Lock-free Concurrent Level Hashing for Persistent Memory

Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo,
Huazhong University of Science and Technology

https://www.usenix.org/conference/atc20/presentation/chen

This paper is included in the Proceedings of the
2020 USENIX Annual Technical Conference.
July 15–17, 2020
978-1-939133-14-4

Lock-free Concurrent Level Hashing for Persistent Memory

Zhangyu Chen, Yu Hua, Bo Ding, Pengfei Zuo
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
Corresponding Author: Yu Hua (csyhua@hust.edu.cn)

Abstract

With high memory density, non-volatility, and DRAM-scale latency, persistent memory (PM) is promising to improve the storage system performance. Hashing-based index structures have been widely used in storage systems to provide fast query services. Recent research proposes crash-consistent and write-efficient hashing indexes for PM. However, existing PM hashing schemes suffer from limited scalability due to expensive lock-based concurrency control, thus making multi-core parallel programing inefficient in PM. The coarse-grained locks used in hash table resizing and queries (i.e., search/insertion/update/deletion) exacerbate the contention. Moreover, the cache line flushes and memory fences for crash consistency in the critical path increase the latency. In order to address the lock contention for concurrent hashing indexes in PM, we propose *clevel* hashing, a lock-free concurrent level hashing, to deliver high performance with crash consistency. In the clevel hashing, we design a multi-level structure for concurrent resizing and queries. Resizing operations are performed by background threads without blocking concurrent queries. For concurrency control, atomic primitives are leveraged to enable lock-free search/insertion/update/deletion. We further propose context-aware schemes to guarantee the correctness of interleaved queries. Using real Intel Optane DC PMM, experimental results with real-world YCSB workloads show that clevel hashing obtains up to 4.2× speedup than the state-of-the-art PM hashing index.

1 Introduction

Non-volatile memory (NVM) deployed as persistent memory (PM) offers the salient features of large capacity, low latency, and real time crash recovery for storage systems [12, 30, 38]. Recently, Intel Optane DC persistent memory module (PMM) [2], the first commercial product of PM, is available on the market. Compared with DRAM, PM has 3× read latency and similar write latency [23,24,36]. In the meantime, the read and write bandwidths of PM achieve $1/3$ and $1/6$ of those of DRAM [23, 24, 27, 36]. PM delivers higher performance than SSD and the maximal 512 GB capacity for a single PM module is attractive for in-memory applications [23].

Building high-performance index structures for PM is important for large-scale storage systems to provide fast query services. Recent schemes propose some crash-consistent tree-based indexes, including NV-Tree [37], wB$^+$-Tree [14], FP-Tree [32], WORT [26], FAST&FAIR [22] and BzTree [9]. However, traversing through pointers in hierarchical trees hinders fast queries. Unlike tree-based schemes, hashing-based index structures leverage hash functions to locate data in flat space, thus enabling constant-scale point query performance. As a result, hash tables are widely used in many in-memory applications, e.g., redis [7] and memcached [4].

Existing hashing-based indexes for PM put many efforts in crash consistency and write optimizations but with little consideration for non-blocking resizing (also called rehashing) [27, 30, 40]. A hash function maps different keys into the same location, called hash collisions. In general, when the hash collisions can't be addressed or the *load factor* (the number of inserted items divided by the capacity) of a hash table approaches the predefined thresholds, the table needs to be expanded to increase the capacity. Traditional resizing operations acquire global locks and move all items from the old hash table to the new one. Level hashing [40] is a two-level write-optimized PM hashing index with cost-efficient resizing. The expansion of level hashing only rehashes items in the smaller level to a new level, which only migrates items in 1/3 buckets. However, the resizing operation in the level hashing is single-threaded and still requires a global lock to ensure correct concurrent executions. P-CLHT [27] is a crash consistent variant of Cache-Line Hash Table (CLHT) [17] converted by RECIPE [27]. The search operation in P-CLHT is lock-free, while the writes into stale buckets (all stored items have been rehashed) would be blocked until the full-table resizing completes. Hence, both schemes suffer from limited resizing performance, since the global lock for resizing blocks queries in other threads. Cacheline-Conscious Extendible Hashing (CCEH) [30], a persistent extendible hashing scheme, supports concurrent lock-based dynamic resizing, however coarse-grained locks for shared resources significantly increase the latency. Specifically, CCEH splits a segment, an array of 1024 slots by default, to increase the capacity, which requires the writer lock for the whole segment. Moreover, when the directory needs to be doubled, the global writer lock for directory is needed before doubling the directory. The Copy-on-Write (CoW) version of CCEH avoids the segment locks with the cost of extra writes due to the migration of inserted items. Hence, the insertion performance of CCEH with CoW is poorer than the default version with lazy deletion [30]. The concurrent_hash_map (cmap) in pmemkv [6] leverages lazy rehashing by amortizing data migration over future queries. However, the deferred rehashing may aggregate to a recursive execution in the critical path of queries, thus leading to non-deterministic query performance. Hence, current PM hashing indexes suffer from poor concurrency and scalability during resizing.

A scalable PM hashing index with concurrent queries is important to exploit the hardware resources and provide high throughput with low latency. Nowadays, a server node is able to provide tens of or even hundreds of threads, which enables the wide use of concurrent index structures. Existing hashing-based schemes for PM [27, 30, 40] leverage locks for inter-thread synchronization. However, coarse-grained exclusive locks in a critical path increase the query latency and decrease the concurrent throughput. Moreover, in terms of PM, the persist operations (e.g., logging, cache line flushes, and memory fences), when holding locks, further increase the waiting time of other threads. Fine-grained locks decrease the critical path but may generate frequent locking and unlocking for multiple shared resources. Moreover, the correctness guarantee is harder than coarse-grained locks.

In summary, in addition to crash consistency, we need to address the following challenges to build a high performance concurrent hashing index for PM.

1) Performance Degradation during Resizing. For concurrent hash tables, resizing operations need to be concurrently executed without blocking other threads. However, the resizing operation accesses and modifies the shared hash tables and metadata. Coarse-grained locks for global data ensure thread safety, but lead to high contention and significant performance degradation when the hash table starts resizing.

2) Poor Scalability for Lock-based Concurrency Control. Locking techniques have been widely used to control concurrent accesses to shared resources. The coarse-grained locks protect the hash table, but they also prevent concurrent accesses and limit the scalability. Moreover, the updates of shared data are often followed by flushing data into PM, which exacerbates lock contention. An efficient concurrent hashing scheme for PM needs to have low contention for high scalability while guarantee the concurrency correctness.

In order to address the above challenges, we propose *clevel hashing*, a crash-consistent and lock-free concurrent hash table for PM. Motivated by our level hashing [40], 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 the level hashing, our proposed clevel hashing aims to provide scalable performance and guarantee the correctness for concurrent executions. Unlike existing schemes [27, 30] that convert concurrent DRAM indexes to persistent ones, we propose a new and efficient way to enable persistent indexes to be concurrent with small overheads and high performance. Hence, the clevel hashing bridges the gap between scalability and PM efficiency.

To alleviate the performance degradation for resizing, we propose a dynamic multi-level index structure with asynchronous rehashing. Levels are dynamically added for resizing and removed when all stored items are migrated to a new level. The rehashing of items is offloaded into background threads, thus never blocking foreground queries. Background threads migrate the items from the last level to the first level via rehashing until there are two remaining levels. The two levels ensure a maximal load factor over 80% and the limited accesses to buckets for queries. Therefore, when rehashing is not running (the usual case for most workloads), the time complexity for search/insertion/update/deletion is constant-scale.

To provide high scalability with low latency, we design write-optimal insertion and lock-free concurrency control for search/insertion/update/deletion. The new items are inserted into empty slots without any data movements, hence ensuring write efficiency. For concurrent modifications to the hash table, clevel hashing exploits the atomicity of pointers and uses Compare-And-Swap (CAS) primitives for lock-free insertion, update, and deletion. Guaranteeing the correctness for lock-free queries with simultaneous resizing is challenging, since interleaved operations can be executed in any order and lead to failures and duplicate items. In the clevel hashing, we propose context-aware algorithms by detecting the metadata information changes to avoid inconsistencies for insertion, update, and deletion. The duplicate items are detected and properly fixed before modifying the hash table. In summary, we have made the following contributions in the clevel hashing.

- **Concurrent Resizing.** In order to address the bottleneck of resizing, we propose a dynamic multi-level structure and concurrent resizing without blocking other threads.
- **Lock-free Concurrency Control.** We design lock-free algorithms for all queries in the clevel hashing. The correctness for lock-free concurrency control is guaranteed with low overheads.
- **System Implementation.** We have implemented the clevel hashing using PMDK [5] and compared our proposed clevel hashing with state-of-the-art schemes on

real Intel Optane PM hardware. The evaluation results using YCSB workloads show the efficacy and efficiency of the clevel hashing. We have released the open-source code for public use.[1]

2 Background

2.1 Crash Consistency in Persistent Memory

Persistent memory (PM) provides the non-volatility for data stored in main memory, thus requiring crash consistency for data in PM. For store instructions, the typical maximal atomic CPU write size is 8 bytes. Therefore, when data size is larger than 8 bytes, system failures during sequential writes of data may lead to partial updates and inconsistency. In the meantime, the persist order of data in write-back caches is different from the issue order of store instructions, thus demanding memory barriers to enforce the consistency. To guarantee consistency, recent CPUs provide instructions for cache line flushes (e.g., *clflush*, *clflushopt*, and *clwb*) and memory barriers (e.g., *sfence*, *lfence*, and *mfence*) [1]. With these instructions, users can use logging or CoW to guarantee crash consistency for data larger than 8 bytes [27, 38]. However, logging and CoW generate extra writes causing the overheads for PM applications [30,40]. In our implementation, we use the interface provided by PMDK [5], which issues clwb and sfence instructions in our machine, to persist the data into PM.

2.2 Lock-free Concurrency Control

Compare-And-Swap (CAS) primitives and CoW have been widely used in existing lock-free algorithms for atomicity. The CAS primitive compares the stored contents with the expected contents. If the contents match, the stored contents are swapped with new values. Otherwise, the expected contents are updated with the stored contents (or just do nothing). The execution of CAS primitives is guaranteed to be atomic, thus avoiding the use of locks. CAS primitives are used in concurrent index structures to provide high scalability [21,34]. However, CAS primitives don't support data sizes larger than the CPU write unit size (e.g., 8 bytes). CoW is used to atomically update data larger than 8 bytes [30]. CoW first copies the data to be modified and performs in-place update 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. In PM, frequent use of CoW causes severe performance degradation [30, 40]. In our clevel hashing, we design the lock-free algorithms using CAS primitives for most writes and lightweight CoW for infrequent metadata updates, thus achieving high scalability with limited extra PM writes.

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

2.3 Basic Hash Tables

Unlike tree-based index structures, hashing-based indexes store the inserted items in flat structures, e.g., an array, thus obtaining $O(1)$ point query performance. Some hashing schemes, e.g., CLHT [17], store key-value items in the hash table, which mitigates the cache line accesses. However, such design doesn't support the storage of variable-length key-value items. Reserving large space in hash tables causes heavy space overheads, since most key-value pairs in real-world scenarios are smaller than a few hundreds of bytes [10, 18]. In order to efficiently support variable-length key-value items, many open-source key-value stores (e.g., redis [7], memcached [4], libcuckoo [28], and the cmap engine in pmemkv [6]) store pointers in hash tables and actual key-value items out of the table. In our clevel hashing, we store pointers in hash tables to support variable-length key-value items.

A typical hash table leverages hash functions to calculate the index of a key-value item. Different key-value items can be indexed to the same storage position, called *hash collisions*. Existing hashing schemes leverage some techniques to address hash collisions, e.g., linear probing [30], multi-slot buckets [18, 28, 30, 40], linked list [6, 16, 27, 29], and data relocation [18, 28, 34, 40]. If hash collisions cannot be addressed, the hash table needs to be resized to increase the capacity. Typical resizing operations consist of three steps: (1) Allocate a new hash table with $2\times$ as many buckets as the old table. (2) Rehash items from the old table to the new table. (3) When all items in the old table have been rehashed, switch to the new table. The resizing in conventional hashing schemes involves intensive data movements [40] and blocks concurrent queries [30].

2.4 Hashing-based Index Structures for PM

Recently, researchers have proposed several hashing-based indexes for PM [6, 16, 30, 40]. Different from DRAM indexes, PM indexes need to remain consistent after system failures. However, the write bandwidth of PM is one sixth as much as DRAM [23, 24, 36], which indicates the significance of write efficiency for concurrent PM hashing indexes.

2.4.1 The Level Hashing Scheme

Our clevel hashing is based on the level hashing index structure [40]. Level hashing has three goals: low-overhead crash consistency, write efficiency, and resizing efficiency. Below, we briefly introduce the relevant components in level hashing.

Level hashing has two levels and the top level has twice the buckets of the bottom level. Each *level* is an array of 4-slot buckets. Besides the 4 slots, a bucket has 4 tokens and each token is one bit corresponding to one slot for crash

Table 1: The Comparisons of Our Clevel Hashing with State-of-the-art Concurrent Resizable Hashing Indexes for PM. *(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, and "CLEVEL" is our clevel hashing. For the memory efficiency and crash consistency, "✓" and "-" indicate good and moderate performance, respectively.)*

	Concurrency Control			Correctness Guarantee		Memory Efficiency	Crash Consistency
	Search	*Insertion/Update/Deletion*	*Resizing*	*Duplication*	*Missing*		
LEVEL	Slot lock	Slot lock	Global metadata lock	No	No	✓	✓
CCEH	Segment reader lock	Segment writer lock	Global directory lock	No	Yes	-	✓
CMAP	Bucket reader lock	Bucket writer lock	Bucket writer lock + lazy rehashing	Yes	Yes	✓	✓
P-CLHT	Lock-free	Bucket lock	Global metadata lock	Yes	Yes	✓	✓
CLEVEL	Lock-free	Lock-free	Asynchronous	Yes	Yes	✓	✓

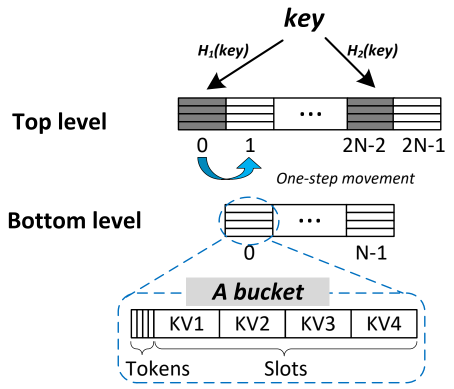


Figure 1: The level hashing index structure

consistency. The overview of level hashing index structure is shown in Figure 1.

By using two independent hash functions, each item has two candidate buckets in one level for storage (16 slots in total for two levels). When the two buckets are full, level hashing tries to perform one-step movement to obtain an empty slot for the item to be inserted. For example, the key in Figure 1 has two candidate buckets in the top level: the first bucket and the second to last bucket. If the two buckets are full and one stored item in the first bucket has empty slots in its alternative candidate bucket (e.g., the second bucket), the stored item is moved to the second bucket so that the key can be inserted to the first bucket. The one-step movement in level hashing improves the maximal load factor before resizing by 10% [40].

For resizing, level hashing creates a new level with 2× (e.g., 4N in Figure 1) as many buckets as the top level and migrates stored items in the bottom level to the new level. The items in the top level are reused without rehashing.

Level hashing uses slot-grained locks for concurrent queries. A fine-grained slot lock is acquired before accessing the corresponding slot and released after completing the access. For resizing, level hashing rehashes items using one thread and blocks concurrent queries of other threads. The concurrency control in level hashing has two correctness problems:

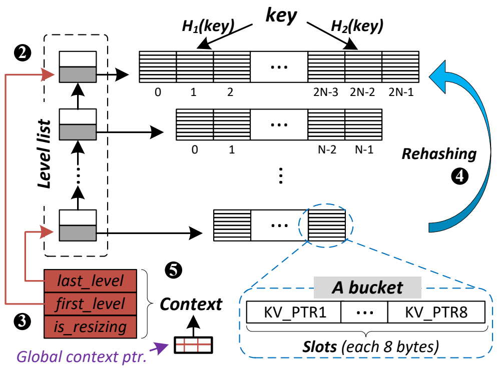
1) Duplicate items. An insertion thread with a single slot lock for an item cannot prevent other threads from inserting items with the same key into other candidate positions, since one item has 16 slots (2 candidate buckets for each level) for storage. Duplicate items in the hash table violate the correctness for updates and deletions: one thread updates or deletes one 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 slot lock cannot stop the movements. As a result, one query with a slot lock may miss inserted items due to concurrent moving of other threads.

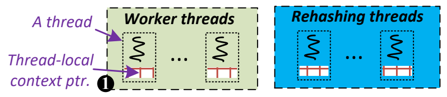
2.4.2 Concurrent Hashing Indexes for PM

Recent schemes design some crash-consistent PM hashing indexes with lock-based concurrency control. CCEH [30] organizes 1024 slots as a segment for dynamic hashing. For concurrent execution, the segment splitting during insertion requires an exclusive writer lock for the segment. Moreover, when the number of segments reaches a predefined threshold, a global directory of segments needs to be doubled with a global directory lock. In pmemkv [6], there is a concurrent linked-list based hashing engine for PM, called cmap, which uses bucket-grained reader-writer locks for concurrency control. The cmap leverages lazy rehashing to amortize data migration in future queries. However, the aggregation of rehashing in the critical path of queries leads to uncertainty and increases the tail latency. P-CLHT [27] is a crash-consistent version of CLHT [17], a cache-efficient hash table with lock-free search. However, when P-CLHT starts resizing, concurrent insertions to the stale buckets (i.e., buckets whose items have been rehashed) have to wait until the resizing completes.

As shown in Table 1, we summarize state-of-the-art concurrent hashing-based index structures with resizing support for PM and compare our clevel hashing with them. All comparable schemes are the open-source versions with default parameter settings. For concurrent queries, CCEH uses coarse segment reader-writer locks, while level hashing and cmap adopt fine-grained locks. P-CLHT leverages bucket-grained



(a) The index structure shared by all threads.



(b) The local context for each thread.

Figure 2: The clevel hashing index overview.

locks for insertion/update/deletion and provides lock-free search. In terms of resizing, level hashing, CCEH, and P-CLHT suffer from the global locks. Though cmap avoids expensive global locks for resizing, the lazy rehashing is in the critical path of queries and affects the scalability. For concurrency correctness, as discussed in §2.4.1, level hashing suffers from duplicate items and missing inserted items. Since CCEH doesn't 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. Unlike existing schemes, our clevel hashing achieves lock-free queries with asynchronous background resizing while guarantees the concurrency correctness and memory efficiency.

3 The Clevel Hashing Design

Our proposed clevel hashing leverages flexible data structures and lock-free concurrency control mechanism to mitigate the competition for the shared resources and improve the scalability. The design of clevel hashing aims to address the three problems in hashing index structures for PM: (1) ***How to support concurrent resizing operations without blocking the queries in other threads?*** (2) ***How to avoid lock contention in concurrent execution?*** (3) ***How to guarantee crash consistency with low overheads?*** In this Section, we first illustrate the dynamic multi-level structure (§3.1), which provides high memory efficiency and supports the low-cost resizing operation. We further present the lock-free concurrency control and correctness guarantee in the clevel hashing (§3.2), i.e., lock-free search/insertion/update/deletion. We finally discuss crash recovery (§3.3).

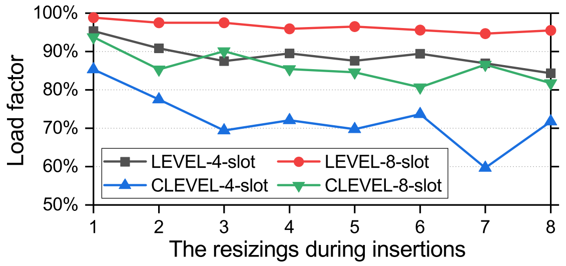


Figure 3: The load factors of level hashing and clevel hashing with different slots per bucket when the resizing occurs.

3.1 The Clevel Hashing Index Structure

3.1.1 Dynamic Multi-level Structure

The global index structure of clevel hashing shared by all threads is shown in Figure 2(a). The hash table in the clevel hashing consists of several levels and each level is an array of buckets. All these levels are organized by a linked list, called *level list*. For two adjacent levels, the upper level has 2× as many buckets as the lower one. 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. Each key-value item is mapped to two candidate buckets in each level via two hash functions. To guarantee high storage utilization, clevel hashing maintains at least two levels [40]. Unlike the 4-slot bucket in the level hashing [40], each bucket in clevel hashing has 8 slots. Each slot consists of 8 bytes and stores a pointer to a key-value item. The actual key-value item is stored outside of the table via dynamic memory allocation. Hence, the 8-slot bucket is 64 bytes and fits the cache line size. By only storing the pointers to key-value items in slots, clevel hashing supports variable-length key-value items. Hence, the content in each slot can be modified using atomic primitives. The atomic visibility of pointers is one of the building blocks for lock-free concurrency control (§3.2) in our clevel hashing.

Different from the level hashing, our clevel hashing index structure is write-optimal for insertion while maintaining high storage utilization. The level hashing tries to perform one movement (one-step movement) for inserted items by copying them into their second candidate bucket in the same level, which causes one extra write for PM. For clevel hashing, the one-step movement is skipped, which decreases the storage utilization. Figure 3 shows the load factors of level hashing and clevel hashing with different slot numbers when successive resizings occur during insertion. Compared with level hashing having the same number of slots per bucket, the load factor of clevel hashing becomes lower due to the lack of one-step movement. However, 8-slot buckets in clevel hashing increase the number of candidate slots for a key in one level, thus achieving a comparable load factor (80%) than the level hashing with 4-slot buckets (default configuration). The number of slots per bucket also affects the throughput, which is discussed in §4.2.

3.1.2 The Support for Concurrent Resizing

Clevel hashing leverages the dynamic multi-level design and context to support concurrent resizing operations. The number of levels in clevel hashing is dynamic: levels are added for resizing and removed when rehashing completes. The *context* in clevel hashing is interpreted as an object containing two level list nodes and the `is_resizing` flag. The two nodes point to the first and last levels while the flag denotes if the table is being resized. There is a global pointer to the context (Figure 2(a)) and each thread maintains a thread-local copy of the context pointer (Figure 2(b)). Hence, the context can be atomically updated using CoW + CAS. Since the context size is 17 bytes (i.e., two pointers and one Boolean flag) and the context changes only when we add/remove a level, the CoW overheads of context are negligible for PM.

The resizing operation in clevel hashing updates the level list and the context. Specifically, when hash collisions can't be addressed, the resizing is performed in the following steps: (Step ❶) Make a local copy of the global pointer to context. (Step ❷) Dereference the local copy of context pointer, and append a new level with twice the buckets of the original first level to the level list using CAS. If the CAS fails, update the local copy of the context pointer and continue with the next step, since other threads have successfully added a new level. (Step ❸) Use CoW + CAS to update the global context by changing the first level to the new level (e.g., L_{new}) and setting `is_resizing` to true. When the CAS fails, check if the new first level's capacity is no smaller than L_{new} and the `is_resizing` is true: if so, update the local context pointer and continue; otherwise, retry the CoW + CAS with the `is_resizing` (true) and optional new level L_{new} (if the first level's capacity is smaller than L_{new}). (Step ❹) Rehash each item in the last level. The rehashing includes two steps: copy the item's pointer to a candidate bucket in the first level via CAS, and delete the pointer in the last level (without CAS, and the correctness for possible duplicate items is guaranteed in insertion §3.2.2 and update §3.2.3). If the CAS fails, find another empty slot. If no empty slot is found, go to step ❷ to expand the table. (Step ❺) When rehashing completes, update the last level and optional `is_resizing` (if only two levels remain after resizing) in the global context atomically using CoW + CAS. If the CAS fails, try again if the last level in current context is unmodified. The resizing workflow is shown in Figure 2. Note that the reasons for three possible CAS failures in resizing are different: the CAS failures in step ❷ come from the concurrent expansion of other threads, i.e., the step ❷ in other threads; the failures in steps ❸ and ❺ are due to the concurrent execution of these two steps (steps ❸ and ❺) in different threads. As a result, the strategies for corresponding CAS failures are different as presented above.

To mitigate the insertion performance degradation due to resizing, clevel hashing leverages background threads to enable asynchronous rehashing. Specifically, we divide the resizing into two stages, including *expansion* (steps ❶, ❷, and ❸) and *rehashing* (steps ❹ and ❺) *stages*. The time-consuming rehashing stage is offloaded into background threads, called *rehashing threads*. Rehashing threads continuously rehash items until there are two levels left. Therefore, when the table is not being resized, the queries in clevel hashing guarantee constant-scale time complexity. The threads serving query requests are called *worker threads*. When the hash collisions can not be addressed by worker threads, they perform the three steps in the expansion stage and then continue the queries. Since the main operations for table expansion are simple memory allocation and lightweight CoW for context (17 bytes), the expansion overheads are low. Moreover, there is no contention for locks during expansion. As a result, the resizing operation no longer blocks queries.

Rehashing performance can be improved by using multiple rehashing threads. To avoid contention for rehashing, a simple modular function is used to partition buckets into independent batches for rehashing threads. For example, if there are two rehashing threads, one thread rehashes odd-number buckets while the other rehashes even-number buckets. After both rehashing threads finish, they update the global context following step ❺.

3.2 Lock-free Concurrency Control

In order to mitigate the contention for shared resources, we propose lock-free algorithms for all queries, i.e., search, insertion, update, and deletion.

3.2.1 Search

The search operation needs to iteratively check possible buckets to find the first key-value item that matches the key. There are two main problems for lock-free search in the clevel hashing: (1) *High read latency for the pointer dereference costs.* Since clevel hashing only stores the pointers in hash tables to support variable-length items, dereferencing is needed to fetch the corresponding key, which results in high cache miss ratios and extra PM reads. (2) *Missing inserted items due to the data movement.* The concurrent resizing moves the items in the last level, and therefore, searching without any locks may miss inserted items.

For the pointer dereference overheads, our proposed clevel hashing leverages a summary tag to avoid the unnecessary reads for full keys. A tag is the summary for a full key, e.g., the leading two bytes of the key's hash value. The hash value is obtained when calculating the candidate buckets via hash functions (e.g., the `std::hash` from C++ [8]). The tag technique is inspired from MemC3 [18] and we add atomicity for the pair of tag and pointer. For each inserted item, its tag is stored in the table. For a search request, only when the tag of a request matches the stored tag of an item, we fetch the stored full key by pointer dereferencing and compare the two keys. A *false positive* case for tags appears when different keys have the same tag. For 16-bit tags, the false positive rate is $1/2^{16}$. Since we check full-size keys when two tags match, the false positives can be identified and will not cause any problem of correctness. Instead of allocating additional space for tags in MemC3, clevel hashing leverages the unused 16 highest bits in pointers to store the tags. Current pointers only consume 48 bits on x86_64, thus leaving 16 bits unused in 64-bit pointers [31,35]. The reuse of reserved bits enables the atomic updates of pointers and tags.

To address the problem of missing inserted items, we propose to search from the last level to the first level, called *bottom-to-top* (*b2t*) search strategy. The intuition behind b2t searching is to follow the direction of bottom-to-top data movement in hash table expansion, which moves items from the last level to the first level. However, a rare case for missing is: after a search operation starts, other threads add a new level through expansion and rehashing threads move the item that matches the key of the search to the new level. To fix this missing, clevel hashing leverages the atomicity of context. Specifically, when no matched item is found after b2t search, clevel hashing checks the global context pointer with the previous local copy. If the two pointers are different, redo the search. The overheads for the re-execution of search are low, since changes of the context pointer are rare, occurring only when a level is added to or removed from the level list. Therefore, the correctness for the lock-free search is guaranteed with low costs.

3.2.2 Insertion

For insertion, a key-value item is inserted if the key does not exist in the table. The insertion first executes lock-free b2t search (§3.2.1) to determine if an item with the same key exists. If none exists, the pointer (with its summary tag) to the key-value item is atomically inserted into a less-loaded candidate bucket. When there is no empty slot, we resize the table by adding a new level with background rehashing (§3.1.2) and redo the insertion. However, lock-free insertion leads to two correctness problems: (1) *Duplicate items from concurrent insertions*. Without locks, concurrent threads may insert items with the same key into different slots, which results in failures for update and deletion. (2) *Loss of new items inserted to the last level*. When new items are inserted into the buckets in the last level that have been processed by rehashing threads, these inserted items are lost after we reclaim the last level.

For concurrent insertions to different slots, it is challenging to avoid the duplication in a lock-free manner, since atomic primitives only guarantee the atomicity of 8 bytes. However, each one of the duplicate items is correct. Hence, we fix the duplication in future updates(§3.2.3) and deletions(§3.2.4).

In order to fix the loss of new items, we design a context-aware insertion scheme to guarantee the correctness of insertion. The context-aware scheme includes two strategies: (1) Before the insertion, we check the global context and do not insert items into the last level when the hash table is resizing, i.e., when the `is_resizing` is true. (2) After the insertion, if the table starts resizing and the item has been already inserted into the last level, we redo the insertion using the same pointer without checking duplicate items. The re-execution of insertion leads to possible duplicate pointers in the hash table. However, duplicate pointers don't affect the correctness of search, because they refer to the same key-value items. Future updates and deletions are able to detect and address the duplication.

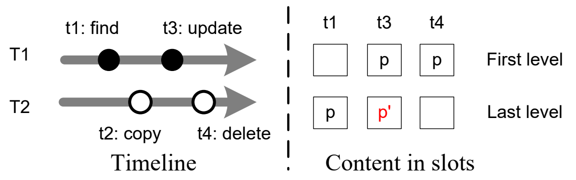


Figure 4: The update failure. (*"T1": an update thread, "T2": a rehashing thread, "t1-t4": timestamps, "p": pointer to the old item, "p′": pointer to the updated item.*)

3.2.3 Update

The update operation in the clevel hashing atomically updates the pointers to the matched key-value items. Different from the insertion, the update needs to fix duplicate items. Otherwise, duplicate items may lead to inconsistency after being updated. Moreover, the concurrent executions of resizing and update may cause update failures due to the data movement for rehashing. This section focuses on our solutions for the two correctness problems.

1) Content-conscious Find to Fix Duplicate Items. There are three cases for duplicate items in our clevel hashing: concurrent insertion with the same key, the retry of the context-aware insertion, and data movement for rehashing. Note that re-insertion after system crash would not generate duplication due to checking of the key before insertion. We observe that duplication from two concurrent insertions leads to two pointers to different items and keeping any one of the two is acceptable. Re-insertion or rehashing generates two pointers to the same item. In this case, we keep the pointer which 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, keeping either pointer is identical. With this knowledge, we design a content-conscious Find process to handle duplication in two steps. First, we apply b2t search to find two slots storing the pointer to the matched key. Second, if two pointers refer to different locations, we delete the item and the pointer that first occurs in the b2t search. If two pointers point to the same location, we simply delete the pointer that first occurs. By removing duplicate items, the Find process returns at most one item for the atomic update.

2) Rehashing-aware Scheme to Avoid Update Failures. As the example shown in Figure 4, even with the Find process, the interleaved execution of update and rehashing is possible to lose the updated values. The updated item referred by p*ʹ* is deleted by the rehashing thread. A straightforward solution is to issue another Find process after the update. However, frequent two-round Find increases the update latency. To decrease the frequency, we design a rehashing-aware scheme by checking the rehashing progress. Specifically, before the first Find process, we record the bucket index (e.g., RB_{idx}) being rehashed by one rehashing thread. After the atomic update, we read the rehashing progress again (e.g., RB'_{idx}). If the global context doesn't change, an additional Find is triggered only when meeting the following constraints: (1) the table is during resizing; (2) the updated bucket is in the last level; (3) the updated bucket index B_{idx} satisfies $RB_{idx} \leq B_{idx} \leq RB'_{idx}$ for all rehashing threads. Since it's a rare case that three constraints are simultaneously satisfied, our rehashing-aware update scheme guarantees the correctness with low overheads.

3.2.4 Deletion

For deletion, clevel hashing atomically deletes the pointers and matched items. Like the update, the deletion also needs to remove duplicate items. Compared with the update, we optimize the scheme to handle duplication in deletion. Briefly speaking, clevel hashing deletes all the matched items through the b2t search. The lock-free deletion also has failures like the lock-free update. Hence, we use the rehashing-aware scheme presented in lock-free update(§3.2.3) to guarantee the correctness.

3.3 Recovery

The fast recovery requires the guarantee for crash consistency, which is nontrivial for PM. Recent studies [16, 35] show that a crash-consistent lock-free PM index needs to persist after stores and not modify PM until all dependent contents are persisted. The crash consistency guarantee in clevel hashing follows this methodology. Specifically, clevel hashing adds cache line flushes and memory fences after each store and persists dependent metadata, e.g., the global context pointer, after the load in insertion/update/deletion. The persist overheads for crash consistency can be further optimized by persisting in batches [16].

For the crash consistency in rehashing, clevel hashing records the index of bucket (e.g., RB_{idx}) in PM after successfully rehashing the items in the bucket. To recover from failures, rehashing threads read the context and bucket index and continue the rehashing with the next bucket (e.g., $RB_{idx} + n$, n is the number of rehashing threads). A crash during the data movement of rehashing may lead to duplicate items, which are fixed in future update (§3.2.3) and deletion (§3.2.4).

To avoid permanent memory leakage [16], we leverage existing PM atomic allocators from PMDK [5] and design lock-free persistent buffers for secure and efficient memory management. The PM atomic allocators atomically allocate and reclaim memory to avoid expensive transactional memory management [5]. A persistent buffer is a global array of persistent pointers (used by the PM allocators for atomic and durable memory management) attached to the root object (an anchor) of the persistent memory pool. The array size is equal to the thread number. Each thread uses the persistent pointer corresponding to its thread ID. Hence, there is no contention for the persistent buffers. When recovering from failures, we scan the persistent buffers and release the unused memory.

4 Performance Evaluation

4.1 Experimental Setup

Our experiments run on a server equipped with six Intel Optane DC PMM (1.5 TB in total), 128 GB DRAM, and 24.75 MB L3 cache. The Optane DC PMMs are configured in the *App Direct* mode and mounted with ext4-DAX file system. There are 2 CPU sockets (i.e., NUMA nodes) in the server and each socket has 36 threads. For a processor in one NUMA node, the latency of accessing local memory (attached to the NUMA node) is lower than non-local memory [13, 25, 33]. Conducting experiments across multiple NUMA nodes introduces the disturbance of non-uniform memory latencies. To avoid the impact of NUMA architectures, we perform all the experiments on one CPU socket by pinning threads to an NUMA node for all schemes, like RECIPE [27]. Existing NUMA optimizations, e.g., Node-Replication [13] to maintain per-node replicas of hash tables and synchronize these replicas through a shared log, are possible to improve the scalability with more NUMA nodes.

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

- **LEVEL**: This is the original concurrent level hashing [40] with consistency support. The level hashing uses slot-grained reader-writer lock for queries and a global resizing lock for resizing.

- **CCEH**: CCEH [30] organizes an array of slots as a segment (e.g., 1024 slots) and uses a directory as an address table. Linear probing (e.g., 16 slots) is used to improve the load factor. CCEH supports dynamic resizing through segment splitting and possible directory doubling. We adopt the default lazy deletion version since it has higher insertion throughput than the CoW version. CCEH uses reader-writer locks for segments and the directory.

- **CMAP**: The concurrent_hash_map storage engine in pmemkv [6] is a linked list based concurrent hashing scheme for PM. It uses reader-writer locks for concurrent accesses to buckets and supports lazy rehashing (rehash the buckets in a linked list when accessing).

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

- **P-CLHT**: P-CLHT [27] is a linked list based cache-efficient hash table. Each bucket has 3 slots. P-CLHT supports lock-free search while the bucket-grained lock is needed for insertion and deletion. The resizing in P-CLHT requires a global lock. When one thread starts rehashing, another helper thread (one helper at most) is allowed to perform concurrent rehashing, which is called helping resizing. The helping resizing mechanism is enabled by default.
- **CLEVEL**: This is our proposed scheme, clevel hashing, which provides asynchronous resizing and lock-free concurrency control for all queries with high memory efficiency.

Since open-source cmap is implemented using PMDK with C++ bindings [5], we implement our clevel hashing with PMDK (version 1.6) and port level hashing, CCEH, and P-CLHT to the same platform for fair comparisons. Like cmap and clevel hashing, we optimize level hashing, CCEH, and P-CLHT to support variable-length items by storing pointers in the hash table. For level hashing and CCEH, we use the same type of reader/writer locks from cmap to avoid the disturbance of lock implementations. During the porting, in addition to the reported bugs of the inconsistencies in directory metadata [27], we observe a concurrent bug for the directory in original CCEH: a thread performing search can access a directory deleted by other threads that are doubling the directory. As a result, failures may occur in search when accessing the reclaimed directory via pointer dereferencing. To ensure the correctness and avoid such failures in experiments, we add the missing reader lock for the directory. The hash functions for all schemes are the same: the `std::hash` from the C++ Standard Template Library (STL) [8]. In addition to conventional locks, we also evaluate the performance of level hashing, CCEH, and cmap with the spinlocks from Intel TBB library [3]. For abbreviation, *LEVEL-TBB*, *CCEH-TBB*, and *CMAP-TBB* are TBB-enabled.

We use YCSB [15] to generate micro-benchmarks in zipfian distribution with default 0.99 skewness [40] to evaluate the throughput of different slot numbers and latencies of different queries. The results using uniformly distributed workloads are similar due to the randomness of hash functions [30]. Different queries are executed in the micro-benchmarks: insertion (unique keys), positive search (queried keys exist), negative search (queried keys not exist), update, and deletion. The items to be updated or deleted are present in the table. To evaluate the concurrent throughput, we leverage the real-world workloads from YCSB as macro-benchmarks, following RECIPE [27]. The workload patterns are described in Table 2. We initialize all indexes with similar capacity (64 thousand for micro-benchmarks and 256 thousand for macro-benchmarks) and use 15-byte keys and 15-byte values for all experiments. The experiment with YCSB workloads consists of two phases: load and run phases. In the load phase, indexes are populated with 16 million and 64 million items for micro- and macro-benchmarks, respectively. In the run phase, there are 16 million queries for micro-benchmarks and 64 million for macro-benchmarks. For concurrent execution, each scheme has the same number of threads in total. During our evaluation of clevel hashing, we observe that one rehashing thread for 35 insertion threads can guarantee the number of levels is under 4. Hence, we set one thread as the rehashing thread by default. The reported latency and throughput are the average values of 5 runs.

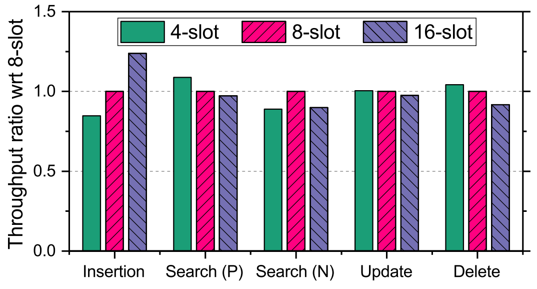


Figure 5: The normalized concurrent throughput of clevel hashing with different slots per bucket. *("Search (P)" is positive search and "Search (N)" is negative search.)*

4.2 Different Slot Numbers and Load Factor

In clevel hashing, the slots per bucket affects not only memory efficiency but also concurrent performance. We run the micro-benchmarks with different slot numbers in clevel hashing and measure the concurrent throughput with 36 threads. The throughput is normalized to that of an 8-slot bucket, as shown in Figure 5. With the increase of slots, the insertion throughput increases. The reason is that more slots per bucket indicate more candidate positions for a key so that it's easier to find an empty slot without resizing. Decreasing the slots per bucket reduces the number of slots to be checked and cache line accesses (a 16-slot bucket requires two cache lines), thus improving the search, update, and deletion throughputs. According to the results shown in Figure 5, 8-slot bucket is a trade-off between 4-slot and 16-slot buckets. Therefore, we set the slot number to 8.

In order to evaluate the memory efficiency of different schemes, we use an insert-only workload to record the load factor (the number of inserted items divided by the number of slots in the table) after every 10K insertions. Since the slot in cmap is allocated on demand for insertions, the load factor is always 100%. The load factors of the other schemes

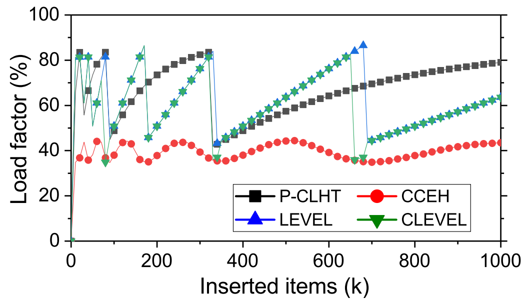


Figure 6: The load factor per 10K insertions. *(The even symbols are skipped for clearness.)*

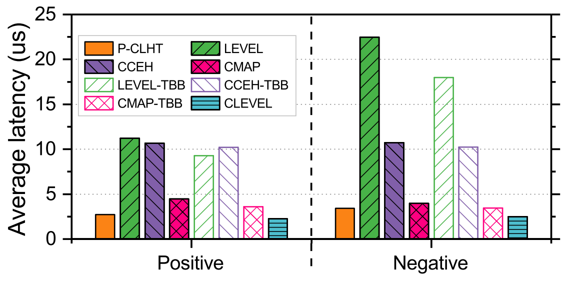


Figure 7: The average latency for concurrent search.

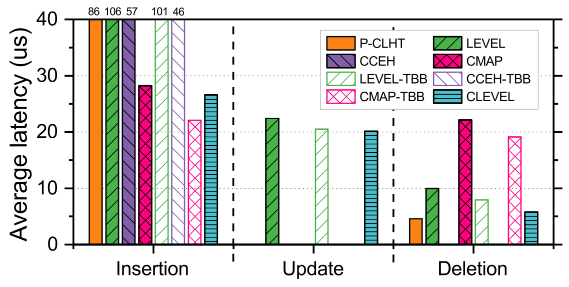


Figure 8: The average latencies for concurrent insertion, update, and deletion.

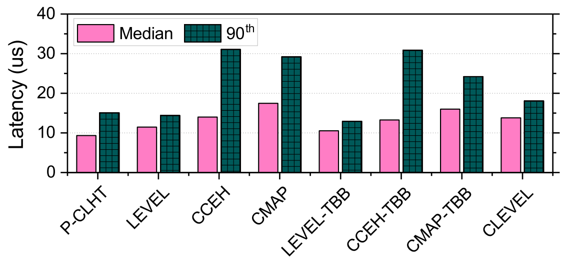


Figure 9: The median and 90^{th} percentile latencies for concurrent insertion.

are shown in Figure 6. The maximal load factor of CCEH is no more than 45%, because CCEH probes only 16 slots to address the hash collisions. Though CCEH is able to increase the linear probing distance for higher memory efficiency, long probing distance leads to more memory accesses and pointer dereferencing, thus decreasing the throughputs for all queries. P-CLHT resizes when the number of inserted items approaches the initial capacity of current hash table. By using the three-slot bucket with linked list, the load factor of P-CLHT is up to 84%. Compared with level hashing, clevel hashing doesn't move items in the same level. However, clevel hashing increases the number of slots per bucket to 8. As a result, the maximal load factor of clevel hashing is comparable with original level hashing, i.e., 86%.

4.3 Micro-benchmarks

We use the micro-benchmarks to evaluate the average query latencies in different PM hashing indexes. The latency of a query is interpreted as the time for executing the query, not including the time waiting for execution. All experiments run with 36 threads. Note that the latencies of micro-benchmarks for search, update, and deletion demonstrate the performance of corresponding queries without the impact on resizing, since there is no insertion in these workloads. For the insert-only workload, the expansion of hash table occurs in the run phase.

For the concurrent search, we measure the average latencies when all keys exist (positive search) or not (negative search) in the table. As shown in Figure 7, the level hashing suffers from frequent locking and unlocking of candidate slots, especially for the negative search, i.e., 16 slot locks in total for 4 candidate 4-slot buckets. The coarse-grained segment lock (1024 slots) in CCEH leads to high search latency. The search in cmap only requires one bucket lock and ensures low latency. Due to the lock-free search, P-CLHT achieves lower latency than lock-based indexes. For clevel hashing, there are only two levels when the table is not resizing, thus ensuring the number of candidate buckets to be checked is at most 4. Moreover, the lock-free search avoids the contention for buckets. Tags filter unnecessary retrievals for keys. As a result, clevel hashing achieves $1.2\times-5.0\times$ speedup for positive search latency and $1.4\times-9.0\times$ speedup for negative search latency, compared with other PM hashing indexes.

The average latencies for insertion/update/deletion are shown in Figure 8. Some bars are missing because the corresponding schemes haven't implemented update (i.e., P-CLHT, cmap, and CCEH) or deletion (i.e., CCEH) in their open-source code.

Insertion: During insertion, all schemes have to expand to accommodate 16 million items. The resizing may block several requests (the number depends on the resizing times and thread numbers) and significantly increase their execution time, thus increasing the average latencies. P-CLHT, level hashing, and CCEH suffer from the global lock for resizing. By amortizing the rehashing over future queries, cmap achieves low average latency for insertions. The average latency of clevel hashing is slightly higher than the cmap with TBB because the expansion needs to durably allocate a large new level via the persistent allocator from PMDK, which is achieved by expensive undo logging [38].

Figure 9 shows the median and 90^{th} percentile insertion

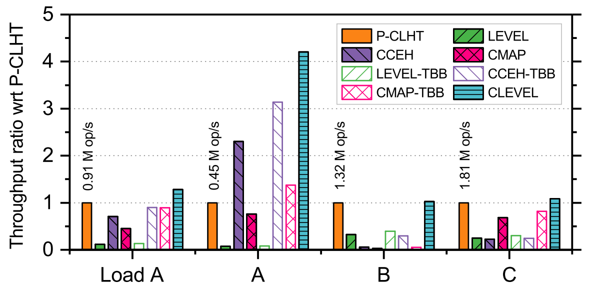


Figure 10: The concurrent throughput of YCSB normalized to P-CLHT.

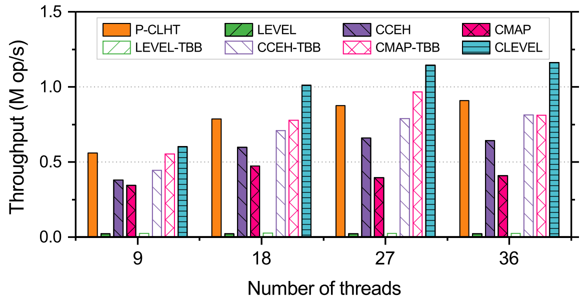


Figure 11: The insertion scalability.

latencies. Unlike the average latency, median and 90th percentile latencies demonstrate the insertion performance without the impact of resizing. The reason is that the ratio of insertions which encounter resizing during their executions is less than 10%. In the meantime, the queuing time for execution is not included in the latency. CCEH suffers from high percentile latencies due to the coarse-grained segment lock. Though cmap leverages fine-grained bucket locks, the amortized rehashing in queries increases the insertion time. Due to the context-aware insertion for correctness guarantee, the median and 90th percentile latencies of clevel hashing are slightly higher than level hashing and P-CLHT but lower than CCEH and cmap.

Update: Compared with original level hashing, clevel hashing obtains slightly lower update latency. The reason is that the benefits of lock-free update compensate for the overheads of correctness guarantee in the clevel hashing, e.g., additional Find operation for duplicate items and checking for update failures.

Deletion: The deletion latency in cmap is higher than other schemes due to rehashing the bucket if necessary before accessing. Level hashing has higher deletion latency than P-CLHT and clevel hashing due to the frequent locking and unlocking when accessing candidate slots. To fix duplication during deletion, clevel hashing checks all candidate slots, thus resulting in a slightly higher latency than P-CLHT.

4.4 Macro-benchmarks

Figure 10 shows the concurrent throughput normalized to P-CLHT of different PM hashing schemes with real-world workloads from YCSB. We run the experiment with 36 threads for all schemes. Since workload Load A is used to populate indexes with 64 million items in the load phase, all indexes resize multiple times (e.g., more than 10 times in clevel hashing) from small sizes. Resizing also occurs in the workload A.

The locks used for concurrency control hinder the index performance. Specifically, the global resizing lock in the level hashing blocks all queries until the single-threaded rehashing completes, which leads to low throughput in workload Load A and A. The global directory lock in CCEH is only used for directory doubling. Therefore, the insertion throughput is much higher than level hashing. Due to the helping mechanism in P-CLHT, there are two threads concurrently rehashing items, which mitigates the overheads of the global resizing lock in the load phase of YCSB (Load A). However, when the table size increases, the two threads are not enough to rehash all items in a short time, which accounts for the low throughput for P-CLHT in workload A. Due to the multiple resizing, the aggregated rehashing hinders the throughput of cmap. Unlike these lock-based indexes, the lock-free concurrency control in clevel hashing avoids the lock contention during insertions. Hence, clevel hashing obtains 1.4× speedup than cmap for insertion throughput. In summary, our clevel hashing achieves up to 4.2× speedup than the state-of-the-art PM hashing index (i.e., P-CLHT).

To evaluate the scalability of clevel hashing, we measure the insertion throughput with different number of threads using the Load A. As shown in Figure 11, with the increase of threads, the throughput of clevel hashing increases and is consistently higher than other schemes. This trend in search throughput is similar.

4.5 Discussion

The reduction of hash table size. The current design of clevel hashing doesn't support the reduction of the hash table size. When most stored items in the hash table are deleted, the table may reduce the table size to improve space utilization. Clevel hashing needs to be adapted to support the reduction. 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. When all the items of the first level are rehashed, the first level is reclaimed. The migration of items for the reduction generates data movement from the top level to the bottom one (i.e., top-to-bottom movement), which is opposite to expansion (i.e., bottom-to-top movement). Therefore, to support concurrent reduction, we carry out top-down search strategy instead of down-top search (§3.2.1) to avoid missing inserted items. Note that all threads need to leverage the same search strategy: either top-down searching for reduction or down-top searching for expansion. The clevel hashing with non-blocking concurrent reduction is our future work.

The isolation level. For the isolation in transactions, clevel hashing has dirty reads, since there is no lock to isolate data. Hence, the isolation level is *read uncommitted* [11]. To support higher isolation levels in a transaction, additional locks or version control schemes are required [11, 22].

Space overhead. The metadata overhead in clevel hashing mainly comes from the persistent buffers (§3.3), which are the arrays of persistent pointers to the allocated memory, for efficient management without contention. Persistent buffers have separate entries for each thread. Therefore, the metadata overhead is proportional to the number of threads. Clevel hashing achieves a maximal load factor over 80% before resizing. During resizing, hashing collisions that cannot be addressed increase the number of levels to more than 3. Due to the randomness of hash functions, the possibility of continuous hash collisions for one position is very low. Moreover, rehashing threads migrate items in the last level until only two levels remain. Hence, the number of total levels is usually small, which enables high storage utilization.

5 Related Work

5.1 Hashing-based Index Structures for PM

In order to optimize the hashing performance on PM, recent work have designed some hashing-based index structures for PM. Path hashing [39] and level hashing [40] leverage sharing-based index structures and write-efficient open-addressing techniques for high memory efficiency with limited extra writes. Level hashing introduces a cost-efficient resizing scheme by only rehashing the items in the old bottom level to the new top level, which only account for 1/3 of total inserted items. LF-HT [16] uses lock-free linked list for each bucket to address hash collisions. However, these three schemes all suffer from poor resizing performance, since the resizing operations require exclusive global lock for metadata. CCEH [30] is based on extendible hashing, which dynamically expands the hash table by segment splitting and optional directory doubling. Although the resizing in CCEH is concurrent with other queries, the use of coarse-grained locks for segments or even directory during resizing causes performance degradation. The cmap storage engine in pmemkv [6] supports concurrent lazy rehashing. The rehashing of items in a bucket is trigger when accessing the bucket. As a result, cmap distributes the rehashing of items to future search/insertion/update/deletion. However, cmap is possible to encounter recursive rehashing due to the lazy rehashing. The rehashing in the critical path of queries decreases the throughputs, especially for search operations. Unlike existing schemes, clevel hashing has dynamical multi-level structure for concurrent asynchronous resizing and designs lock-free algorithms to improve the scalability with low latency.

5.2 Lock-free Concurrent Hashing Indexes

Lock-free algorithms mitigate the lock contention for shared resources and are hard to design because of the challenging concurrency control. The lock-free linked list proposed by Harris [21] is widely used in lock-free concurrent hashing indexes [16, 29]. This class of schemes add a lock-free linked list to each bucket. Though the lock-free linked list enables lock-free insertion in these hash tables, it causes high search overheads due to the sequential iteration over linked lists. The lock-free cuckoo hashing [31] uses marking techniques with helping mechanism to support lock-free cuckoo displacements. However, the recursive data movements bring lots of extra PM writes [40]. Recent work [20] uses PSim [19] 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

Persistent memory offers opportunities to improve the performance of storage systems, but suffers from the lack of efficient and concurrent index structures. Existing PM-friendly hashing indexes only focus on the consistency and write reduction, which overlook the concurrency and resizing of hash tables. In this paper, we propose clevel hashing, a lock-free 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 guarantees the correctness. 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

This work was supported in part by National Key Research and Development Program of China under Grant 2016YFB1000202, National Natural Science Foundation of China (NSFC) under Grant No. 61772212 and No. 61821003, and Key Laboratory of Information Storage System, Ministry of Education of China. We are grateful to our shepherd, Andrea C. Arpaci-Dusseau, and the anonymous reviewers for their constructive comments and suggestions. We have released the source code for public use in GitHub.

References

[1] Intel® Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/en-us/isaextensions, 2019.

[2] Intel® Optane™ DC persistent memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html, 2019.

[3] Intel® Threading Building Blocks. https://github.com/intel/tbb, 2019.

[4] Memcached. https://memcached.org/, 2019.

[5] Persistent Memory Development Kit. http://pmem.io/, 2019.

[6] pmemkv. http://pmem.io/pmemkv/index.html, 2019.

[7] Redis. https://redis.io/, 2019.

[8] The C++ Standard Template Library. http://www.cplusplus.com/reference/functional/hash/, 2020.

[9] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment (PVLDB)*, 11(5):553–565, 2018.

[10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, London, United Kingdom, June 2012.

[11] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. *Database Systems - Concepts, Languages and Architectures*. McGraw-Hill Book Company, 1999.

[12] Daniel Bittman, Darrell D. E. Long, Peter Alvaro, and Ethan L. Miller. Optimizing Systems for Byte-Addressable NVM by Reducing Bit Flipping. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*, Boston, MA, February 2019.

[13] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, Xi'an, China, April 2017.

[14] Shimin Chen and Qin Jin. Persistent B+-Trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment (PVLDB)*, 8(7):786–797, 2015.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, Indiana, USA, June 2010.

[16] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (ATC '18)*, Boston, MA, USA, July 2018.

[17] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, Istanbul, Turkey, March 2015.

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

[19] Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-Efficient Wait-Free Synchronization. *Theory Comput. Syst.*, 55(3):475–520, 2014.

[20] Panagiota Fatourou, Nikolaos D. Kallimanis, and Thomas Ropars. An Efficient Wait-free Resizable Hash Table. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA '18)*, Vienna, Austria, July 2018.

[21] Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *15th International Symposium on Distributed Computing (DISC '01)*, Lisbon, Portugal, October 2001.

[22] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, CA, USA, February 2018.

[23] 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. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.

[24] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Huntsville, ON, Canada, October 2019.

[25] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and Practical Locking with Shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Huntsville, ON, Canada, October 2019.

[26] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, USA, February 2017.

[27] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE : Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Huntsville, ON, Canada, October 2019.

[28] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Ninth Eurosys Conference 2014 (EuroSys '14)*, Amsterdam, The Netherlands, April 2014.

[29] Maged M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*, Winnipeg, Manitoba, Canada, August 2002.

[30] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST '19)*, Boston, MA, February 2019.

[31] Nhan Nguyen and Philippas Tsigas. Lock-Free Cuckoo Hashing. In *IEEE 34th International Conference on Distributed Computing Systems (ICDCS '14)*, Madrid, Spain, June 2014.

[32] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*, San Francisco, CA, USA, June 2016.

[33] Zoran Radovic and Erik Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA '03)*, Anaheim, California, USA, February 2003.

[34] Yuanyuan Sun, Yu Hua, Zhangyu Chen, and Yuncheng Guo. Mitigating Asymmetric Read and Write Costs in Cuckoo Hashing for Storage Systems. In *2019 USENIX Annual Technical Conference (ATC '19)*, Renton, WA, USA, July 2019.

[35] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *34th IEEE International Conference on Data Engineering (ICDE '18)*, Paris, France, April 2018.

[36] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, USA, February 2020.

[37] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, CA, USA, February 2015.

[38] Lu Zhang and Steven Swanson. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *2019 USENIX Annual Technical Conference (ATC '19)*, Renton, WA, USA, July 2019.

[39] Pengfei Zuo and Yu Hua. A Write-Friendly and Cache-Optimized Hashing Scheme for Non-Volatile Memory Systems. *IEEE Trans. Parallel Distrib. Syst.*, 29(5):985–998, 2018.

[40] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, USA, October 2018.