. As shown in Figure 5, each worker thread owns a local block allocator, managing 16 frames in different classes. The *frame class* reflects the class of internal block size. In a frame class of a block allocator, `Top` denotes the index of the first available block in the associated frame. When a worker thread allocates PM for a key-value item, the corresponding thread-local block allocator first tries to find a new block in the matched frame. Specifically, the allocator finds a best-fit frame class in which the payload size is equal to or slightly larger than the item size. Supposing that frame class FC_m ($0 \leq m \leq 15$) is selected, we find an empty block in the frame starting from `Top`. If no vacant blocks are available, then we check FC_m's free list, which references the reclaimed blocks from previous update and delete operations. If the free list is empty, then the allocator allocates a new frame from PM by atomically increasing `frame_counter`, a global atomic frame pointer at the PM pool (growing upwards), and updates the frame pointer of FC_m. Note that the thread-local block allocators not only reduce the contention in allocation, but also are friendly to PM. Due to the local frames for each worker thread, the writes of similar-sized items tend to be sequential PM writes in the same frame, which are known to deliver higher performance than random PM writes [12, 30, 52]. Moreover, block allocators are stored on DRAM and do not incur extra PM writes.

In the clevel hashing, the buckets in a level, i.e., an array of buckets, are preallocated by rehashing threads to reduce the query latencies in worker threads. Similar to block allocation, a global atomic bucket pointer is used to allocate buckets from the end of the PM pool (growing downwards). Rehashing threads maintain some (e.g., 3) preallocated levels and preallocate new levels if necessary before starting migrating items in the last level.

The dynamic regions for frames (*frame region*) and levels of buckets (*level region*) grow in opposite directions in the PM pool: The frame region grows upwards after the index metadata, and the level region grows downwards from the end of the PM pool. When the two dynamic regions meet, the reclaimed levels of buckets (at the end of the PM pool) can be re-allocated for key-value items. When there are no reclaimed levels for items or free space for buckets, the PM pool is considered as out of free memory.

3.2.2 Write-optimized Operation. Our optimized index design, which cooperates with thread-local block allocators, avoids expensive write-ahead logging in existing general-purpose PM management mechanisms [9, 54]. In particular, we leverage the item address stored in the multi-level structure as the commit flag for item allocation and reclamation to enable write-optimized insert/update/delete operations, thus mitigating the software overheads due to the write inefficiency of PM hardware. For example, for an insert operation that does not experience rare resizing operations (Section 3.3.2), only two PM writes are required, i.e., one write (non-temporal store) to

accommodate the variable-length item in allocated PM space and the other (CAS) to atomically store the item address in the index. This optimization is also applied on the item allocation for update operation. The deletion of an item involves two PM writes: one write for atomically clearing the item pointer and the other for the epoch-based block reclamation (Section 3.3.4). Hence, no extra PM write is needed for most insert/update/delete operations in the software stack (PM management and index manipulation) of clevel hashing. We discuss the crash recovery and consistency guarantee of the allocation/reclamation in clevel hashing below.

3.2.3 Lightweight Crash Consistency. By using the index/allocator co-design, our clevel hashing provides low-overhead log-free crash consistency for both stored key-value items and index metadata. Once an unreferenced item, i.e., a key-value item occupying a block that is not referenced in the hash table, is found during recovery after a system failure, the corresponding insert/update/delete operation is considered to be uncompleted and inconsistent. Hence, clevel hashing reclaims the block for the unreferenced item before receiving requests. However, the aforementioned scheme incurs a high performance cost for scanning all blocks in a large PM pool. To alleviate the performance overhead of scanning, we propose to record the addresses of frames referenced by thread-local block allocators, called *working frame addresses*, in the index metadata. As a result, the recovery only needs to scan the blocks in these working frames, thus significantly reducing the recovery time. Moreover, the working frame address is updated only when a new frame is allocated so the overhead for persisting working frame addresses is limited. In addition to the aforementioned crash-recovery mechanism, clevel hashing supports fast restarts from normal exits to enable instant recovery. More details are described in Section 3.4.

In terms of index metadata, clevel hashing only maintains the level ring and critical metadata (e.g., the working frame addresses) on PM, leaving other recoverable metadata (e.g., context and thread-local allocators) on DRAM. All PM-resident metadata are persisted after being modified. Note that a work thread is not possible to access a non-persisted active level, since the context is updated only after the persistency of level activation, avoiding reading non-persisted levels. The metadata in DRAM are rebuilt according to the recovered PM data. For instance, the context is reconstructed according to the restored active levels during recovery. Overall, the overheads for crash-consistency guarantee in our clevel hashing are low due to the log-free mechanism and low persistency frequency, thus limiting query latencies.

3.3 Lock-free Concurrency Control

In clevel hashing, we propose lock-free algorithmsfor search, insert, update, and delete operations.

3.3.1 Search. The search operation in clevel hashing needs to check all candidate buckets until an item matching the key is found. There are two main problems for lock-free search operations: (1) *High latency for pointer dereferencing.* Since clevel hashing only stores item addresses in hash table to support variable-length key-value items, pointer dereferencing is required to obtain the corresponding key for comparison, causing high cache miss ratios and extra PM reads. (2) *Missing inserted items due to the data movement for rehashing.* The asynchronous resizing operation migrates items from the last level to the first level, and therefore, searching without any locks may miss inserted items.

To mitigate the pointer dereferencing overheads, clevel hashing leverages a summary tag to avoid unnecessary PM reads for key-value items. A tag is the summary of a full key, e.g., two bytes from the key's hash value. The tag technique is inspired from MemC3 [21], and we add atomicity for the pair of tag and pointer. To search a key, only when the tag of the key matches the tag stored in hash table, we fetch the full key by pointer dereferencing and compare the two keys. A rare case is that the two keys with the same tag do not match, called *false positive*. For 16-bit tags, the false

positive rate is $1/2^{16}$. However, false positives do not cause any correctness problems, since full-size keys are compared when their tags match. Instead of allocating additional space for tags in MemC3, clevel hashing stores a tag in the unused 16 highest bits of a pointer. Currently, on the widely used four-level paging system, a pointer only consumes 48 bits on the ×86_64 architecture, thus leaving 16 bits unused in 64-bit pointers [43, 48]. By reusing reserved bits, clevel hashing updates a pointer and corresponding tag using one atomic primitive.

To avoid missing inserted items, clevel hashing searches from the last level to the first level, called *bottom-to-top* (*b2t*) search strategy. The idea behind b2t searching is to follow the direction of bottom-to-top data movement for rehashing. However, due to the dynamic change of levels, search operations may still miss items rehashed to a new level: After a search operation starts, a new level is added and the target item pointer is rehashed to the new level. To avoid such missing, clevel hashing checks whether the context after search remains unchanged. If the context changed, then redo the search. The overheads of re-executing search operations are low, since the rare context changes only occur when an active level is added (i.e., index capacity expansion) or removed (i.e., completed rehashing of a level).

3.3.2 Insertion. An insertion operation inserts a key-value item if the key to be inserted does not exist in the index (checked by the b2t search). If the key is absent, then a block is allocated (Section 3.2) to store the key-value item, and the item pointer (with its summary tag) is atomically inserted into a less-loaded candidate bucket. Resizing is triggered if no vacant slots are found in candidate buckets. There are two correctness problems for lock-free insertion: (1) *Duplicate items from concurrent insertions*. Without locks, different threads may simultaneously insert items with the same key into different slots. (2) *Loss of new items inserted to stale buckets of the last level*. If new items are inserted into stale buckets, then these inserted items will be lost after the last level is reclaimed.

It is challenging to avoid duplicate items from concurrent lock-free insertions to different slots, since an atomic primitive only guarantees the atomicity of a CPU write unit (8 bytes). Though these duplicate items can be handled in future updates or deletions [16], the search operation is possible to observe different values for the same key when resizing occurs before the duplication is fixed in updates or deletions. We define the problem of reading transient items due to duplication as *reading uncommitted items*. Essentially, if we directly apply lock-free insertions on multiple candidate positions, e.g., Clevel-Orig [16], then different threads are possible to observe different operation orders, thus violating the (durable) linearizability [29].

We propose ***volatile announcement array (VAA)***, a configurable fixed-size array on DRAM to efficiently synchronize concurrent insertions in a lock-free manner and avoid the aforementioned problem of reading uncommitted items. Each element of VAA consists of eight bytes to store the item pointer with its summary tag. VAA is empty at the beginning of system starts or recovery. A *helping mechanism* is proposed to guarantee the linearizability of insertions. Specifically, for a worker thread performing an insertion operation, after the allocation of the key-value item, the thread tries to atomically insert the item pointer into the corresponding element of VAA (e.g., modulo the VAA size) via one CAS primitive. If the CAS succeeds, indicating the successful announcement of the item to be inserted, then the thread inserts the item pointer into the multi-level hash table on PM (if the key does not exist); otherwise, the thread needs to help insert the item pointer stored in the VAA element and then retries the CAS on VAA to resume the insertion operation. After the insertion operation is completed, the corresponding element of VAA is atomically cleared. As a result, for concurrent insertions with the same key from multiple threads, only the first item, which corresponds to the first thread performing a successful CAS update on the VAA element, is inserted. The insertions of the following duplicate items will fail due to the

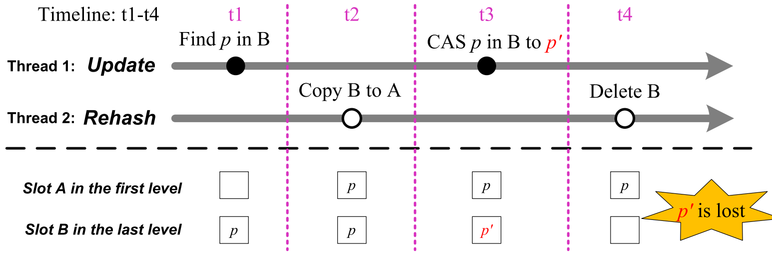


Fig. 6. The update failure due to rehashing. (*"p": pointer to the old item, "p′": pointer to the updated item*).

existence of the same key from the inserted item. Note that the collision of two concurrent insertions in VAA may introduce duplicate pointers to the same item due to the helping mechanism. However, the duplicate pointers do not affect the consistency of search operations, because these pointers refer to the same item. Future updates (Section 3.3.3) and deletions (Section 3.3.4) are able to handle the duplication, thus guaranteeing the linearizability of lock-free concurrent operations (Section 3.5). Moreover, providing that two keys to be inserted are different, the possibility of triggering the helping mechanism depends on the configurable VAA size and is usually low. For example, for the VAA with 65,536 entries, which occupy 512 KB DRAM, the collision rate for two insertions with different keys is about 0.0015% (1/65,536). Hence, the overheads for the proposed helping mechanism are low.

For the second correctness problem, i.e., the loss of new items, we design a context-aware insertion scheme including two strategies: (1) Before an insertion, do not insert an item into the last level when the hash table is resizing; (2) After the insertion, if the table started resizing during the insertion and the item has been inserted into the last level, then we redo the insertion using the same item pointer. The rare re-execution of insertion leads to possible but benign duplicate pointers in the hash table.

3.3.3 Update. For an update operation, given an inserted item matching the key, clevel hashing allocates a block to store the new item and atomically updates the pointer in table via CAS. We propose the following methods to ensure correctness:

(1) Content-conscious `Find` ***to Handle Duplication.*** There are three cases for duplication: the helping mechanism triggered by VAA collisions, the retry of context-aware insertion, and data movement for rehashing. In particular, the duplication refers to multiple pointers to the same item. If two duplicate pointers are stored in different levels, then we need to keep the pointer that is closer to the first level, since rehashing threads may delete the pointer in the last level. If two pointers are in the same level, then keeping either pointer is identical. We design a content-conscious `Find` process before update to handle possible duplication in two steps. First, apply b2t search to check if there are two slots storing pointers to the matched key. Second, delete the pointer that first occurs. By removing duplicate pointers, the `Find` process returns at most one slot address containing the matched item pointer for the atomic update.

(2) Rehashing-aware Scheme to Avoid Update Failures. As shown in Figure 6, a specific execution order of update and rehashing causes loss of updated values. The updated item referenced by p′ is deleted by the rehashing thread. We propose a low-overhead rehashing-aware scheme to avoid such update failures. Specifically, before the `Find`, we record the bucket index (e.g., RB_{idx}) being rehashed. After the CAS for update, check the rehashing progress again (e.g., RB'_{idx}). If the context did not change during the update, then an additional `Find` is required only when meeting three conditions: (1) the table is during resizing; (2) the updated bucket is in the last level; (3) the

updated bucket index B_{idx} satisfies $RB_{idx} \leq B_{idx} \leq RB'_{idx}$. Since it is rare to simultaneously meet all conditions, the correctness is guaranteed with low overheads.

3.3.4 Deletion. Similar to updates, deletion operations also need to handle duplicate pointers. Unlike the careful `Find` for updates to fix duplication, clevel hashing deletes all matched items through the b2t search. In particular, all matched item pointers are deleted (one CAS for each deletion) and then the corresponding block containing the item is gracefully recycled via epoch-based memory reclamation [20, 38] (Section 3.5.1). Lock-free deletions are also prone to failures like update operations. Hence, we use the rehashing-aware scheme described in lock-free update (Section 3.3.3) to guarantee the correctness of deletion.

3.4 Recovery

Clevel hashing supports different recovery modes for prevalent normal exits, called *fast recovery mode*, and rare unexpected crashes, called *crash recovery mode*. Before initialization or recovery, clevel hashing sets the status flag in metadata to 1. The status flag is unset after a normal exit, which ensures that no queries are being processed and resizing operations are completed.

Fast Recovery Mode (the status flag is 0). Only volatile index metadata need to be rebuilt, since all previous queries have been completed and the stored key-value items are consistent. The context is recovered from the level ring. For thread-local allocators, frame pointers are restored from the working frame addresses in persistent metadata. The `Top` and free list in each frame class are reset. Note that VAA is dedicated to synchronize concurrent insertions and does not need to be restored.

Crash Recovery Mode (the status flag is 1). In addition to the aforementioned rebuilding of volatile context and thread-local allocators, clevel hashing also needs to handle the partial items in working frames during recovery. Specifically, for a block in a working frame, if the stored item is linked in the hash table, then the item is consistent and clevel hashing handles duplication. Otherwise, the item is incomplete and the block can be reclaimed, since the corresponding insertion or update is abnormally terminated before the consistent item is durably persisted. For uncompleted rehashing, background threads resume the rehashing in the last level. To detect partial index initialization, clevel hashing writes a magic number (a constant text value as the identifier) as a commit flag in the index metadata after the initialization finishes. After restarts, if the restored magic number is inaccurate, then the index needs to be rebuilt.

3.5 Correctness

3.5.1 The ABA Problem. Clevel hashing guarantees against the ABA problem [41] in the lock-free concurrency control. The ABA problem refers to the problem that a thread using atomic primitives (e.g., CAS) fails to detect the content changes of a memory block, because the block is reclaimed and reused by other threads in a short time (e.g., between copying old data and comparing in CAS). Our clevel hashing leverages epoch-based memory reclamation for thread-safe garbage collection following existing designs [20, 38]. As a result, the corresponding block of a deleted item is reclaimed only after the queries accessing the item are completed, thus avoiding the buggy reuse of reclaimed blocks in concurrent atomic primitives.

3.5.2 Durable Linearizability. Our clevel hashing guarantees durable linearizability [29], a strong consistency model for PM systems. There are two aspects for the requirements of a durable linearizable index for PM. First, the index needs to be linearizable, which implies that different threads must share a consistent operation history as being executed in a sequential manner. Second, the PM modifications in an operation need to be persisted before the operation is completed. Based on these requirements, we clarify the corresponding guarantees in clevel hashing below.

Our clevel hashing is linearizable for concurrent queries. In general, the linearization points for queries (except for concurrent insertions with the same key) correspond to the atomic primitives to slots in the hash table, e.g., the successful CAS update for insertion/update/deletion and the atomic load for search. For concurrent insertions with the same key, the order of these insertions corresponds to the order of successful announcement in VAA via CAS. The proposed helping mechanism for the item pointers in VAA enforces that the previously announced item in a VAA element is inserted and visible before the following items corresponding to the same VAA element, thus enforcing the linearizability for concurrent insertions and avoiding the reading uncommitted problem (Section 3.3.2). Moreover, the duplicate pointers are handled by updates/deletions. Hence, the atomicity of atomic primitives for accessing the table's slots guarantees the linearizability.

The key to satisfy the second requirement is to avoid accessing non-persisted items, namely, *reading unflushed problem*, for lock-free concurrent queries based on atomic primitives. In our clevel hashing, an item is persisted before the corresponding item pointer is inserted into the hash table. Moreover, adopting the existing pointer marking technique [26] ensures the persistency of item pointers for queries. Specifically, an unused bit (e.g., the most significant bit) of the pointer is used as the persistency bit. The thread reading a non-persisted pointer (i.e., the persistency bit is 0) helps flush the pointer and atomically sets the persistency bit to 1. The overheads of the helping mechanism for persisting pointers can be further optimized [20]. For the index metadata, the level ring is persisted before the CAS update of the context. Alternatively, the recent PM platform with eADR support embraces the CPU cache in the persistence domain via additional batteries, thus eliminating the flush requirements for PM modifications. Hence, the reading unflushed problem does not exist on the eADR-enabled PM platform [45]. In summary, by addressing the reading uncommitted and reading unflushed problems, our clevel hashing is durable linearizable.

4 PERFORMANCE EVALUATION

4.1 Experimental Setup

Our experiments run on a server equipped with 768 GB Intel Optane DC PMM 100 Series (128 GB × 6) and 192 GB DRAM (16 GB × 12). There are two CPU sockets in the server and each socket is equipped with a 26-core Intel Xeon Gold 6230R (64 B for cache line size). These six Optane DC PMMs, configured in interleaved and *App Direct* mode, are connected to the same CPU socket. To avoid the impact of **non-uniform memory access (NUMA)**, all the experiments are conducted on the CPU that has local Optane DC PMMs by pinning threads to the CPU, following RECIPE [35]. Existing black-box NUMA-aware optimizations for indexes [15, 47] are applicable to our clevel hashing to improve the scalability when running with more than one NUMA nodes.

In our evaluation, we compare the following concurrent hashing-based index structures for PM:

- **Level:** This is the concurrent level hashing [56] with two levels. The level hashing uses slot-grained reader-writer locks for queries and a global resizing lock for resizing.
- **CCEH:** CCEH [42] is a PM-optimized extendible hashing scheme. CCEH supports dynamic resizing through segment splitting and possible directory doubling. Reader-writer locks are used for segments and the directory.
- **CMAP:** The CMAP in pmemkv [10] is a concurrent hashing table with separate chaining for collided items. CMAP leverages bucket-grained reader-writer locks and supports lazy rehashing (triggered when accessing a bucket).
- **CMAP-TBB:** This is an optimized version of CMAP that adopts the spinlocks from Intel TBB library [5] for concurrency control.
- **P-CLHT:** P-CLHT [35] is a cache-efficient hash table with separate chaining. P-CLHT supports lock-free search while insertions and deletions require bucket-grained locks. The

Table 2. Workloads from YCSB for Macro-benchmarks

Workload	Read ratio (%)	Write ratio (%)
Load A	0	100
A	50	50
B	95	5
C	100	0

resizing in P-CLHT requires a global lock. During resizing, another helper thread (one helper at most by default) is allowed to perform concurrent rehashing.

- **Dash:** Dash [38] denotes a scalable extendible hashing scheme for PM. Unlike CCEH, Dash leverages optimistic concurrency control for search and bucket-grained locks for insert and delete operations.
- **Clevel-Orig:** This is the preliminary version of our clevel hashing [16] providing asynchronous resizing and lock-free concurrency control for all queries. Clevel-Orig leverages a linked list, called *level list*, to manage dynamic levels. Metadata, e.g., the 17-byte context (two pointers to the entries in the level list and a one-byte flag), are stored on PM.
- **Clevel:** This is our clevel hashing, a holistically optimized PM hashing scheme to achieve both high performance and strong consistency (i.e., durable linearizability).

We adopted the open-source PMDK-based versions of Level, CCEH, CMAP, P-CLHT, and Dash for comparisons. The PMDK version is 1.10. For CCEH, we leverage the open-source version provided in Dash's repository which has fixed several known crash-consistency issues and concurrency bugs [38]. Following existing PM hashing schemes, we optimized Level, P-CLHT, and CCEH to support variable-length items by storing pointers in the hash table. Note that CMAP, CMAP-TBB, and Clevel-Orig are implemented using libpmemobj-cpp [9], a C++ binding of PMDK, while other schemes are based on the native PMDK library for C code (i.e., libpmemobj and libpmem).

We leverage YCSB [18] to generate micro-benchmarks in zipfian distribution with default 0.99 skewness [56] to evaluate query latencies. Due to the randomness of hash functions, the trends of results using uniformly distributed workloads are similar to those of skewed workloads [42, 56]. Hence, we omit the results with uniformly distributed workloads. The keys in the workload for insertion are unique. For positive search, update, and delete operations, corresponding keys are present in the index. For negative search operations, queried keys do not exist. We also adapt the real-world workloads from YCSB as macro-benchmarks: The configured update ratio is added to the insert ratio, following existing work [35], since P-CLHT, CCEH, Dash, CMAP, and CMAP-TBB do not support the update operation in corresponding open-source implementations. The workload patterns are described in Table 2, where "read" indicates the search operation and "write" indicates the insert operation. The sizes of each key and value are 19 bytes and 20 bytes, respectively. A typical experiment using the YCSB workload consists of two phases: load and run phases. In the load phase, indexes are, respectively, populated with 16 million and 64 million items for micro- and macro-benchmarks. In the run phase, there are 16 million queries for micro-benchmarks and 64 million for macro-benchmarks. For concurrent executions, the total number of threads for different schemes is 26 by default. During our evaluation of the clevel hashing, we observe that one rehashing thread for 25 insertion threads can guarantee the number of active levels is no more than 5. Hence, we set one thread as the rehashing thread and the level ring size is configured to 16 for all our evaluations. The reported latency and throughput are the average values of 5 runs.

4.2 The Sensitivity of Slots per Bucket

We evaluate the impact of slots per bucket on our clevel hashing in terms of load factor and concurrent performance. Figure 7 shows the load factors of the level hashing and our clevel hashing

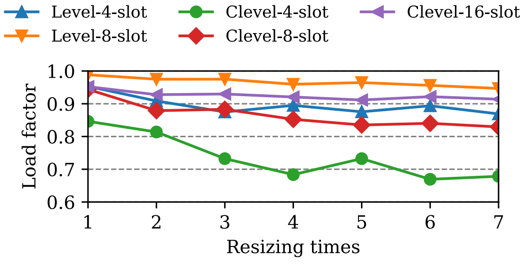


Fig. 7. The impact of slot per bucket on the maximal load factor for level hashing and clevel hashing.

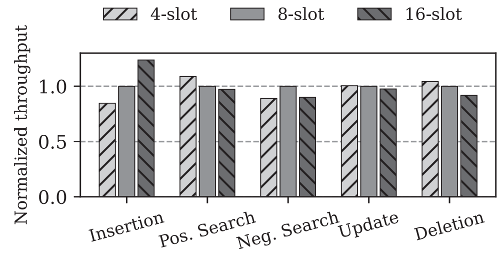


Fig. 8. The normalized concurrent throughput of clevel hashing with different slots per bucket.

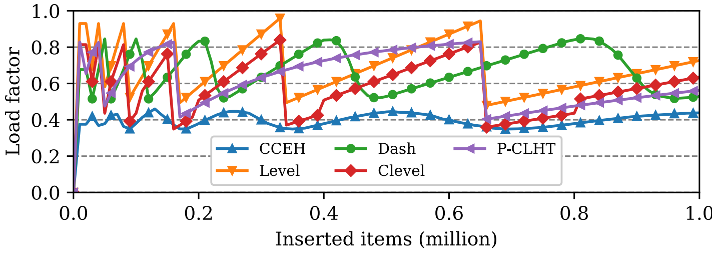


Fig. 9. The load factor per 10,000 insertions.

with different slot numbers when successive resizing operations occur. Due to the lack of one-step movement, the load factor of clevel hashing is lower than that of the level hashing configured with the same number of slots per bucket. However, the 8-slot bucket in clevel hashing increases the number of candidate slots in a bucket, thus achieving a comparable load factor than the level hashing with 4-slot buckets (default configuration).

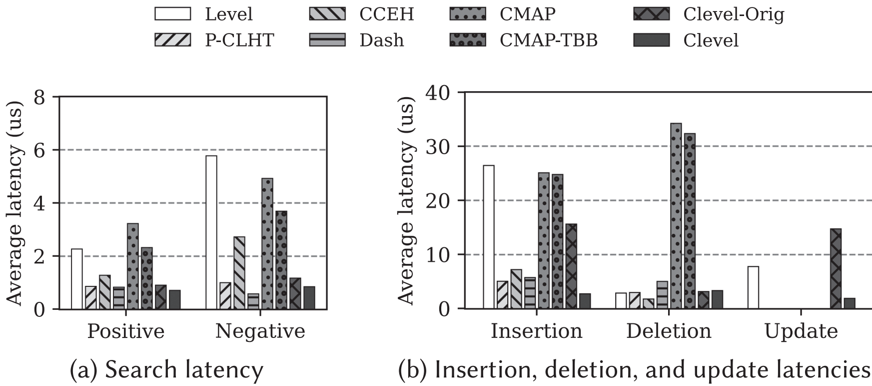
We run the micro-benchmarks with different slot numbers in clevel hashing and measure the concurrent throughput. As shown in Figure 8, with the increase of slots per bucket, the insertion throughput increases due to the higher possibility to find an empty slot in a bucket. Decreasing the slots per bucket reduces the number of slots to be checked and cache line accesses (a 16-slot bucket occupies two cache lines), thus improving the search/update/deletion throughput. Hence, we set the slot number to 8 for the tradeoff between 4-slot and 16-slot buckets.

4.3 The Load Factor

We use an insert-only workload to evaluate the memory efficiency of different PM hashing schemes by recording the load factor after every 10,000 insertions, as shown in Figure 9. The load factor equals to the number of inserted items divided by the number of slots in an index. We omit CMAP because its load factor is always 100%, due to the on-demand memory allocation for each insertion. The maximal load factor of CCEH is no more than 45%, because CCEH probes only 16 slots for hash collisions to constrain the probing overheads. Compared with the level hashing, our clevel hashing does not relocate items in the same level but increases the number of slots per bucket to 8, thus achieving the maximal load factor of 86%.

4.4 Micro-benchmarks

We use the run phase of micro-benchmarks to evaluate the average search latencies (Figure 10(a)) and the results for insertion/update/deletion (Figure 10(b)).



(a) Search latency (b) Insertion, deletion, and update latencies

Fig. 10. The average latency for concurrent queries.

Search. We measure the average latencies when all keys exist (i.e., positive search) or are absent (i.e., negative search) in the table. The amortized rehashing in the critical path induces high search latencies for CMAP and CMAP-TBB. Level hashing needs to check multiple candidate slots and fetch the full-size keys via pointer dereferencing, causing frequent locking/unlocking and multiple PM accesses. The segment-grained locks cause high contention for queries in CCEH. Due to the lock-free operation and tags to filter many unnecessary retrievals for keys, our clevel hashing ensures low average latencies for both positive and negative search operations. Note that compared with Clevel-Orig adopting a similar lock-free search procedure (i.e., checking candidate buckets via the b2t search), Clevel, respectively, reduces the average positive and negative search latencies by 16.4% and 25.2%. The performance improvement is due to the implementation using the low-level PMDK library (i.e., libpmemobj and libpmem) instead of the high-level libpmemobj-cpp, thus eliminating the software overheads for the inefficient indirections. The evaluation results about the throughput using macro-benchmarks (Section 4.5) and tail latencies (Section 4.6) on search performance also demonstrate the inefficiency of libpmemobj-cpp.

Insertion. All schemes have to expand to accommodate 16 million items in the run phase, e.g., Level, P-CLHT, CMAP, CMAP-TBB, Clevel, and Clevel-Orig experience one resizing during this evaluation. In addition to the aforementioned search overheads for checking existence, the resizing operation blocks insertions and increases the average latencies for the level hashing. Similar to the search performance, CMAP and CMAP-TBB suffer from the rehashing in the critical path. The Clevel-Orig also has high average insertion latencies due to the software overheads from libpmemobj-cpp. Moreover, Clevel-Orig needs to persist the context after loading in each insertion and update. Clevel avoids the flush-on-load overheads by adopting the level ring with the context on DRAM. Moreover, our clevel hashing avoids the lock contention in both PM allocation and index manipulation, thus achieving the lowest average insertion latency among all compared schemes.

Update. Due to the PM-aware holistic optimizations, the average update latency of Clevel is about one-seventh of that in Clevel-Orig. Note that the update latencies for other schemes are not presented for the lack of update implementations in the open-source code.

Deletion. The deletion latencies in CMAP and CMAP-TBB are higher than other schemes due to the required rehashing of the bucket if necessary before accessing. CCEH achieves low latency due to the logical deletion by marking the entry as empty, which does not reclaim the allocated memory. Similarly, Level sets the corresponding token for a slot as 0 for the delete operation. Other schemes show similar average latencies for deletion.

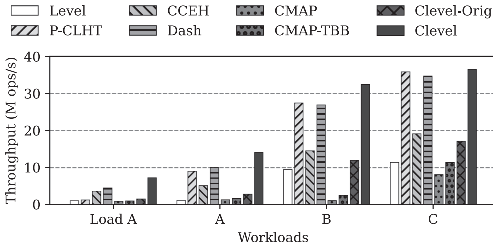


Fig. 11. The concurrent throughput using YCSB workloads.

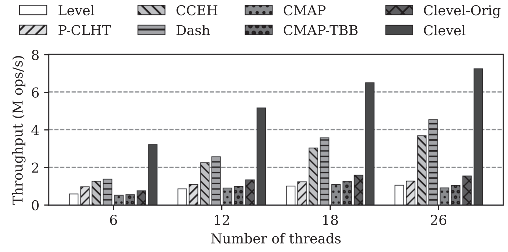


Fig. 12. The insertion scalability using workload Load A.

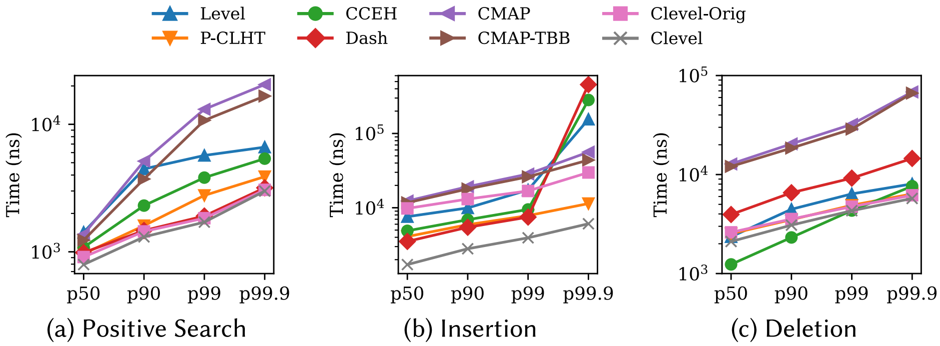


Fig. 13. The median and tail latencies for concurrent queries.

4.5 Macro-benchmarks

We leverage real-world workloads from YCSB as macro-benchmarks to investigate the concurrent throughput of different PM hashing schemes (Figure 11) and the scalability of our clevel hashing (Figure 12). The workload Load A is used in the load phase of the workloads A, B, and C to populate small empty indexes with 64 million items, incurring multiple resize operations (e.g., seven times for Clevel). Resizing also occurs in the run phase of workload A (e.g., one time for Clevel), which involves about 32 million insertions. The concurrent throughput results using workloads Load A and C, respectively, present the trends similar to those in the average search and insertion latencies (Figure 10). For example, CMAP, CMAP-TBB, and Level presenting high insertion and search latencies also show low throughput for workloads Load A (insert-only) and C (search-only). Due to the superior insertion and search performance, our clevel hashing achieves up to 5.7× and 1.6× speedup than state-of-the-art static (P-CLHT) and dynamic (Dash) PM hashing indexes, respectively.

To evaluate the scalability of PM hashing schemes, we measure the insertion throughput with different numbers of threads using the workload Load A. As shown in Figure 12, with the increase of threads, the throughput of clevel hashing increases and is consistently higher than other schemes. The results of other workloads in the macro-benchmarks also show a similar trend.

4.6 Tail Latency

We investigate the tail latency of concurrent positive search, insertion, and deletion operations using the micro-benchmarks. As shown in Figure 13, due to the lock-free progress guarantee and PM-aware holistic optimizations, our clevel hashing experiences the lowest p99.9 latency among the compared schemes, i.e., 2.99 μs, 6.09 μs, and 5.74 μs for positive search, insertion, and deletion

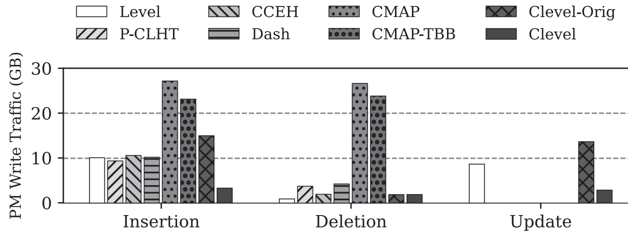


Fig. 14. PM write traffic using micro-benchmarks for concurrent queries.

operations, respectively. Compared with other schemes, Clevel achieves 1.0×–7.7×, 1.9×–7.2×, and 1.0×–7.5× speedup for p99 search, insertion, and deletion latencies, respectively. We observe that the maximal latencies of all schemes are very high, e.g., 13.73 ms for Clevel and 4.41 s for CMAP with the insert-only workload. The main reason is the inefficiency of PM file system (i.e., ext4-DAX). Specifically, the page faults and lazy construction of page metadata introduce spikes in latencies [31, 32]. We have performed a simple test by prefaulting all memory mapped pages, which reduces the maximal insertion latency of Clevel to 73.95 μs but requires 28 seconds for prefaulting during index initialization/recovery. In general, to mitigate the poor maximal latency, we need to address the performance issues in the PM system softwares, e.g., reducing page faults, designing PM-aware file systems, and using raw PM devices (devdax mode). These problems and solutions are orthogonal to the index design and out of the scope of this article.

4.7 PM Write Traffic

PM write traffic is interpreted as the size of written data that are issued from memory controllers and received by PM modules. We evaluate the write traffic using micro-benchmarks, excluding the read-only search workloads, to quantitatively demonstrate the efficiency of the proposed write reduction techniques and potential benefits in PM lifetime. The results are obtained via Intel's `ipmctl` [6], an open-source tool to manage Intel Optane DC PMM. Clevel incurs the lowest PM write traffic among the comparisons for the insert-only workload, e.g., 35.9% of P-CLHT's and 33.0% of Dash's PM writes, as shown in Figure 14. This reduction comes from the log-free allocation and write-optimized insertion via the co-design of index/allocator in our clevel hashing. To further investigate the write reduction, we also implement a memory tracer based on Intel pin [39], a dynamic binary instrumentation tool, to record the store instructions for PM executed by CPUs. Our evaluation shows that to insert a 39-byte key-value item, P-CLHT and Dash, respectively, issue store instructions for 154 and 158 bytes on PM. Note that the SSE instructions generated by PMDK for writing an item demand a slightly larger write size (e.g., 64 bytes) including some metadata. The main insertion overheads are caused by the double writes (e.g., 128 bytes) due to write-ahead logging for the crash-consistent item allocation. By co-designing data index and PM management, our clevel hashing only issues 72 bytes for stores: 64 bytes for the log-free item allocation (with padding) and 8 bytes to store the item pointer. In terms of delete and update operations, Clevel also maintains low PM write traffic. The traffics of Level and CCEH for deletions are very low, since their implementations do not reclaim the deleted items (i.e., logical deletion discussed in Section 4.4).

The PM traces collected by the pin-based tool also show that the resizing of Clevel incurs 8.2% PM write size for the insertion micro-benchmark, indicating limited PM write traffic for resizing. The reason is that rehashing only migrates item pointers. Moreover, a resizing operation only rehashes the items in the last level, thus avoiding the full-table rehashing.

Table 3. The Recovery Time (Milliseconds) with Different Scales of Inserted Items

# Items	CCEH	Dash	Clevel (Fast)	Clevel (Crash)
2 M	47	45	142	592
16 M	57	45	142	463
128 M	215	45	142	670

The "Fast" and "Crash" of Clevel, respectively, indicate fast and crash recovery modes.

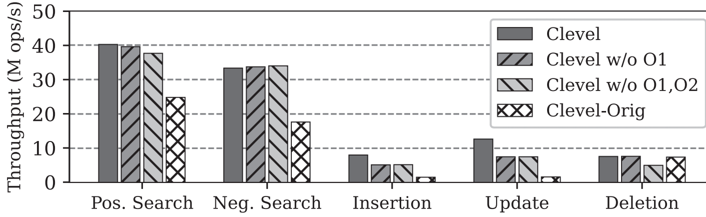


Fig. 15. The improvements of PM-aware optimizations in clevel hashing. *("O1": PM-aware thread-local block allocator, "O2": DRAM-resident context).*

4.8 Recovery Time

We measure the recovery time of different schemes with various scales of inserted items (from 2 to 128 million items). Table 3 shows the recovery time of CCEH, Dash, and Clevel (two recovery modes). Other schemes show a constant recovery time like Dash (i.e., 45 ms) for memory mapping the PM pool. The recovery of CCEH needs to scan the directory, which expands and requires more recovery time with the increase of inserted items. The fast recovery mode in our clevel hashing accounts for prevalent normal exits, and the main recovery time is to rebuild the thread-local block allocators, which is proportional to the number of threads but independent of the item scale. Hence, given a thread number, the recovery time for normal exits of our clevel hashing is also constant, e.g., 142 ms for 25 worker threads. For rare crashes handled by the crash recovery mode, additional checking is required for the working frames and the recovery time depends on the number of items to be checked in these working frames. Since the frame size is small, the checking of the working frames before the crash is fast.

4.9 The Efficiency of PM-aware Optimizations

As shown in Figure 15, we quantitatively analyze the contributions of the main PM-aware optimizations in our clevel hashing: the PM-aware thread-local block allocator, denoted by O1, and the DRAM-based recoverable context design, denoted by O2. We disable corresponding optimizations and evaluate the concurrent throughput using micro-benchmarks. The PM-aware thread-local block allocator often converts a worker thread's item store into sequential PM writes and reduces the thread contention for allocation, providing 54.8% and 68.2% concurrent throughput improvements for insert and update operations, respectively. Due to the index/allocator co-design, the hot metadata context becomes recoverable and is placed on DRAM instead of PM, thus avoiding the expensive flush-on-load operations for crash consistency [20, 48]. For example, by adopting the recoverable context design, the concurrent throughput of delete operations is improved by 51.7%. Note that, even for the (positive) search-only workload, Clevel w/o O1 or O2 still achieves 1.5× speedup than Clevel-Orig. The main reason is the software overheads in libpmemobj-cpp, which is used in the implementation of Clevel-Orig and CMAP (Section 4.4).

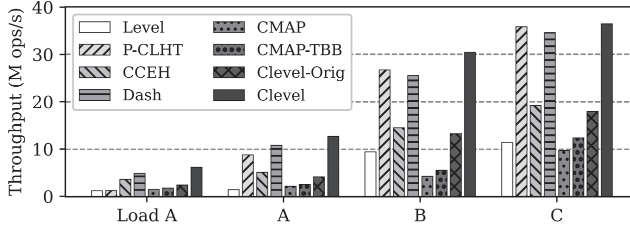


Fig. 16. The concurrent throughput using YCSB workloads on the emulated eADR platform.

We evaluate the performance overhead of VAA using the insert-only micro-benchmark. By disabling VAA, Clevel improves the insertion throughput by 2.3% but suffers from the reading uncommitted problem. The low overhead of VAA comes from the small collision possibility of worker threads in VAA. Assuming there are m worker threads and n entries in VAA, the possibility of triggering the collisions in VAA (i.e., helping for insertion) for one thread is lower than $(m-1)/n$, e.g., 0.037% (24/65,536) for our testbed configuration. Increasing the VAA size or decreasing the insertion ratio in workloads further reduces the performance overhead.

4.10 The Performance Impact on eADR

As shown in Figure 16, we evaluate the throughput of all compared schemes on an emulated eADR platform using macro-benchmarks. The eADR technology is an optional hardware feature (requiring the 3rd generation Intel Xeon processors, Intel Optane PM 200 Series, and OEM supports) to automatically flush the CPU cache data into PM upon system failures by additional batteries [45]. Hence, cache line flushes become unnecessary for eADR-enabled PM [19, 53] but `sfence` is still needed for non-temporal stores [45]. Following existing work [19], we emulate the eADR platform by removing all flushes in the compared schemes. As shown in Figure 16, removing flushes slightly improves the performance in many cases. The marginal improvement in our evaluation mainly comes from the limited CPU cache capacity for large-scale workloads. After populating an index with 64 million key-value items in the load phase, random accesses to buckets and items in the large hash table caused by hash functions introduce frequent cache misses. According to Intel's documentation on eADR [1], eliminating flushes is only beneficial to programs with temporal locality. Moreover, Clevel puts items in PM via non-temporal stores for persistency while avoiding cache pollution. The non-temporal store and the following fence cannot be eliminated on the eADR-enabled PM, which further limits the performance benefits of eADR. Overall, Clevel still shows higher throughput than other schemes, demonstrating the applicability of our lock-free concurrency control mechanism and PM-aware index/allocator co-design on the emulated eADR platform. We leave eADR-based optimizations (e.g., reducing memory footprint) as future work.

4.11 Discussion

The reduction of hash table size. The current implementation of clevel hashing does not support the reduction of hash table size. To adapt the clevel hashing, the direction of resizing and searching needs to be reversed. Specifically, to reduce the table size in clevel hashing, we need to create a new level with half of the buckets in the last level and rehash the items from the first level to the last level. The searching for items needs to be performed in a top-down manner to avoid missing inserted items. The clevel hashing with non-blocking concurrent reduction is our future work.

The isolation level. As discussed in the Section 3.5, our clevel hashing enforces the durable linearizability. Specifically, item insertion/update/deletion operations are persisted before visible to concurrent threads, and the linearization points are the successful CAS update for modifying slots.

Moreover, the epoch-based memory reclamation guarantees the thread-safety. As a result, the items read by one thread are consistent and durable, achieving the *read committed* isolation level.

Space overhead. The main metadata overheads in our clevel hashing come from the working frame addresses on PM (8 bytes per thread) and the volatile announcement array (512 kilobytes for 65,536 entries) on DRAM. For storage utilization, the maximal load factor of clevel hashing is over 80% before resizing. The preallocation of levels by rehashing threads avoids the contention of the level allocation in the expansion stage. Moreover, the number of preallocated levels is 3 and the allocation is log-free. Since all the preallocated levels will be used in the following resizing operations, the overall storage utilization is high in our clevel hashing.

Fragmentation. The thread-local block allocator may experience PM fragmentation. It is possible that the allocated block size is slightly larger than the actual key-value item size, called internal fragmentation. However, due to the best-fit allocation strategy, the space overhead for internal fragmentation is limited. The external fragmentation is interpreted as the lack of contiguous PM space for a request though the total amount of scattered free memory in the PM pool is sufficient. In our clevel hashing, the two dynamic regions (i.e., frame and level regions) grow towards each other. The fixed-size frames avoid generating PM regions smaller than the frame size (e.g., 1 MB). Since the buckets of a new level is allocated in a batch and the reclaimed levels can be re-allocated for items, the external fragmentation is constrained in our clevel hashing.

5 RELATED WORK

5.1 Hashing-based Index Structures for PM

Recent works have proposed some hashing-based index structures optimized for PM. Static hashing schemes, e.g., path hashing [55] and level hashing [56] using sharing-based index structures, LF-HT [20] and P-CLHT [35] based on separate chaining, suffer from poor resizing performance due to the exclusive global lock for metadata. CCEH [42] and Dash-EH [38] are based on extendible hashing. The capacity expansion requires lock-based segment splitting and optional global directory doubling. The `concurrent_hash_map` in pmemkv [10] supports concurrent lazy rehashing. However, the amortized rehashing in the critical path of queries increases the latency and may cause recursive rehashing. Unlike existing schemes, clevel hashing leverages dynamical multi-level structure for concurrent asynchronous resizing, exploits the PM-aware index/allocator co-design, and proposes lock-free algorithms for all queries, showing the feasibility to achieve both high throughput and low (tail) latency.

5.2 Lock-free Concurrent Hashing Indexes

Conventional designs leverage different techniques, e.g., lock-free linked lists [40] and marking with helping mechanism [43], to build lock-free hashing indexes. However, these schemes are unaware of the PM characteristics and cause extra PM writes. Recent work [23] uses PSim [22] to build a wait-free resizable hash table. The wait-free technique relies on copying the shared object and helping mechanism, which still leads to extra writes on PM and introduces overheads due to helping. Moreover, the extendible hashing structures are memory inefficient, as shown in our evaluation. Different from existing lock-free hashing schemes built on DRAM, clevel hashing designs PM friendly and memory efficient multi-level structures with simple but effective context-aware mechanism to guarantee correctness and crash consistency.

6 CONCLUSION

In this work, we propose clevel hashing, a lock-free high-performance concurrent hashing scheme for PM. Clevel hashing leverages the dynamic memory-efficient multi-level design and

asynchronous resizing to address the blocking issue due to resizing. The lock-free concurrency control avoids the lock contention for all queries while guaranteeing the durable linearizability. The PM-aware holistic optimizations with the index/allocator co-design reduce query latencies. Our results using Intel Optane DC PMM demonstrate that clevel hashing achieves higher concurrent throughput with lower latency than state-of-the-art hashing indexes for PM.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their constructive comments and suggestions. We have released the source code for public use in GitHub.

REFERENCES

[1] Intel. 2022. eADR: New Opportunities for Persistent Memory Applications. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.
[2] Intel. 2022. Intel Optane Persistent Memory 200 Series Brief. Retrieved from https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-persistent-memory-200-series-brief.html.
[3] Intel. 2022. Intel®Architecture Instruction Set Extensions Programming Reference. Retrieved from https://www.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf.
[4] Intel. 2022. Intel®Optane™DC persistent memory. Retrieved from https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html.
[5] Intel. 2022. Intel®Threading Building Blocks. Retrieved from https://github.com/intel/tbb.
[6] Intel. 2022. IPMCTL User Guide. Retrieved from https://docs.pmem.io/ipmctl-user-guide/.
[7] Brad Fitzpatrick. 2022. Memcached. Retrieved from https://memcached.org/.
[8] Linux. 2022. perf: Linux profiling with performance counter. Retrieved from https://perf.wiki.kernel.org/index.php/Main_Page.
[9] Intel. 2022. Persistent Memory Development Kit. Retrieved from http://pmem.io/.
[10] Intel. 2022. pmemkv. Retrieved from http://pmem.io/pmemkv/index.html.
[11] Redis Ltd. 2022. Redis. Retrieved from https://redis.io/.
[12] Shoaib Akram. 2021. Performance evaluation of intel optane memory for managed workloads. *ACM Trans. Archit. Code Optim.* 18, 3 (2021), 29:1–29:26. DOI: https://doi.org/10.1145/3451342
[13] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.* 11, 5 (2018), 553–565.
[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. ACM, 53–64.
[15] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box concurrent data structures for NUMA architectures. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 207–221.
[16] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free concurrent level hashing for persistent memory. In *USENIX Annual Technical Conference (USENIX ATC)*. USENIX, 799–812.
[17] Ping Chi, Wang-Chien Lee, and Yuan Xie. 2016. Adapting B^+ -tree for emerging nonvolatile memory-based main memory. *IEEE Trans. Comput.-aid. Des. Integr. Circuits Syst.* 35, 9 (2016), 1461–1474. DOI: https://doi.org/10.1109/TCAD.2015.2512899
[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *1st ACM Symposium on Cloud Computing (SoCC)*. ACM, 143–154.
[19] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. 2022. NVAlloc: Rethinking heap metadata management in persistent memory allocators. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 115–127.
[20] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-free concurrent data structures. In *USENIX Annual Technical Conference (USENIX ATC)*. USENIX, 373–386.
[21] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 371–384.

[22] Panagiota Fatourou and Nikolaos D. Kallimanis. 2014. Highly-efficient wait-free synchronization. *Theor. Comput. Syst.* 55, 3 (2014), 475–520.

[23] Panagiota Fatourou, Nikolaos D. Kallimanis, and Thomas Ropars. 2018. An efficient wait-free resizable hash table. In *30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 111–120.

[24] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. 2021. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. ACM, 100–115.

[25] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the idiosyncrasies of real persistent memory. *Proc. VLDB Endow.* 14, 4 (2020), 626–639.

[26] Timothy L. Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *15th International Symposium on Distributed Computing (DISC)*. Springer, 300–314.

[27] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. DOI: https://doi.org/10.1145/78969.78972

[28] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 187–200.

[29] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *30th International Symposium on Distributed Computing (DISC)*. Springer, 313–327.

[30] 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).

[31] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: A hugepage-aware file system for persistent memory that ages gracefully. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. ACM, 804–818.

[32] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *27th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 494–508.

[33] Wook-Hee Kim, Madhava Krishnan Ramanathan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A high performance persistent range index using PAC guidelines. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*. ACM, 424–439.

[34] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 257–270.

[35] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE: Converting concurrent DRAM indexes to persistent-memory indexes. In *27th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 462–477.

[36] Mengya Lei, Fan Li, Fang Wang, Dan Feng, Xiaomin Zou, and Renzhi Xiao. 2022. SecNVM: An efficient and write-friendly metadata crash consistency scheme for secure NVM. *ACM Trans. Archit. Code Optim.* 19, 1 (2022), 8:1–8:26. DOI: https://doi.org/10.1145/3488724

[37] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent cuckoo hashing. In *9th Eurosys Conference (EuroSys)*. ACM, 27:1–27:14.

[38] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. *Proc. VLDB Endow.* 13, 8 (2020), 1147–1161.

[39] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 190–200.

[40] Maged M. Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. ACM, 73–82.

[41] Maged M. Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (2004), 491–504.

[42] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 31–44.

[43] Nhan Nguyen and Philippas Tsigas. 2014. Lock-free cuckoo hashing. In *IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 627–636.

[44] Poovaiah M. Palangappa and Kartik Mohanram. 2017. CompEx++: Compression-expansion coding for energy, latency, and lifetime improvements in MLC/TLC NVMs. *ACM Trans. Archit. Code Optim.* 14, 1 (2017), 10:1–10:30. DOI: https://doi.org/10.1145/3050440

[45] Steve Scargall. 2020. Persistent memory architecture. In *Programming Persistent Memory: A Comprehensive Guide for Developers*. Apress, Berkeley, CA, 11–30. DOI: https://doi.org/10.1007/978-1-4842-4932-1_2

[46] Yuanyuan Sun, Yu Hua, Zhangyu Chen, and Yuncheng Guo. 2019. Mitigating asymmetric read and write costs in cuckoo hashing for storage systems. In *USENIX Annual Technical Conference (USENIX ATC)*. USENIX, 329–344.

[47] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. NAP: A black-box approach to NUMA-aware persistent memory indexes. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 93–111.

[48] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. 2018. Easy lock-free indexing in non-volatile memory. In *34th IEEE International Conference on Data Engineering (ICDE)*. IEEE, 461–472.

[49] H.-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T. Chen, and Ming-Jinn Tsai. 2012. Metal-oxide RRAM. *Proc. IEEE* 100, 6 (2012), 1951–1970.

[50] 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. DOI: https://doi.org/10.1109/JPROC.2010.2070050

[51] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2022. NyxCache: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 1–16.

[52] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 169–182.

[53] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. 2022. HTMFS: Strong consistency comes for free with hardware transactional memory in persistent memory file systems. In *20th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, 17–34.

[54] Lu Zhang and Steven Swanson. 2019. Pangolin: A fault-tolerant persistent memory programming library. In *USENIX Annual Technical Conference (USENIX ATC)*. USENIX, 897–912.

[55] Pengfei Zuo and Yu Hua. 2018. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Trans. Parallel Distrib. Syst.* 29, 5 (2018), 985–998.

[56] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 461–476.

Received 15 May 2022; revised 24 July 2022; accepted 26 August 2022