Write-Optimized Dynamic Hashing for Persistent Memory

Moohyeon Nam, *UNIST* (*Ulsan National Institute of Science and Technology*); Hokeun Cha, *Sungkyunkwan University;* Young-ri Choi and Sam H. Noh, *UNIST* (*Ulsan National Institute of Science and Technology*); Beomseok Nam, *Sungkyunkwan University*

https://www.usenix.org/conference/fast19/presentation/nam

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

Write-Optimized Dynamic Hashing for Persistent Memory

Moohyeon Nam[†], Hokeun Cha[‡], Young-ri Choi[†], Sam H. Noh[†], Beomseok Nam[‡]
UNIST (Ulsan National Institute of Science and Technology)[†]
Sungkyunkwan University[‡]

Abstract

Low latency storage media such as byte-addressable persistent memory (PM) requires rethinking of various data structures in terms of optimization. One of the main challenges in implementing hash-based indexing structures on PM is how to achieve efficiency by making effective use of cachelines while guaranteeing failure-atomicity for *dynamic hash expansion and shrinkage*. In this paper, we present *Cacheline-Conscious Extendible Hashing* (CCEH) that reduces the overhead of dynamic memory block management while guaranteeing constant hash table lookup time. CCEH guarantees failure-atomicity without making use of explicit logging. Our experiments show that CCEH effectively adapts its size as the demand increases under the fine-grained failure-atomicity constraint and its maximum query latency is an order of magnitude lower compared to the state-of-the-art hashing techniques.

1 Introduction

In the past few years, there have been numerous efforts to leverage the byte-addressability, durability, and high performance of persistent memory (PM) [7, 13, 18, 32, 34, 39, 40, 45, 47]. In particular, latency critical transactions on storage systems can benefit from storing a small number of bytes to persistent memory. The fine-grained unit of data I/O in persistent memory has generated interest in redesigning block-based data structures such as B+-trees [2, 11, 20, 28, 46]. Although a large number of previous studies have improved tree-based indexing structures for byte-addressable persistent memory, only a few have attempted to adapt hash-based indexing structures to persistent memory [48, 49]. One of the main challenges in hash-based indexing for PM is in achieving efficient *dynamic rehashing* under the fine-grained failure-atomicity constraint. In this paper, we present Cacheline-Conscious Extendible Hashing (CCEH), which is a variant of extendible hashing [6] optimized for PM to minimize cacheline accesses and satisfy failure-atomicity without explicit logging.

Due to the static flat structure of hash-based indexes, they can achieve constant lookup time. However, static hashing does not come without limitations. Such traditional hashing schemes must typically estimate the size of hash tables and allocate sufficient buckets in advance. For certain applications, this is a feasible task. For example, in-memory hash tables in key-value stores play a role of fixed-sized buffer cache, i.e., recent key-value records replace old records. Hence, we can set the hash table size a priori based on the available memory space.

However, not all applications can estimate the hash table size in advance, with database systems and file systems being typical examples. If data elements are dynamically inserted and deleted, static fixed-sized hashing schemes suffer from hash collisions, overflows, or under-utilization. To resolve these problems, dynamic resizing must be employed to adjust the hash table size proportional to the number of records. In a typical situation where the load factor (bucket utilization) becomes high, a larger hash table must be created, and a *rehash* that moves existing records to new bucket locations must be performed.

Unfortunately, rehashing is not desirable as it degrades system throughput as the index is prevented from being accessed during rehashing, which significantly increases the tail latency of queries. To mitigate the rehashing overhead, various optimization techniques, such as *linear probing*, *separate chaining*, and *cuckoo hashing*, have been developed to handle hash collisions [4, 14, 25, 29, 31]. However, these optimizations do not address the root cause of hash collisions but defer the rehashing problem. As such, static hashing schemes have no choice but to perform expensive full-table (or 1/3-table [49]) rehash operations later if the allocated hash table size is not sufficient.

In light of PM, rehashing requires a large number of writes to persistent memory. As writes are expected to induce higher latency and higher energy consumption in PM, this further aggravates performance. Furthermore, with lifetime of PM expected to be shorter than DRAM, such extra writes can be detrimental to systems employing PM.

Unlike these static hashing schemes, *extendible hashing* [6] dynamically allocates and deallocates memory space on demand as in tree-structured indexes. In file systems, extendible hash tables and tree-structured indexes such as B-trees are used because of their dynamic expansion and shrinkage capabilities. For example, extendible hashing is used in Oracle ZFS [26], IBM GPFS [30, 33], Redhat GFS, and GFS2 file systems [38, 44], while tree structured indexes are used for SGI XFS, ReiserFS, and Linux EXT file systems. However, it is noteworthy that static hashing schemes are not as popular as dynamic indexes because they fall short of the dynamic requirements of file systems.

In this work, we show the effectiveness of extendible hashing in the context of PM. Byte-addressable PM places new challenges on dynamic data structures because the issue of failure-atomicity and recovery must be considered with care so that when recovered from failure, the data structure returns to a consistent state. Unfortunately, extendible hashing cannot be used as-is, but requires a couple of sophisticated changes to accommodate failure-atomicity of dynamic memory allocations on PM. As in other dynamic indexes, extendible hashing manages discontiguous memory spaces for hash buckets and the addresses of buckets are stored in a separate directory structure. When a bucket overflows or is underutilized, extendible hashing performs split or merge operations as in a tree-structured index, which must be performed in a failure-atomic way to guarantee consistency.

Cacheline-Conscious Extendible Hashing (CCEH) is a variant of extendible hashing with engineering decisions for low latency byte-addressable storage such as PM. For low latency PM, making effective use of cachelines becomes very important [11, 16, 35, 43]. Therefore, CCEH sets the size of buckets to a cacheline in order to minimize the number of cacheline accesses. Although CCEH manages a fine-grained bucket size, CCEH reduces the overhead of directory management by grouping a large number of buckets into an intermediate-sized *segment*. That is, CCEH works in three levels, namely, the global directory, segments pointed by the directory, and cache-line sized buckets in the segment. We also present how CCEH guarantees the failure-atomicity and recoverability of extendible hash tables by carefully enforcing the ordering of store instructions.

The main contributions of this work are as follows:

- First, we propose to use cacheline-sized buckets but reduce the size of the directory by introducing intermediate level segments to extendible hashing. The three-level structure of our cacheline-conscious extendible hashing (CCEH) guarantees that a record can be found within two cacheline accesses.
- Second, we present a failure-atomic rehashing (split and merge) algorithm for CCEH and a recovery algorithm based on MSB (most significant bit) keys that does not use explicit logging. We also show that MSB rather than LSB (least significant bit) is a more effective key for extendible hashing on PM, which is contrary to popular belief.
- Third, our extensive performance study shows that CCEH effectively adapts its size as needed while guaranteeing failure-atomicity and that the tail latency of CCEH is up to 3.4× and 8× shorter than that of the state-of-the-art Level Hashing [49] and Path Hashing [48], respectively.

The rest of this paper is organized as follows. In Section 2, we present the background and the challenges of extendible hashing on PM. In Section 3, we present Cacheline-Conscious Extendible Hashing and show how it provides failure-atomicity while reducing the amount of writes to PM. In Section 4, we present the recovery algorithm of CCEH. In Section 5, we discuss concurrency and consistency issues of CCEH. In Section 6, we evaluate the performance of PM-based hash tables. Finally, we conclude the paper in Section 7.

2 Background and Related Work

The focus of this paper is on dynamic hashing, that is, hashing that allows the structure to grow and shrink according to need. While various methods have been proposed [17, 19, 22], our discussion concentrates on extendible hashing as this has been adopted in numerous real systems [26, 30, 33, 38, 44] and as our study extends it for PM.

Extendible Hashing: Extendible hashing was developed for time-sensitive applications that need to be less affected by full-table rehashing [6]. In extendible hashing, re-hashing is an incremental operation, i.e., rehashing takes place per bucket as hash collisions make a bucket overflow. Since extendible hashing allocates a bucket as needed, pointers to dynamically allocated buckets need to be managed in a hierarchical manner as in B-trees in such a way that the split history can be kept track of. This is necessary in order to identify the correct bucket for a given hash key.

Figure 1 shows the legacy design of extendible hashing. In extendible hashing, a hash bucket is pointed to by an entry of a *directory*. The directory, which is simply a *bucket address table*, is indexed by either the leading (most significant) or the trailing (least significant) bits of the key. In the example shown in Figure 1, we assume the trailing bits are used as in common practice and each bucket can store a maximum of five key-value records. The *global depth G* stores the number of bits used to determine a directory entry. Hence, it determines the *maximum* number of buckets, that is, there are 2^G directory entries. When more hash buckets are needed, extendible hashing doubles the size of the directory by incrementing *G*. From the example, *G* is 2, so we use the low end 2 bits of the key to designate the directory entry in the directory of size 4 (2^2). Eventually, when the buckets fill up and split, needing more directory entries, *G* can be incremented to 3, resulting in a directory of size 8.

While every directory entry points to a bucket, a single bucket may be pointed to by multiple directory entries. Thus,

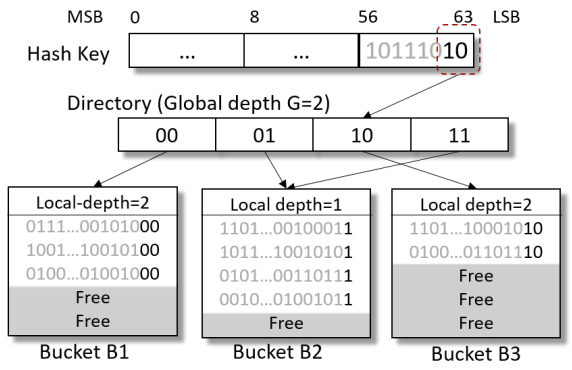


Figure 1: *Extendible Hash Table Structure*

each bucket is associated with a *local depth* (L), which indicates the length of the common hash key in the bucket. If a hash bucket is pointed by k directory entries, the local depth of the bucket is $L = G - log_2 k$. For example in Figure 1, B2 is pointed to by 2 directory entries. For this bucket, as the global depth (G) is 2 and the bucket is pointed to by two directory entries, the local depth of the bucket (L) is 1.

When a hash bucket overflows, extendible hashing compares its local depth against the global depth. If the local depth is smaller, this means that there are multiple directory entries pointing to the bucket, as for bucket B2 in Figure 1. Thus, if B2 overflows, it can be split without increasing the size of the directory by dividing the directory entries to point to two split buckets. Thus, G will remain the same, but the Ls for the two resulting buckets will both be incremented to 2. In the case where the bucket whose local depth is equal to the global depth overflows, i.e., B1 or B3 in Figure 1, the directory needs to be doubled. In so doing, both the global depth and the local depth of the two buckets that result from splitting the overflowing bucket also need to be incremented. Note, however, that in so doing, overhead is small as rehashing of the keys or moving of data only occur for keys within the bucket. With the larger global and local depths, the only change is that now, one more bit of the hash key is used to address the new buckets.

The main advantage of extendible hashing compared to other hashing schemes is that the rehashing overhead is independent of the index size. Also, unlike other static hash tables, no extra buckets need to be reserved for future growth that results in extendible hashing having higher space utilization than other hashing schemes [37]. The disadvantage of extendible hashing is that each hash table reference requires an extra access to the directory. Other static hashing schemes do not have this extra level of indirection, at the cost of full-table rehashing. However, it is known that the directory access incurs only minor performance overhead [23, 37].

PM-based Hashing: Recently a few hashing schemes, such as *Level Hashing* [49], *Path Hashing* [48], and PCM(Phase-Change Memory)-friendly hash table (PFHT) [3] have been proposed for persistent memory as the legacy in-memory hashing schemes fail to work on persistent memory due to the lack of consistency guarantees. Furthermore, persistent memory is expected to have limited endurance and asymmetric read-write latencies. We now review these previous studies.

PFHT is a variant of bucketized cuckoo hashing designed to reduce write accesses to PCM as it allows only one cuckoo displacement to avoid cascading writes. The insertion performance of cuckoo hashing is known to be about $20 \sim 30\%$ slower than the simplest linear probing [29]. Furthermore, in cuckoo hashing, if the load factor is above 50%, the expected insertion time is no longer constant. To improve the insertion performance of cuckoo hashing, PFHT uses a stash to defer full-table rehashing and improve the load factor. However, the stash is not a cache friendly structure as it linearly searches a long overflow chain when failing to find a key in a bucket. As a result, PFHT fails to guarantee the constant lookup cost, i.e., its lookup cost is not $O(1)$ but $O(\mathrm{S})$ where S is the stash size.

Path hashing is similar to PFHT in that it uses a stash although the stash is organized as an inverted binary tree structure. With the binary tree structure, path hashing reduces the lookup cost. However, its lookup time is still not constant but in log scale, i.e., $O(log\mathrm{B})$, where B is the number of buckets.

Level hashing consists of two hash tables organized in two levels. The top level and bottom level hash tables take turns playing the role of the stash. When the bottom level overflows, the records stored in the bottom level are rehashed to a $4\times$ larger hash table and the new hash table becomes the new top level, while the previous top level hash table becomes the new bottom level stash. Unlike path hashing and PFHT, level hashing guarantees constant lookup time.

While level hashing is an improvement over previous work, our analysis shows that the rehashing overhead is no smaller than legacy static hashing schemes. As the bottom level hash table is always almost full in level hashing, it fails to accommodate a collided record resulting in another rehash. The end result is that level hashing is simply performing a full-table rehash in two separate steps. Consider the following scenario. Say, we have a top level hash table that holds 100 records and the bottom level stash holds 50 records. Hence, we can insert 150 records without rehashing if a hash collision does not occur. When the next 151st insertion incurs a hash collision in the bottom level, the 50 records in the bottom level stash will be rehashed to a new top level hash table of size 200 such that we have 150 free slots. After the rehash, subsequent 150 insertions will make the top level hash table overflow. However, since the bottom level hash table does not have free space either, the 100 records in the bottom level hash table have to be rehashed. To expand a hash table size to hold 600 records, level hashing rehashes a total of 150 records, that is, 50 records for the first rehashing

and another 100 records for the second rehashing.

On the other hand, suppose the same workload is processed by a legacy hash table that can store 150 records as the initial level hash table does. Since the 151st insertion requires more space in the hash table, we increase the hash table size by four times instead of two as the level hashing does for the bottom level stash. Since the table now has 600 free spaces, we do not need to perform rehashing until the 601th insertion. Up to this point, we performed rehashing only once and only 150 records have been rehashed. Interestingly, the number of rehashed records are no different. We note that the rehashing overhead is determined by the hash table size, not by the number of levels. As we will show in Section 6, the overhead of rehashing in level hashing is no smaller than other legacy static hashing schemes.

To mitigate the shortage of space in bottom-level stash, level hashing proposes to use the bottom-to-top cuckoo displacement that evicts records from the bottom level stash to the top level hash table. However, in our experiments, when we insert 160 million records into a level hash table we observe the bottom-to-top cuckoo displacement occurs with a probability of 0.001% (only 1882 times) while rehashing occurs 14 times. As such, we find that in our experiments, bottom-to-top eviction rarely helps in improving the load factor or postponing rehashing.

One of the challenges in cuckoo displacement is that two cachelines need to be updated in a failure-atomic manner as we move a record into another bucket. If a system crashes during migration, there can be duplicate records after the system recovers. Suppose one of the duplicate records exists in the top level and the other record is in the bottom level. When a subsequent transaction updates the record, the one in the top level will be updated. Later, the top level hash table becomes the bottom level stash and another transaction will access the new top level hash table and find the stale record, which is not acceptable. Level hashing proposes to delete one of the two items when a subsequent transaction updates the item. Since every update transaction has to detect if there is a duplicate record, update transactions in level hashing needs to access other cachelines that have the possibility of having a duplicate record. In the worst case, each update transaction has to access every cacheline in each bucket referenced by two cuckoo hash functions in both levels. We note that such a worst case happens when there are no duplicate records, which would be the most common case in practice. To fix the problem in a more efficient way, we need to scan the entire hash table every time the system recovers from failure.

3 Cacheline-Conscious Extendible Hashing

In this section, we present Cacheline-Conscious Extendible Hashing (CCEH), a variant of extendible hashing that overcomes the shortcomings of traditional extendible hashing by guaranteeing failure-atomicity and reducing the number of cacheline accesses for the benefit of byte-addressable PM.

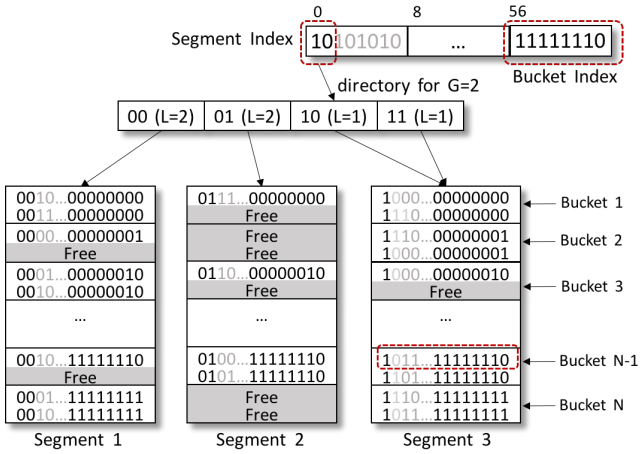


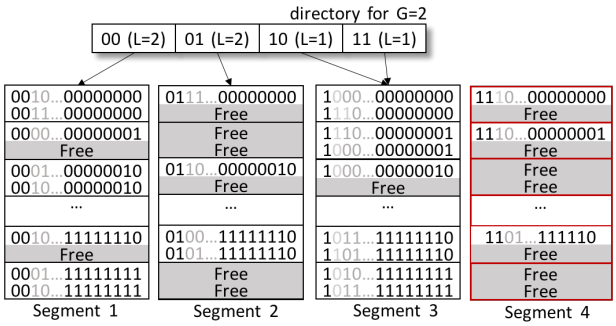
Figure 2: *Cacheline-Conscious Extendible Hashing*

3.1 Three Level Structure of CCEH

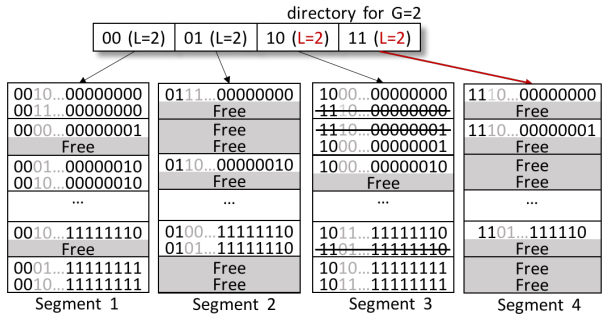
In byte-addressable PM, the unit of an atomic write is a word but the unit of data transfer between the CPU and memory corresponds to a cacheline. Therefore, the write-optimal size of a hash bucket is a cacheline. However, a cacheline, which is typically 64 bytes, can hold no more than four key-value pairs if the keys and values are word types. Considering that each cacheline-sized bucket needs an 8-byte pointer in the directory, the directory can be the tail wagging the dog, i.e., if each 64-byte bucket is pointed by a single 8-byte directory entry, the directory can be as large as 1/8 of the total bucket size. If multiple directory entries point to the same bucket, the directory size can be even larger. To keep the directory size under control, we can increase the bucket size. However, there is a trade-off between bucket size and lookup performance as increasing the bucket size will make lookup performance suffer from the large number of cacheline accesses and failure to exploit cache locality.

In order to strike a balance between the directory size and lookup performance, we propose to use an intermediate layer between the directory and buckets, which we refer to as a *segment*. That is, a segment in CCEH is simply a group of buckets pointed to by the directory. The structure of CCEH is illustrated in Figure 2. To address a bucket in the three level structure, we use the G bits (which represents the global depth) as a segment index and an additional B bits (which determines the number of cachelines in a segment) as a bucket index to locate a bucket in a segment.

In the example shown in Figure 2, we assume each bucket can store two records (delimited by the solid lines within the segments in the figure). If we use B bits as the bucket index, we can decrease the directory size by a factor of $1/2^B$ (1/256 in the example) compared to when the directory addresses each bucket directly. Note that although the three level structure decreases the directory size, it allows access to a specific



(a) Step 1: Create Sibling



(b) Step 2: Split and Lazy Deletion

Figure 3: *Failure-Atomic Segment Split Example*

bucket (cacheline) without accessing the irrelevant cachelines in the segment.

Continuing the example in Figure 2, suppose the given hash key is $10101010...11111110_{(2)}$ and we use the least significant byte as the bucket index and the first two leading bits as the segment index since the global depth is 2. We will discuss why we use the leading bits instead of trailing bits as the segment index later in Section 3.4. Using the segment index, we can lookup the address of the corresponding segment (Segment 3). With the address of Segment 3 and the bucket index ($11111110_{(2)}$), we can directly locate the address of the bucket containing the search key, i.e., (&Segment3+64 $\times$ $11111110_{(2)}$). Even with large segments, the requested record can be found by accessing only two cachelines — one for the directory entry and the other for the corresponding bucket (cacheline) in the segment.

3.2 Failure-Atomic Segment Split

A split performs a large number of memory operations. As such, a segment split in CCEH cannot be performed by a single atomic instruction. Unlike full-table rehashing that requires a single failure-atomic update of the hash table pointer, extendible hashing is designed to reuse most of the segments and directory entries. Therefore, the segment split algorithm of extendible hashing performs several in-place updates in the directory and copy-on-writes.

In the following, we use the example depicted in Figure 3 to walk through the detailed workings of our proposed failure-atomic segment split algorithm. Suppose we are to insert key $1010...11111110_{(2)}$. Segment 3 is chosen as the leftmost bit is 1, but the 255th ($11111111_{(2)}$th) bucket in the segment has no free space, i.e., a hash collision occurs. To resolve the hash collision, CCEH allocates a new Segment and copies key-value records not only in the collided bucket of the segment but also in the other buckets of the same segment according to their hash keys. In the example, we allocate a new Segment 4 and copy the records, whose key prefix starts with 11, from Segment 3 to Segment 4. We use the two leading bits because the local depth of Segment 3 will be increased to 2. If the prefix is 10, the record remains in Segment 3, as illustrated in Figure 3(a).

In the next step, we update the directory entry for the new Segment 4 as shown in Figure 3(b). First, (1) the pointer and the local depth for the new bucket are updated. Then, (2) we update the local depth of the segment that we split, Segment 3. I.e., we update the directory entries from right to left. The ordering of these updates must be enforced by inserting an `mfence` instruction in between each instruction. Also, we must call `clflush` when it crosses the boundary of cachelines, as was done in FAST and FAIR B-tree [11]. Enforcing the order of these updates is particularly important to guarantee recovery. Note that these three operations cannot be done in an atomic manner. That is, if a system crashes during the segment split, the directory can find itself in a partially updated inconsistent state. For example, the updated pointer to a new segment is flushed to PM but two local depths are not updated in PM. However, we note that this inconsistency can be easily detected and fixed by a recovery process without explicit logging. We detail our recovery algorithm later in Section 4.

A potential drawback of our split algorithm for three level CCEH is that a hash collision may split a large segment even if other buckets in the same segment have free space. To improve space utilization and avoid frequent memory allocation, we can employ ad hoc optimizations such as *linear probing* or *cuckoo displacement*. Although these ad hoc optimizations help defer expensive split operations, they increase the number of cacheline accesses and degrade the index lookup performance. Thus, they must be used with care. In modern processors, serial memory accesses to adjacent cachelines benefit from hardware prefetching and memory level parallelism [11]. Therefore, we employ simple linear probing that bounds the number of buckets to probe to four cachelines to leverage memory level parallelism.

Similar to the segment split, a segment merge performs the same operations, but in reverse order. That is, (1) we migrate the records from the right segment to the left segment. Next, (2) we decrease the local depths and update pointers of the two segments in the directory. Note that we must update these directory entries from left to right, which is the

opposite direction to that used for segment splits. This ordering is particularly important for recovery. Details about the ordering and recovery will be discussed in Section 4.

3.3 Lazy Deletion

In legacy extendible hashing, a bucket is atomically cleaned up via a page write after a split such that the bucket does not have migrated records. For failure-atomicity, disk-based extendible hashing updates the local depth and deletes migrated records with a single page write.

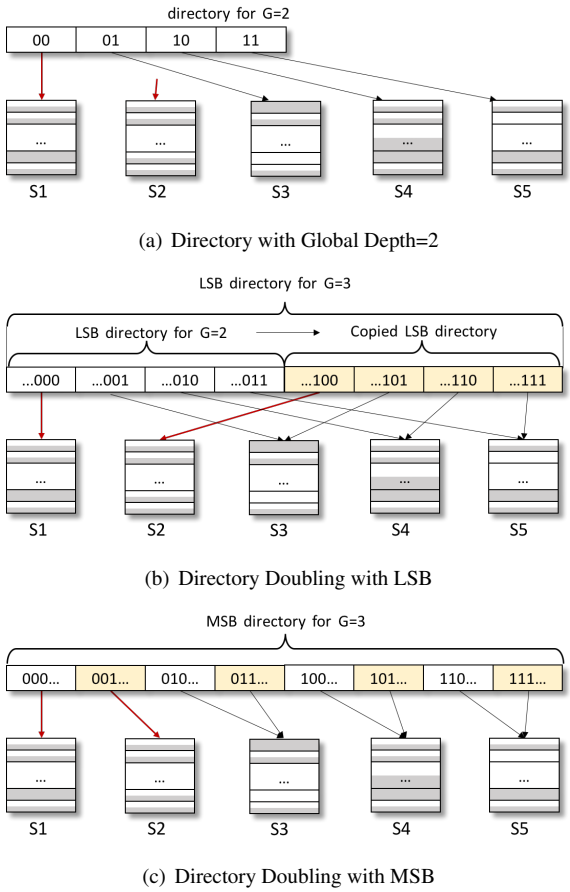
Unlike legacy extendible hashing, CCEH does not delete migrated records from the split segment. As shown in Figure 3(b), even if Segments 3 and 4 have duplicate key-value records, this does no harm. Once the directory entry is updated, queries that search for migrated records will visit the new segment and queries that search for non-migrated records will visit the old segment but they always succeed in finding the search key since the split Segment 3 contains all the key-value records, with some unneeded duplicates.

Instead of deleting the migrated records immediately, we propose *lazy deletion*, which helps avoid the expensive copy-on-write and reduce the split overhead. Once we increase the local depth of the split segment in the directory entry, the migrated keys (those crossed-out keys in Figure 3(b)) will be considered invalid by subsequent transactions. Therefore, there is no need to eagerly overwrite migrated records because they will be ignored by read transactions and they can be overwritten by subsequent insert transactions in a lazy manner. For example, if we insert a record whose hash key is $1010...11111110_{(2)}$, we access the second to last bucket of Segment 3 (in Figure 3(b)) and find the first record's hash key is $1000...11111110_{(2)}$, which is valid, but the second record's hash key is $1101...11111110_{(2)}$, which is invalid. Then, the insert transaction replaces the second record with the new record. Since the validity of each record is determined by the local depth, the ordering of updating directory entries must be preserved for consistency and failure-atomicity.

3.4 Segment Split and Directory Doubling

Although storing a large number of buckets in each segment can significantly reduce the directory size, directory doubling is potentially the most expensive operation in large CCEH tables. Suppose the segment pointed to by the first directory entry splits, as shown in Figure 4(a). To accommodate the additional segment, we need to double the size of the directory and make each existing segment referenced by two entries in the new directory. Except for the two new segments, the local depths of existing segments are unmodified and they are all smaller than the new global depth.

For disk-based extendible hashing, it is well known that using the least significant bits (LSB) allows us to reuse the directory file and to reduce the I/O overhead of directory



(a) Directory with Global Depth=2

(b) Directory Doubling with LSB

(c) Directory Doubling with MSB

Figure 4: *MSB segment index makes adjacent directory entries be modified together when a segment splits*

doubling because we can just copy the directory entries as one contiguous block and append it to the end of the file as shown in Figure 4(b). If we use the most significant bits (MSB) for the directory, new directory entries have to be sandwiched in between existing entries, which makes all pages in the directory file dirty.

Based on this description, it would seem that making use of the LSB bits would be the natural choice for PM as well. In contrary, however, it turns out when we store the directory in PM, using the most significant bits (MSB) performs better than using the LSB bits. This is because the existing directory entries cannot be reused even if we use LSB since all the directory entries need to be stored in contiguous memory space. That is, when using LSB, we must allocate twice as much memory as the old directory uses, copy the old directory to the first half as well as to the second half.

The directory doubling is particularly expensive because of cacheline flushes that are required for failure atomicity. In fact, the overhead of doubling the directory with two `memcpy()` function calls and iterating through a loop to duplicate each directory entry is minimal compared to the

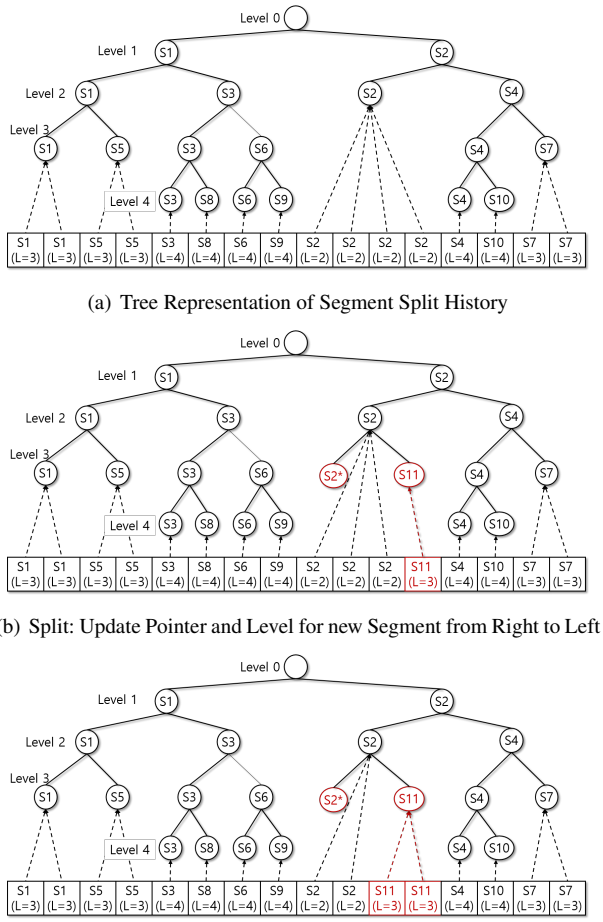
overhead of `clflush`. Note that when we index 16 million records using 16 KByte segments, it takes 555 usec and 631 usec to double the directory when we use LSB and MSB respectively. However, `clflush()` takes about 2 msec (3∼4× higher). In conclusion, LSB does not help reduce the overhead of enlarging the directory size unlike the directory file on disks.

The main advantage of using MSB over LSB comes from reducing the overhead of segment splits, not from reducing the overhead of directory doubling. If we use MSB for the directory, as shown in Figure 4(c), the directory entries for the same segment will be adjacent to each other such that they benefit from spatial locality. That is, if a segment splits later, multiple directory entries that need to be updated will be adjacent. Therefore, using MSB as segment index reduces the number of cacheline flushes no matter what local depth a split segment has. We note, however, that even though this has a positive effect of reducing the overhead for directory doubling, in terms of performance, it is more important to reduce the overhead of segment splits as segment splits occur much more frequently. Even though preserving the spatial locality has little performance effect on reducing the overhead of directory doubling because both MSB and LSB segment index call the same number of `clflush` instructions in batches when doubling the directory, MSB segment index has a positive effect of reducing the overhead of segment splits, which occur much more frequently than directory doubling. As we will see next, using MSB has another benefit of allowing for easier recovery.

4 Recovery

Various system failures such as power loss can occur while hash tables are being modified. Here, we present how CCEH achieves failure-atomicity by discussing system failures at each step of the hash table modification process.

Suppose a system crashes when we store a new record into a bucket. First, we store the value and its key next. If the key is of 8 bytes, the key can be atomically stored using the key itself as a commit mark. Even if the key is larger than 8 bytes, we can make use of the leading 8 bytes of the key as a commit mark. For example, suppose the key type is a 32 byte string and we use the MSB bits as the segment index and the least significant byte as the bucket index. We can write the 24 byte suffix first, call `mfence`, store the leading 8 bytes as a commit mark, and call `clflush`. This ordering guarantees that the leading 8 bytes are written after all the other parts of the record have been written. Even if the cacheline is evicted from the CPU cache, partially written records will be ignored because the key is not valid for the segment, i.e., the MSB bits are not a valid segment index. This is the same situation as when our lazy deletion considers a slot with any invalid MSB segment index as free space. Therefore, the partially written records without the correct leading 8 bytes will



(a) Tree Representation of Segment Split History

(b) Split: Update Pointer and Level for new Segment from Right to Left

(c) Split: Increase Level of Split Segment from Right to Left

Figure 5: *Buddy Tree Traversal for Recovery*

be ignored by subsequent transactions. Since all hash tables including CCEH initialize new hash tables or segments when they are first allocated, there is no chance for an invalid key to have a valid MSB segment index by pure luck. To delete a record, we change the leading 8 bytes to make the key invalid for the segment. Therefore, the insertion and deletion operations that do not incur bucket splits are failure-atomic in CCEH.

Making use of the MSB bits as a segment index not only helps reduce the number of cacheline flushes but also makes the recovery process easy. As shown in Figure 5, with the MSB bits, the directory entries allow us to keep track of the segment split history as a binary buddy tree where each node in the tree represents a segment. When a system crashes, we visit directory entries as in binary tree traversal and check their consistency, which can be checked by making use of G and L. That is, we use the fact that, as we see in Figure 3, if G is larger than L then the directory buddies must point to the same segment, while if G and L are equal, then each must point to different segments.

Algorithm 1 Directory Recovery

```
1:  while i < Directory.Capacity do
2:      Depth_Cur ← Directory[i].Depth_local
3:      Stride ← 2^(Depth_global − Depth_Cur)
4:      j ← i + Stride                                  ▷ Buddy Index
5:      Depth_Buddy ← Directory[j].Depth_local
6:      if Depth_Cur < Depth_Buddy then                 ▷ Left half
7:          for k ← j − 1; i < k; k ← k − 1 do
8:              Directory[k].Depth_local ← Depth_Cur
9:      else
10:         if Depth_Cur = Depth_Buddy then             ▷ Right half
11:             for k ← j + 1; k < j + Stride; k ← k + 1 do
12:                 Directory[k] ← Directory[j]
13:         else                  ▷ Depth_Cur > Depth_Buddy; Shrink
14:             for k ← j + Stride − 1; j <= k; k ← k − 1 do
15:                 Directory[k] ← Directory[j + Stride − 1]
16:     i ← i + 2^(Depth_global − (Depth_Cur − 1))
```

Let us now see how we traverse the directories. Note that the local depth of each segment and the global depth determine the segment's stride in the directory, i.e., how many times the segment appears contiguously in the directory. Since the leftmost directory entry is always mapped to the root node of the buddy tree because of the in-place split algorithm, we first visit the leftmost directory entry and check its buddy entry. In the walking example, the buddy of S1 (directory[0]) is S5 (directory[2]) since its stride is $2^{G-L} = 2$. After checking the local depth and pointer of its right buddy, we visit the parent node by decreasing the local depth by one. I.e., S1 in level 2. Now, the stride of S1 in level 2 is $2^{G-L} = 4$. Hence, we visit S3 (directory[4]) and check its local depth. Since the local depth S3 is higher (4 in the example), we can figure out that S3 has split twice and its stride is 1. Hence, we visit directory[5] and check its consistency, continuing this check until we find any inconsistency. The pseudo code of this algorithm is shown in Algorithm 1.

Suppose a system crashes while splitting segment S2 in the example. According to the split algorithm we described in Section 3.2, we update the directory entries for the split segment from right to left. Say, a system crashes after making directory[11], colored red in the Figure 5(b), point to a new segment S11. The recovery process will traverse the buddy tree and visit directory[8]. Since the stride of S2 is 4, the recovery process will make sure directory[9], directory[10], and directory[11] have the same local depth and point to the same segment. Since directory[11] points to a different segment, we can detect the inconsistency and fix it by restoring its pointer. If a system crashes after we update directory[10] and directory[11] as shown in Figure 5(c), we can either restore the two buddies or increase the local depth of directory[8] and directory[9].

5 Concurrency and Consistency Model

Rehashing is particularly challenging when a large number of transactions are concurrently running because rehashing requires all concurrent write threads to wait until rehashing is complete. To manage concurrent accesses in a thread-safe way in CCEH, we adapt and make minor modifications to the two level locking scheme proposed by Ellis [5], which is known to show reasonable performance for extendible hashing [24]. For buckets, we protect them using a reader/writer lock. For segments, we have two options. One option is that we protect each segment using a reader/writer lock as with buckets. The other option is the lock-free access to segments.

Let us first describe the default reader/writer lock option. Although making use of a reader/writer lock for each segment access is expensive, this is necessary because of the in-place lazy deletion algorithm that we described in Section 3.2. Suppose a read transaction T1 visits a segment but goes to sleep before reading a record in the segment. If we do not protect the segment using a reader/writer lock, another write transaction T2 can split the segment and migrate the record to a new segment. Then, another transaction accesses the split segment and overwrites the record that the sleeping transaction is to read. Later, transaction T1 will not find the record although the record exists in the new buddy segment.

The other option is lock-free access. Although lock-free search cannot enforce the ordering of transactions, which makes queries vulnerable to *phantom and dirty reads* problems [37], it is useful for certain types of queries, such as OLAP queries, that do not require a strong consistency model because lock-free search helps reduce query latency.

To enable lock-free search in CCEH, we cannot use the lazy deletion and in-place updates. Instead, we can copy-on-write (CoW) split segments. With CoW split, we do not overwrite any existing record in the split segment. Therefore, a lock-free query accesses the old split segment until we replace the pointer in the directory with a new segment. Unless we immediately deallocate the split segment, the read query can find the correct key-value records even after the split segment is replaced by two new segments. To deallocate the split segment in a thread-safe way, we keep count of how many read transactions are referencing the split segment. If the reference count becomes zero, we ask the persistent heap memory manager to deallocate the segment. As such, a write transaction can split a segment even while it is being accessed by read transactions.

We note that the default CCEH with lazy deletion has a much smaller overhead for segment split than the CCEH with CoW split, which we denote as `CCEH(C)`, because it reuses the original segment so that it can allocate and copy only half the amount required for `CCEH(C)`. If a system failure occurs during a segment split, the recovery cost for lazy deletion is also only half of that of `CCEH(C)`. On the other hand, `CCEH(C)` that enables lock-free search at the cost of

weak consistency guarantee and higher split overhead shows faster and more scalable search performance, as we will show in Section 6. Another benefit of CCEH(C) is that its probing cost for search operations is smaller than that of CCEH with lazy deletion because all the invalid keys are overwritten as NULL.

For more scalable systems, lock-free extendible hashing has been studied by Shalev et al. [36]. However, such lock-free extendible hashing manages each key-value record as a *split-ordered* list, which fails to leverage memory level parallelism and suffers from a large number of cacheline accesses.

To minimize the impact of rehashing and reduce the tail latency, numerous hash table implementations including Java Concurrent Package and Intel Thread Building Block partition the hash table into small regions and use an exclusive lock for each region [8, 12, 21, 27], hence avoiding full-table rehashing. Such region-based rehashing is similar to our CCEH in the sense that CCEH rehashes only one segment at a time. However, we note that the existing region-based concurrent hash table implementations are not designed to guarantee failure-atomicity for PM. Furthermore, their concurrent hash tables use separate chaining hash tables, not dynamic hash tables [8, 12, 21, 27].

6 Experiments

We run experiments on a workstation that has four Intel Xeon Haswell-EX E7-4809 v3 processors (8 cores, 2.0GHz, 8×32KB instruction cache, 8×32KB data cache, 8×256KB L2 cache, and 20MB L3 cache) and 64GB of DDR3 DRAM. Since byte-addressable persistent main memory is not commercially available yet, we emulate persistent memory using *Quartz*, a DRAM-based PM latency emulator [9, 41]. To emulate write latency, we inject stall cycles after each clflush instructions, as was done in previous studies [10, 20, 15, 35, 42].

A major reason to use dynamic hashing over static hashing is to dynamically expand or shrink hash table sizes. Therefore, we set the initial hash table sizes such that they can store only a maximum of 2048 records. For all experiments, we insert 160 million random keys, whose keys and values are of 8 bytes. Although we do not show experimental results for non-uniformly distributed keys such as skewed distributions due to the page limit, the results are similar because well designed hash functions convert a non-uniform distribution into one that is close to uniform [1].

6.1 Quantification of CCEH Design

In the first set of experiments, we quantify the performance effect of each design of CCEH. Figure 6 shows the insertion throughput and the number of cacheline flushes when we insert 160 million records into variants of the extendible hash table, while increasing the size of the memory blocks pointed by directory entries, i.e., the segment in CCEH and the hash bucket in extendible hashing. We fix the size of the bucket in CCEH to a single cacheline, but employ linear probing and bound the probing distance to four cachelines to leverage memory level parallelism.

CCEH(MSB) and CCEH(LSB) show the performance of CCEH when using MSB and LSB bits, respectively, as the segment index and LSB and MSB bits, respectively, as the bucket index. EXTH(LSB) shows the performance of legacy extendible hashing that uses LSB as the bucket index, which is the popular practice.

When the bucket size is 256 bytes, each insertion into EXTH(LSB) calls clflush instructions about 3.5 times on average. Considering an insertion without collision requires only a single clflush to store a record in a bucket, 2.5 cacheline flushes are the amortized cost of bucket splits and directory doubling. Note that CCEH(LSB) and EXTH(LSB) are the same hash tables when a segment can hold a single bucket. Therefore, their throughputs and number of cacheline accesses are similar when the segment size of CCEH(LSB) and the bucket size of EXTH(LSB) are 256 bytes.

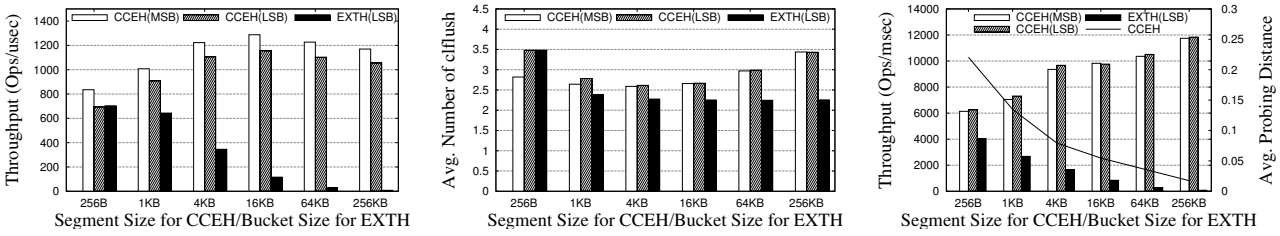
As we increase the bucket size, EXTH(LSB) splits buckets less frequently, decreasing the number of clflush down to 2.3. However, despite the fewer number of clflush calls, the insertion and search throughput of EXTH(LSB) decreases sharply as we increase the bucket size. This is because EXTH(LSB) reads a larger number of cachelines to find free space as the bucket size increases.

In contrast, as we increase the segment size up to 16KB, the insertion throughput of CCEH(MSB) and CCEH(LSB) increase because segment splits occur less frequently while the number of cachelines to read, i.e., LLC (Last Level Cache) misses, is not affected by the large segment size. However, if the segment size is larger than 16KB, the segment split results in a large number of cacheline flushes, which starts degrading the insertion throughput.

Figure 6(b) shows CCEH(MSB) and CCEH(LSB) call a larger number of clflush than EXTH(LSB) as the segment size grows. This is because CCEH(MSB) and CCEH(LSB) store records in a sparse manner according to the bucket index whereas EXTH(LSB) sequentially stores rehashed records without fragmented free spaces. Thus, the number of updated cachelines written by EXTH(LSB) is only about two-third of CCEH(LSB) and CCEH(MSB). From the experiments, we observe the reasonable segment size is in the range of 4KB to 16KB.

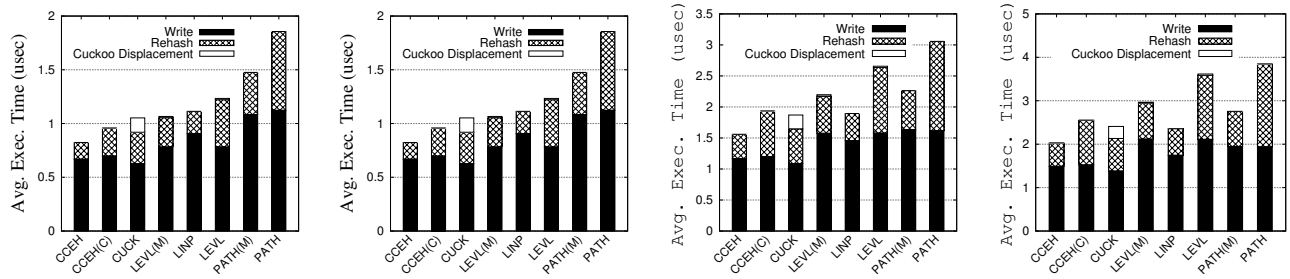
When the segment size is small, the amortized cost of segment splits in CCEH(MSB) is up to 29% smaller than that of CCEH(LSB) because CCEH(MSB) updates adjacent directory entries, minimizing the number of clflush instructions. However, CCEH(LSB) accesses scattered cachelines and fails to leverage memory level parallelism, which results in about 10% higher insertion time on average.

It is noteworthy that the search performance of



(a) Insertion Throughput (b) Number of clflush (c) Search Throughput

Figure 6: *Throughput with Varying Segment/Bucket Size*



(a) 120/120 (DRAM) (b) 240/300 (c) 240/500 (d) 240/700

Figure 7: *Breakdown of Time Spent for Insertion While Varying R/W latency of PM*

`CCEH(MSB)` and `CCEH(LSB)` improves as the segment size grows. This is because the larger the segment size, the more bits CCEH uses to determine which cacheline in the segment needs to be accessed, which helps CCEH perform linear probing less frequently. Figure 6(c) shows the average number of extra cache line accesses per query caused by linear probing. As we increases the segment size, the average probing distance decreases from 0.221 cacheline to 0.017 cacheline.

6.2 Comparative Performance

For the rest of the experiments, we use a single byte as the bucket index such that the segment size is 16 Kbytes, and we do not show the performance of `CCEH(LSB)` since `CCEH(MSB)` consistently outperforms `CCEH(LSB)`. We compare the performance of CCEH against a static hash table with linear probing (`LINP`), cuckoo hashing [29] (`CUCK`), path hashing [48] (`PATH`), and level hashing [49] (`LEVL`).[1]

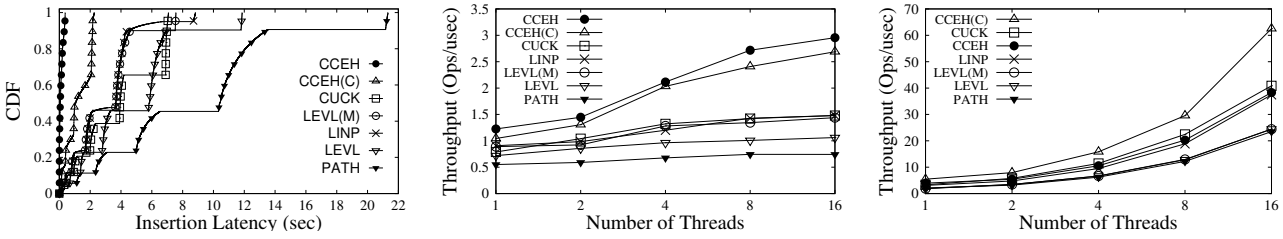
For path hashing, we set the reserved level to 8, which achieves 92% maximum load factor as suggested by the authors [48]. For cuckoo hashing, we let `CUCK` perform full-table rehashing when it fails to displace a collided record 16 times, which shows the fastest insertion performance on our testbed machine. Linear probing rehashes when the load factor reaches 95%.

[1]Our implementations of CCEH, linear probing (LINP), and cuckoo hashing (CUCK) are available at https://github.com/DICL/CCEH. For path hashing (`PATH`) and level hashing (`LEVL`), we downloaded the authors' implementations from https://github.com/Pfzuo/Level-Hashing.

In the experiments shown in Figure 7, as the latency for reads and writes of PM are changed, we insert 160 million records in batches and breakdown the insertion time into (1) the bucket search and write time (denoted as `Write`), (2) the rehashing time (denoted as `Rehash`), and (3) the time to displace existing records to another bucket, which is necessary for cuckoo hashing (denoted as `Cuckoo Displacement`).

`CCEH` shows the fastest average insertion time throughout all read/write latencies. Even if we disable lazy deletion but perform copy-on-write for segment splits, denoted as `CCEH(C)`, `CCEH(C)` outperforms `LEVL`. Note that the `Rehash` overhead of `CCEH(C)` is twice higher than that of `CCEH` that reuses the split segment via lazy deletion. However, as the write latency of PM increases, `CCEH(C)` is outperformed by `CUCK` and `LINP` because of frequent memory allocations and expensive copy-on-write operations.

Interestingly, the rehashing overhead of `LEVL` is even higher than that of `LINP`, which is just a single array that employs linear probing for hash collisions. Although `LINP` suffers from a large number of cacheline accesses due to open addressing, its rehashing overhead is smaller than all the other hashing schemes except `CCEH`. We note that the rehashing overhead of `LEVL` and `PATH` is much higher than that of `LINP` because the rehashing implementation of `LEVL` calls `clflush` to delete each record in the bottom level stash when rehashing it to the new enlarged hash table. This extra `clflush` is unnecessary for `LINP` and `CUCK`, because we can simply deallocate the previous hash table when the new hash table is ready. If a system crashes before the new hash table



(a) Insertion Latency CDF (b) Insertion Throughput (c) Search Throughput

Figure 8: *Performance of concurrent execution: latency CDF and insertion/search throughput*

is ready, we discard the new hash table and perform rehashing from the beginning. As LEVL and PATH can employ the same rehashing strategy, we implement the improved rehashing code for them, denoted as LEVL(M) and PATH(M). With the modification, LEVL(M) shows similar rehashing overhead with CCEH(C). However, it is outperformed by CCEH and LINP because its two-level structure and ad hoc optimizations such as bucketization increases the number of cacheline accesses. Note that the bucket search and write time (Write) of LEVL(M) is higher than that of CCEH and even CUCK. It is noteworthy that LEVL performs the cuckoo displacement much less frequently than CUCK and its overhead is almost negligible.

PATH hashing shows the worst performance throughout all our experiments mainly because its lookup cost is not constant, but $O(log_2 \mathrm{N})$. As the write latency increases, the performance gap between LEVL and PATH narrows down because the lookup cost becomes relatively inexpensive compared to the Write time.

6.3 Concurrency and Latency

Full-table rehashing is particularly challenging when multiple queries are concurrently accessing a hash table because it requires exclusive access to the entire hash table, which blocks subsequent queries and increases the response time. Therefore, we measure the latency of concurrent insertion queries including the waiting time, whose CDF is shown in Figure 8(a). For the workload, we generated query inter-arrival patterns using Poisson distribution where the λ rate is set to the batch processing throughput of LINP.

While the average batch insertion times differ by only up to 180%, the maximum latency of PATH is up to 56× higher than that of CCEH (378 msec vs. 21.3 sec), as shown in Figure 8(a). This is because full-table rehashing blocks a large number of concurrent queries and significantly increases their waiting time. The length of each flat region in the CDF graph represents how long each full-table rehashing takes. PATH takes the longest time for rehashing whereas LEVL, LINP, and CUCK spend a similar amount of time on rehashing. In contrast, we do not find any flat region in the graph for CCEH. Compared to LEVL, the maximum latency of CCEH is reduced by over 90%.

For the experimental results shown in Figures 8(b) and (c), we evaluate the performance of the multi-threaded versions of the hashing schemes. Each thread inserts 160/k million records in batches where k is the number of threads. Overall, as we run a larger number of insertion threads, the insertion throughputs of all hashing schemes improve slightly but not linearly due to lock contention.

Individually, CCEH shows slightly higher insertion throughput than CCEH(C) because of smaller split overhead. LEVL, LINP, CUCK, and PATH use a fine-grained reader/writer lock for each sub-array that contains 256 records (4 KBytes), which is even smaller than the segment size of CCEH (16 KBytes), but they fail to scale because of the rehashing overhead. We note that these static hash tables must obtain exclusive locks for all the fine-grained sub-arrays to perform rehashing. Otherwise, queries will access a stale hash table and return inconsistent records.

In terms of search throughput, CCEH(C) outperforms CCEH as CCEH(C) enables lock-free search by disabling lazy deletion and in-place updates as we described in Section 5. Since the read transactions of CCEH(C) are non-blocking, search throughput of CCEH(C) is 1.63×, 1.53×, and 2.74× higher than that of CCEH, CUCK, and LEVL, respectively. Interestingly, LEVL shows worse search performance than LINP. Since level hashing uses cuckoo displacement and two-level tables, which accesses noncontiguous cachelines multiple times, it fails to leverage memory level parallelism and increases the LLC misses. In addition, level hashing uses small-sized buckets as in bucketized hashing and performs linear probing for at most four buckets, which further increases the number of cacheline accesses, hurting search performance even more. As a result, LEVL shows poor search throughput.

While the results in Figure 8(c) were for queries where the lookup keys all existed in the hash table, Figure 9 shows search performance for non-existent keys. Since CUCK accesses no more than two cachelines, it shows even higher search performance than CCEH, which accesses up to four cachelines due to linear probing. Although LINP shows similar search performance with CCEH for positive queries, it suffers from long probing distance for negative queries and

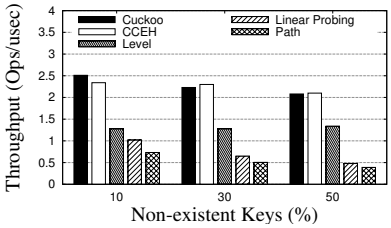


Figure 9: *Negative search*

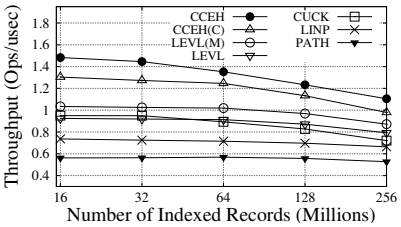


Figure 10: *YCSB throughput (Workload D: Read Latest)*

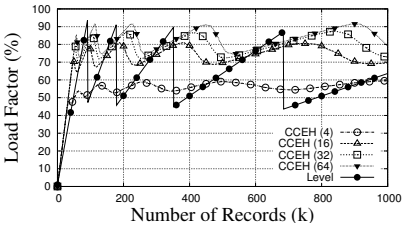


Figure 11: *Load factor per thousand insertions*

shows very poor search performance. We see that LEVL also suffers as making use of long probing, cuckoo displacement, and stash hurts search performance even more. Interestingly, PATH shows even worse search performance than LINP because of its non-constant lookup time.

We now consider the YCSB benchmarks representing realistic workloads. Figure 10 shows the throughput results of YCSB workload D as we vary the number of indexed records. In the workload, 50% of the queries insert records of size 32 bytes, while the other 50% read recently inserted records. As we increase the number of indexed records, the size of CCEH grows from 593 MBytes to 8.65 GBytes. Since hashing allows for constant time lookups, insertion and search throughput of most of the hashing schemes are insensitive to the size of hash tables. However, we observe that throughput of CCEH decreases linearly because of the hierarchical structure. When CCEH indexes 16 million records, the directory size is only 1 MBytes. Since the directory is more frequently accessed than segments, it has a higher probability of being in the CPU cache. However, when CCEH indexes 256 million records, the directory size becomes 16 MBytes while the total size of all segments is 8 GBytes. Considering that the LLC size of our testbed machine is 20 MBytes, the LLC miss ratio for the directory increases as the directory size grows. As a result, search performance of CCEH becomes similar to that of LEVL and CUCK when we index more than 64 million records and the throughput gap between CCEH and LEVL(M) narrows down.

6.4 Load Factor and Recovery Overhead

Figure 11 shows the memory utilization of CCEH and LEVL. The load factor of LEVL fluctuates between 50% and 90% because of the full-table rehashing. On each rehash, the bottom level hash table is quadrupled and the load factor drops down to 50%, which is no different from other static hash tables as we discussed in Section 2. In contrast, CCEH shows more smooth curves as it dynamically allocates small segments. Note that we can improve the load factor by increasing the linear probing distance as CCEH allocates a new segment when linear probing fails to insert a record into adjacent buckets. When we set the linear probing distance to 4 and 16, the load factor of CCEH, denoted as CCEH(4) and CCEH(16), range from 50% to 60% and from 70% to 80%, respectively. As we increase the distance up to 64, the load factor of CCEH increases up to 92%. However, as we increase the linear probing distance, the overall insertion and search performance suffers from the larger number of cacheline accesses.

While recovery is trivial in other static hash tables, CCEH requires a recovery process. To measure the recovery latency of CCEH, we varied the number of indexed records and deliberately injected faults. When we insert 32 million and 128 million records, the directory size is only 2 MBytes and 8 MBytes, respectively, and our experiments show that recovery takes 13.7 msec and 59.5 msec, respectively.

7 Conclusion

In this work, we presented the design and implementation of the cacheline-conscious extendible hash (CCEH) scheme, a failure-atomic variant of extendible hashing [6], that makes effective use of cachelines to get the most benefit out of byte-addressable persistent memory. By introducing an intermediate layer between the directory and cacheline-sized buckets, CCEH effectively reduces the directory management overhead and finds a record with at most two cacheline accesses. Our experiments show that CCEH eliminates the full-table rehashing overhead and outperforms other hash table schemes by a large margin on PM as well as DRAM.

Acknowledgments

We would like to give our special thanks to our shepherd Dr. Vasily Tarasov and the anonymous reviewers for their valuable comments and suggestions. This work was supported by the R&D program of NST (grant B551179-12-04-00) and ETRI R&D program (grant 18ZS1220), National Research Foundation of Korea (NRF) (grant No. NRF-2018R1A2B3006681 and NRF-2016M3C4A7952587), and Institute for Information & Communications Technology Promotion(IITP) (grant No. 2018-0-00549) funded by Ministry of Science and ICT, Korea. The corresponding author is Beomseok Nam.

References

[1] CARTER, J. L., AND WEGMAN, M. N. Universal classes of hash functions (extended abstract). In *Proceedings of the ACM 9th Symposium on Theory of Computing (STOC)* (1977), pp. 106–112.

[2] CHEN, S., AND JIN, Q. Persistent B+-Trees in non-volatile main memory. *Proceedings of the VLDB Endowment (PVLDB) 8*, 7 (2015), 786–797.

[3] DEBNATH, B., HAGHDOOST, A., KADAV, A., KHATIB, M. G., AND UNGUREANU, C. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads* (2015), INFLOW '15, pp. 1:1–1:9.

[4] DIETZFELBINGER, M., AND WEIDLING, C. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science 380*, 1-2 (2007), 47–68.

[5] ELLIS, C. S. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (New York, NY, USA, 1983), PODS '83, ACM, pp. 106–116.

[6] FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H. R. Extendible hashing - a fast access method for dynamic files. *ACM Trans. Database Syst. 4*, 3 (Sept. 1979).

[7] FANG, R., HSIAO, H.-I., HE, B., MOHAN, C., AND WANG, Y. High performance database logging using storage class memory. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)* (2011), pp. 1221–1231.

[8] GOETZ, B. Building a better HashMap: How ConcurrentHashMap offers higher concurrency without compromising thread safety, 2003. *https://www.ibm.com/developerworks/java/library/j-jtp08223/*.

[9] HPE. Quartz, 2018. *https://github.com/HewlettPackard/quartz*.

[10] HUANG, J., SCHWAN, K., AND QURESHI, M. K. Nvram-aware logging in transaction systems. *Proceedings of the VLDB Endowment 8*, 4 (2014).

[11] HWANG, D., KIM, W.-H., WON, Y., AND NAM, B. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Trees. In *Proceedings of the 11th USENIX Conference on File and Storage (FAST)* (2018).

[12] INTEL. Intel Threading Building Blocks Developer Reference , 2018. *https://software.intel.com/en-us/tbb-reference-manual*.

[13] IZRAELEVITZ, J., KELLY, T., AND KOLLI, A. Failure-atomic persistent memory updates via JUSTDO logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages (ASPLOS))* (2016).

[14] JOHNSON, L. An indirect chaining method for addressing on secondary keys. *Communications of the ACM 4*, 5 (1961), 218–222.

[15] KIM, W.-H., KIM, J., BAEK, W., NAM, B., AND WON, Y. NVWAL: Exploiting NVRAM in write-ahead logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[16] KIM, W.-H., SEO, J., KIM, J., AND NAM, B. clfB-tree: Cacheline friendly persistent B-tree for NVRAM. *ACM Transactions on Storage (TOS), Special Issue on NVM and Storage* (2018).

[17] KNOTT, G. D. Expandable open addressing hash table storage and retrieval. In *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control* (1971), ACM, pp. 187–206.

[18] KOLLI, A., PELLEY, S., SAIDI, A., CHEN, P. M., AND WENISCH, T. F. High-performance transactions for persistent memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016), pp. 399–411.

[19] LARSON, P.-Å. Dynamic hashing. *BIT Numerical Mathematics 18*, 2 (1978), 184–201.

[20] LEE, S. K., LIM, K. H., SONG, H., NAM, B., AND NOH, S. H. WORT: Write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)* (2017).

[21] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 27.

[22] LITWIN, W. Virtual hashing: A dynamically changing hashing. In *Proceedings of the 4th International Conference on Very Large Data Bases-Volume 4* (1978), VLDB Endowment, pp. 517–523.

[23] MENDELSON, H. Analysis of extendible hashing. *IEEE Transactions on Software Engineering*, 6 (1982), 611–619.

[24] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2002).

[25] MORRIS, R. Scatter storage techniques. *Communications of the ACM 11*, 1 (1968), 38–44.

[26] ORACLE. Architectural Overview of the Oracle ZFS Storage Appliance, 2018. *https://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/o14-001-architecture-overview-zfsa-2099942.pdf*.

[27] ORACLE. Java Platform, Standard Edition 7 API Specification, 2018. *https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html*.

[28] OUKID, I., LASPERAS, J., NICA, A., WILLHALM, T., AND LEHNER, W. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2016).

[29] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[30] PATIL, S., AND GIBSON, G. A. Scale and concurrency of giga+: File system directories with millions of files. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2011), vol. 11, pp. 13–13.

[31] PETERSON, W. W. Addressing for random-access storage. *IBM journal of Research and Development 1*, 2 (1957), 130–146.

[32] RUDOFF, A. Programming models for emerging non-volatile memory technologies. *;login 38*, 3 (June 2013), 40–45.

[33] SCHMUCK, F. B., AND HASKIN, R. L. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2002), vol. 2.

[34] SEHGAL, P., BASU, S., SRINIVASAN, K., AND VORUGANTI, K. An empirical study of file systems on nvm. In *Proceedings of the 31st International Conference on Massive Stroage Systems (MSST)* (2015).

[35] SEO, J., KIM, W.-H., BAEK, W., NAM, B., AND NOH, S. H. Failure-atomic slotted paging for persistent memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[36] SHALEV, O., AND SHAVIT, N. Split-ordered lists: Lock-free extensible hash tables. *J. ACM 53*, 3 (May 2006), 379–405.

[37] SILBERSCHATZ, A., KORTH, H., AND SUDARSHAN, S. *Database Systems Concepts*. McGraw-Hill, 2005.

[38] SOLTIS, S. R., RUWART, T. M., AND OKEEFE, M. T. The global file system. In *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies* (1996), vol. 2, pp. 319—-342.

[39] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)* (2003), pp. 43–58.

[40] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)* (2011).

[41] VOLOS, H., MAGALHAES, G., CHERKASOVA, L., AND LI, J. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 15th Annual Middleware Conference (Middleware '15)* (2015).

[42] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[43] WEISS, Z., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Densefs: a cache-compact filesystem. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (2018).

[44] WHITEHOUSE, S. The gfs2 filesystem. In *Proceedings of the Linux Symposium* (2007), Citeseer, pp. 253–259.

[45] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)* (2016).

[46] YANG, J., WEI, Q., CHEN, C., WANG, C., AND YONG, K. L. NV-Tree: reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)* (2015).

[47] ZHAO, J., LI, S., YOON, D. H., XIE, Y., AND JOUPPI, N. P. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2013), pp. 421–432.

[48] ZUO, P., AND HUA, Y. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the 33st International Conference on Massive Storage Systems and Technology (MSST)* (2017).

[49] ZUO, P., HUA, Y., AND WU, J. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, 2018).