SEPH: Scalable, Efficient, and Predictable Hashing on Persistent Memory

Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang,
The Chinese University of Hong Kong

https://www.usenix.org/conference/osdi23/presentation/wang-chao

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

SEPH: Scalable, Efficient, and Predictable Hashing on Persistent Memory

Chao Wang, Junliang Hu, Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang
The Chinese University of Hong Kong

Abstract

With the merits of high density, non-volatility, and DRAM-scale latency/bandwidth, persistent memory (PM) brings hope to high-performance storage systems, in which hashing-based index structures receive great attention owing to the efficient query performance. Though lots of efforts have been made to rethink the hashing schemes for PM in recent years, nevertheless, based on our investigation, none of them can hit performance scalability, efficiency, and predictability with one stone, seriously limiting their practicality to time-sensitive or latency-critical applications. To this end, this paper presents SEPH, a Scalable, Efficient, and Predictable Hashing for PM. SEPH paves a new direction to build the hash table by introducing the novel Level Segment (LS) structure, a key to breaking the dilemma between efficiency and predictability standing in front of the existing hashing schemes for PM. With the LS-based hash table structure, SEPH further enables a low-overhead split to greatly suppress the resizing-incurred unpredictability, and develops a semi lock-free concurrency control that requires a nearly-minimal amount of writes to handle an item insertion for achieving ever-higher efficiency and scalability while ensuring the correctness and crash consistency. Compared to state-of-the-art hashing schemes, SEPH demonstrates higher efficiency (up to 15.4× higher throughput), better scalability (performance scales up to 48 threads), and more reliable predictability (improving the tail latency by up to 19.3×).

1 Introduction

Persistent memory (PM) offers storages systems the potentials of large capacity, low latency, high throughput, and instant recovery [8,48]. The first commercial product of PM, i.e., Intel® Optane™ DC Persistent Memory Module (DCPMM) [3], is currently available on the market. As shown in Table 1, compared with DRAM, Intel® Optane™ DCPMM delivers similar write latency yet has about 2× sequential read latency and 3× sequential read latency [20,21,45]; besides, the read and write bandwidths of Intel® Optane™ DCPMM achieve nearly 1/3 and 1/6 of those of DRAM [20,21,26]. When compared with SSD, Intel® Optane™ DCPMM is even much more superior in every of these performance metrics [45]. Together with the maximal 512 GB capacity for a single module, Intel® Optane™ DCPMM is especially attractive to in-memory applications [43,45].

Table 1: Performance Comparison between DRAM and PM (i.e., Intel® Optane™ DCPMM 100 Series) [45].

	DRAM	PM	PM/DRAM
Latency of Seq. Read (ns)	81	169	208.64%
Latency of Ran. Read (ns)	101	305	301.98%
Latency of Write (ns)	57	62	108.77%
Bandwidth of Read (GB/s)	105.6	37.6	35.61%
Bandwidth of Write (GB/s)	76.8	12.5	16.28%

Index structure is a vital component for high-performance storage systems to offer efficient queries. To rapidly deploy the well-developed indexes on PM, RECIPE [26] presents a principled approach to convert concurrent DRAM indexes, including tree-based and hashing-based indexes, into crash-consistent indexes for PM. However, to better unleash the full potentials of PM, more researches focus on developing carefully-tailored indexes for PM. For example, a series of researches develops tree-based indexes for PM especially, like NV-tree [46], FAST&FAIR [19], wB+-Tree [9], LB+-Trees [31], WORT [25], BzTree [5], and ROART [35]. However, the search operation of tree-based indexes usually performs in the complexity of $O(logN)$, where N is the size of data structure, because of the hierarchical structure of trees.

By contrast, hashing-based indexes can provide constant-scale query time complexity due to the flat structures, so they are widely adopted by in-memory systems [18, 23, 29, 47]. Hashing indexes can be generally categorized into two classes: *static* and *dynamic*. Static hashing must estimate and allocate sufficient space in advance, but it suffers from hash collisions, overflows or under-utilization since the size of the hash table is hard to estimate precisely in some applications like database systems and file systems [36–38, 40]. Dynamic hashing [24], on the other hand, features in dynamically adjusting the size

of the hash table as needed by the *resizing* operation. In view of this, many delicately-designed dynamic hashing schemes are proposed for PM to achieve different optimizations, like PFHT [12], Path Hashing [49], Level Hashing [50], CCEH [36], Dash [33], and Clevel Hashing [10]. This work also focuses on the dynamic hashing schemes for PM.

Thanks to all of these efforts, the existing dynamic hashing schemes especially developed for PM have made remarkable progress on improving the overall *performance efficiency* in terms of mean throughput or mean latency. Nevertheless, surprisingly less attention has been given to the *performance predictability*, a particularly important metric in situations where the high-percentile performance would largely affect the quality of service (QoS) or the end-user experience [27, 28, 34]. In view of this, we conduct intensive experiments on a 24-core/48-thread CPU socket with six 128 GB Intel® Optane™ DCPMM to examine the in-depth performance of PM hashing schemes by utilizing different number of concurrent threads (ranging from 1 to 48). Our results (presented in §2.2) disclose that the representative hashing schemes for PM might 1) encounter the dilemma of simultaneously maintaining high *performance efficiency* and alleviating the resizing-incurred *performance unpredictability*, and 2) fall short of exhibiting good *performance scalability* under highly-concurrent queries due to their excessive writes in handling insert operations. Both seriously limit the practicality of existing PM hashing schemes to time-sensitive or latency-critical applications.

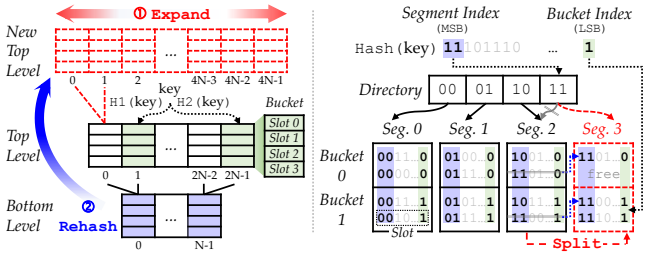
Aiming at developing a more practicable dynamic hashing scheme on PM, this paper present SEPH, a Scalable, Efficient, and Predictable Hashing for PM, to hit "three birds" with one stone. First of all, to break the dilemma between efficiency and predictability, SEPH introduces a new structure called *level segment (LS)* to build the hash table with a unique and delicate indexing mechanism (i.e., *level segment index* and *sliding bucket index*). Particularly, with the LS-based hash table structure, SEPH mitigates the inefficiency in probing items randomly, and embraces the incremental resizing (i.e., the split operation) to prevent other concurrent threads from being blocked. Second, SEPH further enables a *low-overhead split* operation to significantly suppress the resizing-incurred performance unpredictability: It not only reduces the number of KV items to be rehashed to one-third of an LS (i.e., *one-third splitting*) but even avoids the pointer dereference required to rehash a KV item for most of the time (i.e., *dereference-free rehashing*). Third, to achieve ever-higher efficiency and scalability while ensuring the correctness and crash consistency, SEPH devises a *semi lock-free mechanism* that requires a "nearly-minimal" amount of writes to handle an insertion. Our results show that SEPH performs better than the state-of-the-art hashing schemes from three perspectives. First, for efficiency, SEPH averagely achieves 2.12× higher throughput than EH-based hashing schemes, and even deliver 15.4× higher average throughput than level-based hashing schemes. Second, in terms of scalability, as the number of threads increases from 24 to 48, the performance of SEPH still scales up noticeably whereas the other hashing schemes barely improve. Third, SEPH provides more reliable predictability by achieving 11.4×~19.3× lower tail latency. SEPH is implemented in C++ and is available[1] for public use.

The rest of this paper is organized as follows. §2 presents the background and motivation regarding this work. Next, §3 introduces the design details of SEPH. Finally, §4 demonstrates the evaluations results and §5 concludes this work.

2 Background and Motivation

2.1 Hashing Schemes for Persistent Memory

Due to various *structural designs*, different hashing schemes typically have their own way to perform the *resizing operation*, an essential but expensive operation entailing extra reads and writes to enlarge the hash table for accommodating more key-value (KV) items. The existing hashing schemes, especially developed for PM, can be generally categorized into two series: 1) **Level-based hashing**, a series that features a *multi-level structure* to enable *cost-efficient resizing*, and 2) **EH-based hashing**, a series that inherits the advantage of *incremental resizing* from Extendible Hash (EH) [13].



(a) Level-based (Level Hashing) (b) EH-based (CCEH)

Figure 1: Two Series of Hashing Schemes for PM.

2.1.1 Level-based Hashing

Level Hashing. Since memory writes in PM typically consume more time and energy than memory reads, the extra writes entailed by the resizing operation might bring a negative impact on PM in terms of both performance and endurance. Because of this reason, Level Hashing [50] introduces a new *sharing-based two-level structure* to enable a *cost-efficient resizing operation* for PM.

As illustrated in Figure 1(a), Level Hashing organizes KV items into two levels (i.e., *top level* and *bottom level*) of Bucketized Cuckoo Hashing (BCH) [14] with every bottom-level bucket shared by two consecutive top-level buckets, and thus the total number of buckets in the bottom level is just a half of that in the top level. Just like the design of BCH, Level Hashing employs a pair of hash functions (denoted as `H1` and `H2` in Figure 1(a)) so that any KV item can be associated with two buckets (aka *candidate buckets* [50]) in each level and can be placed in any *slot* of the candidate buckets. But in

[1] https://github.com/cuhk-mass/SEPH

Level Hashing, only the top-level buckets are addressable by hash functions while a bottom-level bucket is mainly served as standby slots to keep conflicting KV items. That is, if a hash collision occurs in a top-level bucket and all slots in that bucket are used, the conflicting KV item can be stored in its corresponding standby bucket in the bottom level.

When all possible candidate buckets are full, Level Hashing cost-efficiently resizes the hash table as follows. As illustrated in Figure 1(b), the hash table is firstly "expanded" by allocating a *new top level* that is twice the size of the original top level and is with all the contents cleared with zeroes (denoted as ①); then, all the KV items in the original bottom level are "rehashed" into the newly allocated top level (denoted as ②); finally, the newly allocated top level and the original top level form a new sharing-based two-level structure. That is, with the cost-efficient resizing operation, Level Hashing doubles the total size of hash table but only rehashes and migrates KV items in one-third of buckets of the original hash table.

Clevel Hashing. Although Level Hashing enables a cost-efficient resizing operation, it must lock the entire hash table structure and inevitably blocks the normal hash operations (e.g., insert and search operations) from concurrent threads. To address this issue, Clevel Hashing [10], a *crash-consistent and lock-free concurrent hash table* is developed based on Level Hashing. Specifically, to avoid blocking the concurrent accesses during the resizing operation, in Clevel Hashing, the thread that triggers the resizing operation only expands the hash table for completing the insertion of KV item in the newly allocated top level, while the remaining work of rehashing is postponed and offloaded to dedicated background thread(s). As a result, in Clevel Hashing, the hash table may consist of more than two levels when it is under resizing.

2.1.2 EH-based Hashing

Another series of PM hashing schemes is evolved from Extendible Hashing (EH) [13], a widely-adopted hashing scheme that features the incremental resizing operation, called *split operation*, to avoid the full-table rehashing. Particularly, EH organizes KV items in *buckets* of a fixed number of *slots*, where a *directory* is maintained to index buckets based on the hashed value of a key (hereafter called the *hashed key* for simplicity). When a bucket overflows, EH performs the split operation to resize the hash table in the granularity of bucket rather than the entire hash table.

Cacheline-Conscious Extendible Hashing (CCEH). On the basis of EH, CCEH [36] is developed to make effective use of cachelines for better performance while guaranteeing failure-atomicity for dynamic resizing. Specifically, CCEH proposes to set the bucket size to the size of a cacheline (e.g., 64-byte) for minimizing the number of cacheline accesses for visiting a bucket. Besides, as shown in Figure 1(b), CCEH introduces an intermediate granularity named *segment*, which consists of a fixed number of buckets indexed by the same directory entry, so that the directory can be greatly shrunk to have a higher probability of being in the CPU cache. CCEH also introduces a new way to associate KV items with segments and buckets: The most significant bits of the hashed key are used to locate a segment (denoted as *segment index*) while the least significant bits are used to index a bucket within a segment (denoted as *bucket index*). To further increase the load factor, CCEH adopts linear probing [15] so that a KV item can also be placed in the next few (e.g., four) buckets following the indexed one (by the bucket index).

When all candidate buckets (i.e., the indexed bucket and the following few that can be linearly probed) are all full for a newly-inserted KV item, CCEH resizes its hash table via the split operation (i.e., an incremental resizing operation introduced by EH) as follows: First, as illustrated in Figure 1(b), a new empty segment is dynamically allocated. Second, KV items in the collided segment are either stayed or rehashed into the newly allocated segment according to their segment and bucket indexes. Finally, after all KV items are rehashed, the directory is updated to ensure that the newly allocated segment will be indexed properly by the corresponding directory entry.

Dynamic and Scalable Hashing (Dash). Dash [33] further introduces several advancements to two classical hashing schemes (i.e., Extendible Hashing (EH) [13] and Linear Hashing (LH) [30])) and showcases its effectiveness on real PM product (i.e., Intel Optane DCPMM [3]).

Dash for EH (Dash-EH) inherits most of designs from CCEH [36] but aligns the bucket size with the XPLine size (i.e., 256-byte) of Intel® Optane DCPMM for better locality [33]. Moreover, Dash-EH divides every bucket into a record region (224-byte) and a metadata region (32-byte), where the former maintains pointers to KV items for supporting variable-length keys and values while the latter is dedicated to optimizing the probing and load factor. On the one hand, for every KV item, Dash-EH keeps the second least significant byte of the hashed key as a *fingerprint* in the metadata region, so that the number of pointer dereferences, required by probing or checking the uniqueness of a KV item, can be thereby reduced; besides, Dash-EH adopts an optimistic concurrency control to avoid locking the entire segment when searching a KV item. On the other hand, Dash-EH combines a variety of techniques to increase the load factor, such as probing one more bucket, balancing the load factor of candidate buckets, allowing one movement among the indexed and linearly-probed buckets, and adding a few (e.g., two or four) *stash buckets* into each segment to accommodate conflicting KV items.

2.2 Motivation

Though the existing studies have made remarkable progress on advancing the hashing schemes for PM, this section will disclose that the existing two series of hashing schemes might 1) encounter the dilemma of achieving both high *performance efficiency* and high *performance predictability* simultaneously

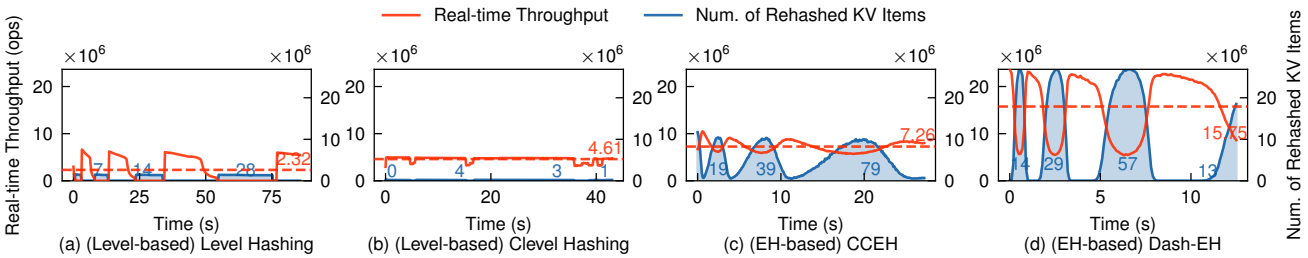


Figure 2: Real-time Throughput and Resizing Overhead under Mixed Workload (i.e., 50% Insertion and 50% Search).

(see §2.2.1) and 2) fall short of exhibiting good *performance scalability* under highly-concurrent queries (see §2.2.2). Both seriously limit the practicality of existing PM hashing schemes to time-sensitive or latency-critical applications.

2.2.1 Dilemma between Efficiency and Predictability

When examining the performance of PM hashing schemes, most of the existing studies mainly focus on the overall *performance efficiency* (i.e., the average performance) yet overlook the *performance predictability* (i.e., the high-percentile performance [27, 28, 34]), which is particularly crucial to time-sensitive or latency-critical applications in practice.

To investigate both performance efficiency and performance predictability of the existing two series of PM hashing schemes, we conduct intensive experiments on a 24-core/48-thread CPU socket with six 128 GB Intel® Optane™ DCPMM configured as the *App Direct* mode. More detailed experimental setups and implementation notes can be found in §4.1. Particularly, we preload each hashing scheme with 10 millions of KV items, and then measure the real-time performance of executing 200 millions of realistic mixed workloads with 48 concurrent threads, where the workloads consist of 50% search and 50% insert operations generated by YCSB [11] under the Zipf distribution with both key and value sizes set to 16B. Figure 2 shows the real-time throughput (in terms of operations per second) and the real-time resizing overhead (in terms of the number of rehashed KV items) of the four representative PM hashing schemes presented in §2.1. The results reveal that the existing PM hashing schemes might encounter the dilemma of achieving both high efficiency and high predictability simultaneously based on the following two key observations.

Observation 1: *Compared with Level-based hashing schemes, EH-based hashing schemes demonstrate the strength in performance efficiency yet entail heavier resizing-incurred overhead to degrade its performance predictability.*

It can be firstly observed from Figure 2 that EH-based hashing schemes demonstrate superior performance efficiency (i.e., at least 57.48% faster in terms of average throughput) than Level-based hashing schemes. The rationale behind this can be attributed to how these hash schemes probe the candidate buckets for a query. Specifically, EH-based hashing schemes probe the candidates buckets by sequential accesses, while Level-based hashing schemes entail one random access for each of the candidate bucket (which is inherited from BCH [14]). Given that the latency of random read is about 1.8× longer than that of sequential read on PM (according to Table 1), it turns out that EH-based hashing schemes hold the advantage in performance efficiency.

Nevertheless, since EH-based hashing schemes naturally entail heavier resizing overhead (i.e., the number of rehashed KV items) than Level-based hashing schemes, their performance predictability can be affected more considerably. It can be clearly observed that the real-time throughput of EH-based hashing schemes gets degraded severely while KV items are being rehashed at that time; additionally, the more KV items are being rehashed, the lower throughput would suffer. Moreover, it is worth noting that, though Dash-EH utilizes a variety of techniques to postpone split operations for higher load factor, it may concentrate the occurrence of split operations as an adverse effect, leaving the performance predictability of Dash-EH unimproved or even degraded. As shown in Figures 2(c) and 2(d), when compared with CCEH, Dash-EH achieves 2.16*X* higher average throughput but suffers 5.78% lower worst throughput (i.e., the 100th percentile throughput).

Observation 2: *Compared with EH-based hashing schemes, Level-based hashing schemes entail lower resizing-incurred overhead yet still fail to deliver good performance predictability due to its low performance efficiency.*

As revealed by Figure 2, thanks to the cost-efficient resizing, Level-based hashing schemes greatly alleviate the total resizing overhead than EH-based hashing schemes. Cumulatively, Level-based hashing schemes incur at least 55.45% less number of rehashed KV items than EH-based hashing schemes after handling the same amount of insert operations. It is also worthy to note that, to avoid locking the entire hash table and blocking all the other concurrent requests during the resizing (as Level Hashing does), Clevel Hashing advocates a lock-free scheme and further postpones and offloads the rehashing of KV items to dedicated background thread(s), which explains why Clevel Hashing could incur even less number of rehashed KV items than Level Hashing in the evaluation.

However, unfortunately, the effective reduction in the resizing overhead is insufficient in helping Level-based hashing schemes with delivering good performance predictability. This is because Level-based hashing schemes suffer much worse performance efficiency, not only the average but also the worst ones, when compared with EH-based hashing schemes.

Specifically, even though the worst throughput (i.e., the 100th percentile throughput) of Clevel Hashing seems to drop less from its average throughput, it is still worse than that of CCEH and Dash-EH by 47.95% and 44.87% respectively.

2.2.2 Limited Scalability

Apart from the efficiency and predictability, the *performance scalability* is also an important indicator that reflects how efficient a hashing scheme is in processing concurrent requests. To this end, we repeat the experiments presented in Figure 2 with a different number of concurrent threads, ranging from 1 to 48, and show the measured average throughput of different hashing schemes in Figure 3.

Observation 3: *The existing PM hashing schemes fall short of exhibiting good performance scalability under highly-concurrent requests due to the excessive writes in handling insert operations.*

From Figure 3(a), it can be clearly observed that none of the evaluated hashing schemes could scale up the average throughput well from 24 concurrent threads. To find out the potential bottleneck to achieve good performance scalability, we further measure the total writes of PM media introduced by different hashing schemes (by reading the hardware counters of DCPMM [45]), since the write bandwidth is one of the major weaknesses of PM (according to Table 1). The results in Figure 3(b) identify that all the evaluated hashing schemes introduce more than twice amount of writes than expected (which was estimated by multiplying the number of insert operations by the XPLine size (i.e., 256-byte)), except the lock-free Clevel Hashing.

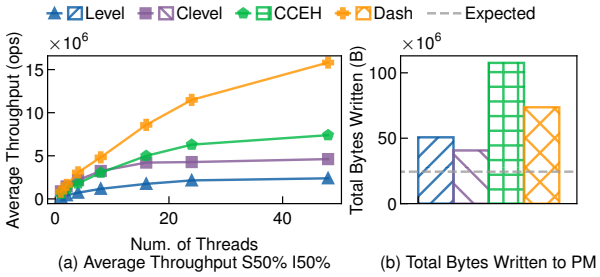


Figure 3: Scalability of Existing PM Hashing Schemes under Mixed Workload (i.e., 50% Insertion and 50% Search).

Based on our further investigation, such excessive amount of writes can be attributed to different root causes about how the PM hashing scheme handles an insertion. Specifically, for every insert operation, lock-based hashing schemes (such as Level Hashing, CCEH and Dash-EH) require one PM write to lock and insert the KV item, and another PM write for unlocking. As for a lock-free hashing scheme (like Clevel Hashing), it always requires one additional flush to persist its metadata, before every insertion, for the sake of crash consistency. In addition, it allows multiple threads to concurrently expand the hash table for the same level, but only one expansion would succeed eventually. This results in that the other failed expansions must waste PM writes to clear the memory space.

3 Design of SEPH

This section presents SEPH, a hashing on PM which can hit the *scalability*, *efficiency* and *predictability* with one stone. In this section, we first present a new structure called *level segment (LS)*, a key enabler to achieve both high efficiency and predictability, and elaborate on how SEPH builds a hash table based on LS (§3.1). Then, we show how LS can further enable a *low-overhead split* to greatly suppress the performance unpredictability caused by resizing (§3.2). Finally, we put forward a *semi lock-free concurrency control* that requires a *nearly-minimal* amount of writes to handle an insertion for achieving ever-higher efficiency and scalability while ensuring the correctness and crash consistency (§3.3).

3.1 Level Segment based Hash Table

To resolve the dilemma between efficiency and predictability disclosed by §2.2.1, SEPH introduces a new structure called *level segment (LS)* to build the hash table by combining the respective strengths of the existing two series of PM hashing. Specifically, as we are going to see in this section, LS learns from EH-based hashing to achieve better efficiency in two ways: 1) LS limits the number of buckets that need to be randomly read for a query; and 2) LS enables the incremental resizing (i.e., the split operation) to avoid the full-table rehashing. Moreover, as we will elaborate in §3.2, LS further enables a low-overhead split operation, which is inspired by the two-level structure of Level-based hashing, to greatly harness the performance unpredictability caused by resizing.

3.1.1 Structure

Physical Segment. To ease the dynamic memory allocation of PM space, SEPH manages the PM space as fixed-sized units called *physical segment (PS)*, which can be regarded as a "segment" in the EH-based hashing. To be more specific, a PS in SEPH also comprises a fixed number (e.g., 2^B) of *buckets*, each bucket also consists of a fixed number of *slots*, and each slot can also accommodate one KV item. Besides, as suggested by Dash [33], SEPH also aligns the bucket size with the XPLine size (i.e., 256-byte) of Intel® Optane™ DCPMM for achieving better locality.

Level Segment. To combine the respective strengths of the existing two series of PM hashing, SEPH further organizes PSs into a two-level structure called *level segment (LS)*, which is also the granularity for splitting. As depicted in Figure 4, given one PS at lower level (e.g., PS 0) and two PSs at higher level (e.g., PS 1 and PS 2), SEPH organizes the "left half" of lower-level PS and one higher-level PS into a LS (e.g., LS 0 which is denoted by blue-shaded region) and organizes the "right half" of the lower-level PS and another higher-level PS into the second LS (e.g., LS 1 which is denoted by green-shaded region). Moreover, within an LS, every two physically-consecutive higher-level buckets share a lower-level bucket, but SEPH gives higher priority to the lower-level

buckets for accommodating newly inserted KV items than the higher-level buckets for the sake of concurrency control (see §3.3). That is, only if all slots in a lower-level bucket of an LS are fully occupied, will a new KV item be inserted into the higher-level bucket of that LS. In view of this, when querying a KV item in an LS, SEPH also searches the lower-level candidate bucket before searching the higher-level candidate bucket for better search efficiency.

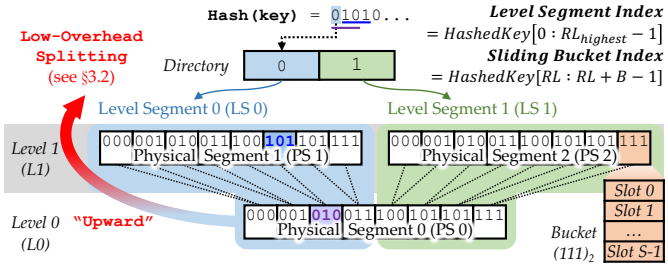


Figure 4: Level Segment based Hash Table of SEPH.

Upward Splitting. When the lower-level and higher-level candidate buckets are both full to a new KV item, SEPH *splits* that LS into two to enlarge the hash table in a copy-on-write (CoW) fashion (i.e., by rehashing existing KV items into newly allocated PSs). However, unlike the EH-based hashing splits horizontally, SEPH splits an LS in an "upward" and "low-overhead" fashion (see §3.2 for details). Consequently, as SEPH keeps splitting LSs to accommodate more KV items, the overall hash table will grow upwardly, since some LSs are evolved from more times of upward splitting and thereby reach higher levels than other LSs.

Residing Level. To precisely maintain *which level a PS currently resides*, SEPH associates every PS with an attribute called *residing level (RL)*. Taking the blue-shaded LS 0 depicted in Figure 4 as an example, the *RL*s of its higher-level PS and lower-level PS are `1` and `0`, respectively.

3.1.2 Indexing

Level Segment Index. Like the EH-based hashing, SEPH also maintains a *directory* and utilizes the most significant bits to index an LS. Specifically, if the current highest residing level among all PSs is $RL_{highest}$, there must be $2^{RL_{highest}}$ entries in the directory, and SEPH utilizes the first $RL_{highest}$ most significant bits of the hashed key as the *level segment index*. Consider the example illustrated in Figure 4 where the current highest residing level $RL_{highest}$ is 1. SEPH will associate the given KV item, whose hashed key starts with **"01010"**, with the LS indexed by the first directory entry (since the most significant bit in the hashed key is **"0"**).

Sliding Bucket Index. However, to facilitate the low-overhead split (introduced in §3.2), unlike the EH-based hashing that utilizes the least significant bits to index a bucket, SEPH introduces a unique *sliding bucket indexing* to index the candidate bucket in a PS for a KV item. Specifically, suppose *RL* denotes the *residing level* of a PS comprising 2^B buckets, SEPH uses the RL^{th} to $(RL+B-1)^{th}$ of the most significant bits in the hashed key (denoted as $HashedKey[RL : RL+B-1]$) to locate the candidate bucket according to the residing level *RL* of the PS.

Following the rule, SEPH can easily locate two candidate buckets (one in the lower-level PS and the other in the higher-level PS) in an LS for any KV item. As the example shown in Figure 4 where B is 3, given a hashed key starting with **"01010"**, the candidate bucket at lower level PS (e.g., PS 0 at Level 0) is the third one (e.g., Buckets $(010)_2$) since $HashedKey[0:2]$ is **"010"**, and the candidate bucket at higher level PS (i.e., PS 1 at Level 1) is the sixth one (i.e., Buckets $(101)_2$) since $HashedKey[1:3]$ is **"101"**.

3.2 Low-Overhead Split

To suppress the resizing-incurred performance unpredictability, SEPH proposes a *low-overhead split* operation, which not only reduces the number of KV items to be rehashed to one-third of an LS (i.e., *one-third splitting* in §3.2.1) but even avoids the pointer dereference required to rehash a KV item for most of the time (i.e., *dereference-free rehashing* in §3.2.2).

3.2.1 One-Third Splitting

With the novel Level Segment (LS) based hash table structure and the unique indexing mechanism (presented in §3.1), SEPH enables the *one-third splitting*, which only needs to rehash "one-third" of the KV items upon splitting an LS (i.e., the victim LS) into two new LSs as follows: ❶ Two new PSs are allocated at one level higher than the higher-level PS of the victim LS to address the hash collision; ❷ Only the KV items in the lower-level buckets (i.e., one-third) of the victim LS are rehashed into the two newly allocated PSs but the two newly allocated PSs and the KV items stayed in the original higher-level PS of the victim LS amazingly form two new LSs at one level higher, thanks to the unique level segment and sliding bucket indexes presented in §3.1; ❸ The corresponding directory entries are updated accordingly to point to the two newly formed LSs; ❹ The PM space occupied by the lower-level buckets of the victim LS is safely released.

Figure 5 depicts an example that walks through the whole process of the one-third splitting, where each PS is of 8 buckets (i.e., *B* equals 3). Suppose we are going to insert a new KV item with the hashed key starting with **"00011"** into LS 0, but the two candidate buckets (i.e., Bucket $(000)_2$ of PS 0 and Bucket $(001)_2$ of PS 1) are both full. To address such hash collision, SEPH splits LS 0 by rehashing only its lower-level buckets into the two newly allocated PSs (i.e., PS 3 and PS 4) at Level 2. That is, with the unique level segment index and sliding bucket index, the KV items in Buckets $(000)_2$ and $(001)_2$ of PS 0 are rehashed into the newly allocated PS 3 while the KV items in Buckets $(010)_2$ and $(011)_2$ of PS 0 are rehashed into the newly allocated PS 4; however, there is no need to rehash any KV items in PS 1 since the two newly

allocated PSs (i.e., PS 3 and PS 4), along with the existing PS 1, amazingly form two new LSs (i.e., LS 2 and LS 3). At last, the directory entries are accordingly modified to index two newly formed LSs, and the "to-be-inserted KV item" can be eventually inserted into Bucket $(011)_2$ of PS 3 of the newly formed LS 2.

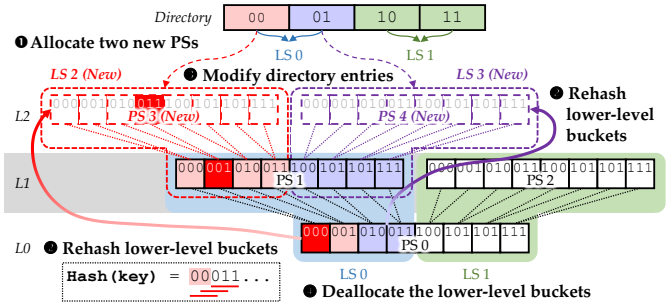


Figure 5: An Illustrative Example of One-Third Splitting.

3.2.2 Dereference-Free Rehashing

To support variable-length keys and values, like many representative PM hashing schemes (e.g., Clevel Hashing and Dash), SEPH keeps the pointers to KV items in slots of buckets. Consequently, to rehash a KV item (during the resize/split operation), typically, the pointer needs to be first *dereferenced* and a subsequent memory read is needed to get the content of a KV item, resulting in a considerable amount of random reads to degrade the performance on PM. In view of this, SEPH further enables the *dereference-free rehashing* that circumvents the pointer dereferences required to rehash KV items for minimizing the overhead of one-third splitting.

Key Insight. The main purpose of dereferencing a pointer during resizing is to locate the new candidate bucket a KV item based on the re-calculated hashed key. It means that if the new candidate bucket can be known by some means, a KV item can be directly moved into the new candidate bucket without dereferencing the pointer. Thanks to its unique sliding bucket indexing, SEPH can simply infer the new candidate bucket *if the two subsequent bits, following the current sliding bucket index of the hashed key, can be known*. This is because, during the one-third splitting, the KV items are always rehashed from the lower-level buckets of the victim LS into a newly allocated PS, which locates at two-level higher. To be more specific, for any KV items stored in the lower-level buckets of the victim LS residing at Level RL, its current sliding bucket index equals $HashedKey[RL : RL+B-1]$; since this KV item will be rehashed into a new PS located at two-lever higher (i.e., Level $RL+2$), its new sliding bucket index will become $HashedKey[RL+2 : RL+B+1]$. In other words, SEPH can infer the new candidate bucket for this KV item by only requiring two extra bits, i.e., the $(RL+B)^{th}$ and $(RL+B+1)^{th}$ bits in its hashed key.

Bucket Index Foreseer. Based on this key insight, SEPH proposes to maintain a small chunk of the hashed key, called *bucket index foreseer* (or *foreseer* for simplicity), which contains the required "two bits" for dereference-free rehashing, along with the pointer to that KV item in the slot. In our implementation, the size of the foreseer is set to 16 bits since the modern 64-bit operating systems typically use 48 or fewer bits of pointers. As shown in Figure 6, when inserting a new KV item into a PS of 2^5 buckets residing at Level 0, SEPH keeps not only the pointer to this KV item but also the first two bytes of the hashed key (i.e., "00101010 10101101") as the foreseer in the 64-bit slot. Later, when this KV item needs to be rehashed into a newly allocated PS at Level 2, since the 6^{th} and 7^{th} bits of the hashed key (i.e., "01") are maintained in the foreseer, SEPH can directly move this KV item into Bucket $(10101)_2$) in the new PS without dereferencing the pointer (denoted by ❶ in Figure 6).

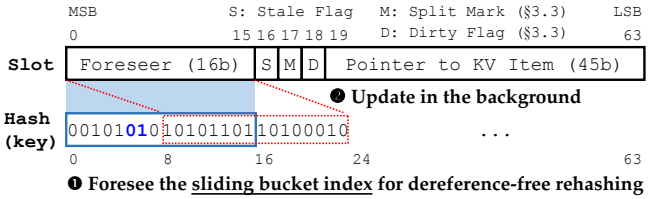


Figure 6: An Running Example of Bucket Index Foreseer.

Nevertheless, with the growth of the hash table, the foreseer might contain fewer and fewer "unused" bits and eventually "fail to foresee" the new sliding bucket index during the subsequent split operations. Thus, SEPH proposes to keep the number of unused bits in the foreseer more than half (i.e., 8 bits) at most times by updating the foreseer in the background (denoted by ❷ in Figure 6). To this end, SEPH maintains another bit called *stale flag* in the slot to indicate the staleness of the foreseer. If the unused bits of the foreseer will be less than half of its size after a dereferencte-free rehashing, SEPH sets the stale flag to 1 and submits a job to a dedicated thread, which can update the foreseer and reset the stale flag, via an atomic operation without consistency issue, in the background. Notably, if all the unused bits in a foreseer have really been used up without being timely updated, SEPH alternatively updates the foreseer right away in the foreground before rehashing the KV item.

It is also worth mentioning that the foreseer can also be utilized as the "tag" in Clevel Hashing [10] or "fingerprint" in Dash [33] to avoid unnecessary dereferences of pointers during bucket probing. This is because the foreseer will get updated timely to contain bits, which are neither LS nor sliding bucket indexes, so that it is particularly effective in telling whether a KV item exists in a bucket.

3.3 Semi Lock-Free Concurrency Control

As discussed in §2.2.2, the existing PM hashing schemes introduce an excessive amount of writes to handle an insertion. This not only brings considerable degradation to efficiency, but even largely limits the scalability especially under highly-concurrent and insert-intensive scenarios. To

achieve ever-higher efficiency and scalability, SEPH proposes a *semi lock-free concurrency control* that requires a nearly-minimal amount of writes to handle an insertion. §3.3.1 and §3.3.2 shall first introduce the design concept and the correctness challenges of the semi lock-free concurrency control, respectively. Then, §3.3.3 elaborates on how to conduct different hash operations while guaranteeing the correctness and crash consistency in depth.

3.3.1 Design Concept

Thanks to its atomicity, the compare-and-swap (CAS) primitive [4] has been widely adopted by many existing lock-free data structure and algorithm designs to allow concurrent operations without explicitly managing locks [22, 32, 39, 41]. The CAS primitive "compares" the stored content of a word in memory with a given value and, only if they match, "swaps" the content of that word with the given value; meanwhile, a Boolean value is returned to indicate whether the swap takes place or not. Notably, the execution of the CAS primitive is guaranteed to be atomic in the sense that the content of the memory word is either completely swapped or stays unchanged in an "all-or-nothing" fashion. Since SEPH sets the slot size to the word size (e.g., 8 bytes) (see §3.2.2), we can also leverage the CAS primitive to realize lock-free operations for the avoidance of excessive amounts of writes to PM due to the lock manipulation. However, in view of the fact that the split operation occurs relatively infrequent but could complicate the correctness guarantee of other frequent hash operations (i.e., insert/update/delete/search operations) [10], we propose to prudently apply the lock only to the split operation. That is, SEPH puts forward a *semi lock-free concurrency control* in which only the (infrequent) split operation needs to acquire the lock for splitting an LS, while other (frequent) hash operations (i.e., insert/update/delete/search operations) are all lock-free even when the involved LSs are under splitting.

3.3.2 Correctness Challenges

Guaranteeing the correctness of concurrent executions is one of the most critical challenges when designing lock-free data structures or algorithms. However, even though the CAS primitive can guarantee the atomicity of manipulating a slot, according to [10], performing hash operations in a lock-free manner may still lead to two correctness problems, i.e., *duplicate items* and *loss of items*.

1) Duplicate Items. The correctness problem of *duplicate items* is that concurrent lock-free insertions may place multiple KV items with the same key into different slots of the hash table. This may violate the correctness of subsequent update/delete/search operations, since the update or delete operation may only take place in one slot while the search operation may access other duplicate slots that are unmodified.

2) Loss of Items. The correctness problem of *loss of items* is that the modifications to the hash table (made by insert/update/delete operations) may be lost when the hash table is under resizing concurrently. This is possible since the concurrent modifications may be left behind (i.e., not rehashed) by the resizing, making those non-rehashed modifications "invisible" to the subsequent operations incorrectly.

3.3.3 Operation Details

Lock-Free Insert. In SEPH, the insertion operation is designed to be *lock-free* for the avoidance of the concurrency control overhead in manipulating locks. In order to address the correctness problem of *duplicate items* (see §3.3.2) caused by concurrent insertions, SEPH regulates the order of empty slot allocation in the two candidate buckets to accommodate newly inserted KV items based on the following two rules of thumb: 1) The slots in the lower-level candidate bucket must be first used up before using the slots in the high-level candidate bucket. 2) In a candidate bucket, the slots must always be allocated from the first one to the last one, where no empty slots can exist before any allocated slots and no deleted slots can be re-allocated. In summary, together with the two rules and the atomicity of CAS primitive, SEPH guarantees that concurrent insertions to the same LS will always compete for not only the same candidate bucket (rule 1) but also for the same empty slot in that bucket (rule 2) so that the correctness problem of duplicate items can be nicely avoided.

Algorithm 1 elaborates the lock-free insert operation in detail. First, it looks up the directory to find out the corresponding LS for the to-be-inserted KV item (Line 2) and performs a uniqueness check to ensure that in the two candidate buckets of that LS, there are no existing KV items holding the same key as the to-be-inserted KV item (Lines 3–9). Specifically, the uniqueness check employs the `atomic load` instruction [2, 10, 17] to atomically fetch every allocated slot one after the other for examination (Line 5) and rejects an insertion request if its key matches the key of any allocated slots in the LS (Lines 8–9). Please note that the atomic instructions will not always lead to direct accesses to PM, since these requests can also be served in the cache [2].

Then, it starts to compete for the empty slot in the two candidate buckets based on our rules for the empty slot allocation (Lines 10–21). That is, the lower-level bucket is used before using the high-level bucket (rule 1) and the slots in a bucket are allocated from the first one to the last one (rule 2), so that all the concurrent insertions to the same LS will be regulated to always compete for the same empty slot. Especially, the CAS primitive is utilized to atomically check a slot is empty (by *comparing* the content of slot with an "empty" value) and fill in the slot (by *swapping* the content of slot with the pointer to the to-be-inserted KV item) (Line 12). Thanks to the atomicity of CAS primitive, even if there are concurrent insertions competing for the same empty slot, only one thread can successfully fill in it; then, the only "CAS-succeeded" thread utilizes the `clwb` [2] and `mfence` instructions [2] to persist the filled slot into PM [7, 10, 33, 36] (Lines 13–15). Meanwhile, all the other "CAS-failed" concurrent threads must check every slot they failed to fill in, since those slot(s)

Algorithm 1: LOCK-FREE INSERT

```
   Input: kv: the pointer to the to-be-inserted KV item
   Output: the result of insertion (SUCCESS or FAIL)
1  RETRY:
2  Look up the directory to find out the LS for kv;
   // uniqueness check
3  for bucket in [LS-> PS_lower, LS-> PS_higher] do
4      foreach slot in bucket do
5          slot' ← ATOMIC LOAD(slot);
6          if slot' has a set split mark then
7              goto RETRY;
8          if kv and slot' have the same key then
9              return FAIL;
   // CAS insert
10 for bucket in [LS-> PS_lower, LS-> PS_higher] do
11     foreach slot in bucket do
12         result_CAS ← CAS(&slot, empty, kv);
13         if result_CAS is successful then
14             Persist slot into PM;
15             return SUCCESS;
16         else
17             slot' ← ATOMIC LOAD(slot);
18             if slot' has a set split mark then
19                 goto RETRY;
20             if kv and slot' have the same key then
21                 return FAIL;
   // split (both cand. buckets are full)
22 Perform LOCK-BASED ONE-THIRD SPLIT() to split LS;
23 goto RETRY;
```

may be inserted with KV items with the same key (Lines 16–21). If so, the insertion must be rejected to avoid duplicate items (Lines 20–21); otherwise, the CAS-failed thread(s) will continue to compete for the next empty slot iteratively.

Finally, if all the slots in the two candidate buckets are used up, the thread needs to trigger the one-third split operation (see Algorithm 2) to split the LS (Line 22), followed by retrying the insertion in a lock-free way (Line 23).

Lock-Based One-Third Split. In SEPH, the one-third split operation is *lock-based*. That is, an LS could only be split by one of the concurrent threads successfully. Even so, SEPH may still be threatened by the correctness problem of *loss of items* introduced in §3.3.2. Particularly, as the one-third split operation is rehashing the KV items from the lower-level PS of an LS into newly allocated PSs, some other concurrent lock-free /deletion operations may be making changes to that lower-level PS (since their lower-level candidate buckets are still not full), leaving some of these modifications not rehashed correctly. To resolve this correctness problem, SEPH devises a lightweight mechanism that allows a split operation to timely notify other concurrent lock-free operations of the rehashing status of every slot. Specifically, SEPH reserves a one-bit "split mark" in every slot, and the split mark will only be set atomically once the split operation has started to process it. It ensures that other concurrent lock-free operations can avoid

Algorithm 2: LOCK-BASED ONE-THIRD SPLIT

```
   Input: LS: the LS that needs to be split
1  if TRY ACQUIRE LOCK(LS) == SUCCESS then
2      Allocate two new PSs for one-third splitting;
3      foreach bucket in LS-> PS_lower do
4          foreach slot in bucket
5              do    // CAS Loop
6                  slot' ← ATOMIC LOAD(slot);
7                  slot'_m ← slot' | split mark;
8                  result_CAS ← CAS(&slot, slot', slot'_m);
9              while result_CAS is not successful ;
10             if slot is not an empty or deleted slot then
11                 Perform DEREFERENCE-FREE REHASH() to
                     rehash slot into the two new PSs;
12     Persist the two new PSs into PM;
13     Form two new LSs and update the directory;
```

making changes to slots with a set split mark for the avoidance of loss of items, since the "compare" of CAS primitive will fail due to the set split mark bit.

As shown in Algorithm 2, the one-third split operation in SEPH needs to first acquire the lock for splitting any LS (Line 1), and only the thread successfully acquired the lock can rehash KV items from the lower-level PS of the to-be-split LS to the two newly allocated PSs (Lines 2–13). Especially, for every slot (including empty slots), the split operation shall first exploit the CAS loop [17] to ensure the split mark can be successfully set even in the presence of the concurrent operations (Lines 5–9). Then, the *dereference-free rehashing* (see §3.2.2) is employed to rehash the KV item if it exists (Lines 10–11). Finally, only after all the slots have been set with the split mark and all the KV items have been successfully rehashed and persisted into PM (Lines 12), should the directory entry be updated to index the two newly formed LSs (Line 13). It is the key step to ensure that no other concurrent operations can access the slots in the two newly allocated PSs when the splitting is still taking place. Notably, there is no need to release the split lock after splitting, because the split LS would become stale and any other concurrent threads should not split it again. Besides, we adapt the epoch-based reclamation [16] to recycle the stale PS only after no other concurrent lock-free readers are using it [33].

With the split marks, the problem of loss of items can be avoided as follows. Particularly, if the insertion ends with a CAS success, it implies that the concurrent split operation has not yet set the split mark for that slot and will rehash the inserted KV item later. Otherwise, if the insertion ends with a CAS failure because of a set split mark, it means that a concurrent split operation has already started to process this slot by first setting the split mark with the atomic CAS primitive (i.e., Line 8 in Algorithm 2). Thus, SEPH shall retry the entire insert operation to avoid leaving an insertion of KV item behind the concurrent split operation (i.e., Lines 18–19 in Algorithm 1). It is also worthy to note that during the uniqueness check, if a slot with a set split mark is found,

SEPH shall also retry the entire insert operation to avoid accessing stale KV items (i.e., Lines 6–7 in Algorithm 1); in addition, both two candidate buckets shall be examined, since the higher-level candidate bucket may have become the lower-level candidate bucket due to concurrent splits.

Notably, the problem of duplicate items can also be avoided even if an insertion ends with a CAS success but has time overlap with a split operation to the same LS. Let's first discuss the situation that the insertion can successfully fill in a slot in the "lower-level" candidate bucket of the LS. In this scenario, the split operation must be not over yet (since the split mark of this slot has not been set) and all the other concurrent insertions with the same key must fail to compete for the same slot in the same lower-level candidate bucket (thanks to the atomicity of CAS primitive). Next, let's discuss the other situation where the insertion can successfully fill in a slot in the "higher-level" candidate bucket (denoted as *b*) of the LS (i.e., the corresponding lower-level candidate bucket is already full). In this scenario, the split operation may still be in the process or may have been completed by the time that slot is filled in. Specifically, if the split operation is not over yet, all the other concurrent insertions with the same key must compete in the same higher-level candidate bucket *b* but fail eventually. If the split operation is complete already, all the other concurrent same-key insertions launched before the split completion must compete in the same high-level candidate bucket *b* but fail eventually; meanwhile, since the higher-level candidate bucket *b* has become the lower-level candidate bucket in the new LS after the split completion (see §3.2.1), all the other concurrent same-key insertions with the same key launched after the split completion shall first compete in the same candidate bucket *b* (based on the rule 1 of empty slot allocation) but fail eventually.

Lock-Free Update/Delete. With the help of CAS primitive, in SEPH, the update and delete operations are also designed to be *lock-free*. To locate the KV item for update or deletion in the candidate buckets, the `atomic load` instruction is utilized (similar to how the uniqueness check is performed in the lock-free insertion). Then, if the slot with the desired key can be found, SEPH takes advantage of the atomicity of the CAS primitive to update the slot so that only one current thread can successfully modify it at a time. However, if the CAS primitive fails due to a set split mark in the slot, SEPH shall retry the entire update/delete operation to avoid leaving modifications to slots that have been processed by a concurrent split operation for the avoidance of loss of items.

On the other hand, based on the rules of thumb for empty slot allocation, no empty slots can exist before any allocated slots and no deleted slots can be re-allocated. Thus, in SEPH, the delete operation is realized in a way very similar to the update operation. The only difference is that the deletion replaces the desired slot with a "tombstone" instead of the updated KV item. In our implementation, we consider a slot that has all 1s for its pointer to KV item as a tombstone slot. By doing so, a tombstone slot can be easily identified and more importantly, we can still exploit the split mark and the CAS primitive to avoid losing deletions in slots that have been processed by a concurrent split operation (as how we avoid losing updates).

Lock-Free Search. Since SEPH takes advantage of the atomicity of CAS primitive to modify a slot and utilizes the `atomic load` instruction to atomically read a slot, the search operation can be easily realized as *lock-free*. However, to avoid reading stale KV items, SEPH shall retry the search operation if any slot with split mark is accessed (as what we do in the uniqueness check of insertion).

3.3.4 Persistence for CAS

The compare-and-swap (CAS) atomic instruction [4] achieves the synchronization in multithreading; however, since the processor cache is typically volatile, a thread might access data that have not been persisted yet, resulting in data inconsistency in the presence of crashes. In our current implementation of SEPH, we utilize the *persistent single-word compare-and-swap (PSwCAS)* [42] primitive that can address this problem by adding a dirty bit on each 8-byte word operated by the CAS instruction. Specifically, the PSwCAS primitive requires that 1) the CAS instruction always stores a word of data with the dirty bit set; and 2) a thread must first persist the required word into PM if the word is set with the dirty bit, followed by clearing the dirty bit to mark the word as persistent.

Notably, the extended asynchronous DRAM refresh (eADR), a new feature supported by Intel® Optane™ DCPMM 200 series and 3rd Xeon® Scalable processors, ensures that CPU caches are also included in the power fail protected domain [44]. That is, with the eADR technique, the CAS primitive can be used directly with the data consistency guarantee even in the presence of crashes [1]. Thus, we envision that SEPH shall be greatly benefited by the eADR feature to deliver even superior performance.

3.3.5 Crash Consistency

No inconsistency will occur in SEPH against crashes for the following reasons: 1) The insertion/update/deletion can be done atomically and their crash consistency can be guaranteed by the PSwCAS. 2) The split operation is protected by the lock and is conducted in a copy-on-write (CoW) manner, so the split operation is an all-or-nothing process; moreover, an unfinished split operation (which is broke off by the occurrence of crash) can also be identified (by examining the split locks of LS) and correctly redone. 3) The crash consistency for the directory can be secured by the directory recovery algorithm proposed in CCEH [36].

4 Performance Evaluation

4.1 Experimental setup

Environment. All experiments are conducted on a 24-core/48-thread Intel Xeon Platinum 8260 2.40 GHz CPU socket with

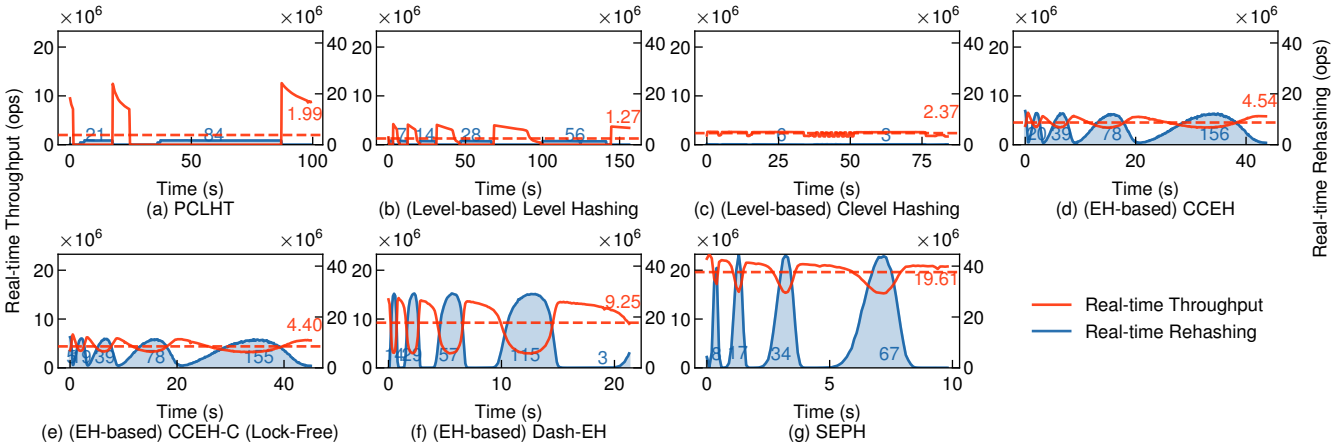


Figure 7: Real-time Throughput and Resizing Overhead of Insertion.

six 32 GB DRAM and six 128 GB Intel® Optane™ DCPMM 100 series configured as the *App Direct* mode. The operating system is 64-bit Ubuntu Server 22.04 with Linux kernel version 5.15, and Persistent Memory Development Kit (PMDK) version is 1.11. All the codes are implemented in C++ and compiled using GCC 11.2 with all optimizations enabled.

Evaluated Hashing Schemes. We evaluate the following hashing schemes especially developed for PM, and adopt parameter settings suggested by their original papers for achieving the best performance on Intel® Optane™ DCPMM 100 series.

- `PCLHT`: PCLHT is a search-optimized hashing scheme adopting a linked list based cache-efficient hash table converted by RECIPE [26].
- `Level`: Level Hashing [50] is the origin of Level-based hashing schemes (see §2.1.1). It uses 128-byte buckets (i.e., two cachelines).
- `Clevel`: Clevel Hashing [10] is an extension of Level Hashing with lock-free concurrency control (see §2.1.1). It uses 64-byte buckets (i.e., one cacheline) and employs one dedicated background thread to perform the resizing.
- `CCEH/CCEH-C`: CCEH [36] is developed based on Extendible Hashing (EH) [13] with effective use of cachelines for better performance (see §2.1.2). It uses 16 KB segments and 64-byte buckets (i.e., one cacheline) with a probing distance of 4. CCEH-C is a variant of CCEH that conducts the split operation in a CoW way to support lock-free search [36].
- `Dash`: Dash-EH [33] is an enhanced version of CCEH [36] with several technique advancements (see §2.1.2). It uses 16 KB segments and 256-byte buckets (i.e., an XPLine) with two additional stash buckets.
- `SEPH`: This is our proposed hashing scheme. To compress two PS pointers into one 8-byte word, we implement a segment allocator that supports atomic aligned segment allocation and crash consistency for SEPH. Besides, we set the size of a PS to 16 KB (which is also the segment size of CCEH and Dash used in the evaluation), and thus, the total size of an LS is 24 KB. Moreover, SEPH employs one dedicated background thread to update the bucket index foreseer.

For the sake of fairness, since the more recent PM hashing schemes (e.g., Clevel Hashing, Dash, and our proposal) support variable-length keys and values, all the evaluated hashing schemes are unified to only keep the pointers to KV items in slots. Besides, the length of a persistent pointer in PMDK is 16 bytes (i.e., 8 bytes for the base address of a PM pool and 8 bytes for the offset in pool), which cannot be operated by the CAS atomic instruction. To resolve this issue, Clevel Hashing [10] proposes to only maintain the offset in the PM pool as an 8-byte persistent pointer, since the base address of a PM pool will be fixed once the pool is mapped. We also apply this offset-only pointer to all the evaluated hashing schemes.

Benchmark. For the micro-benchmarks used in §4.2, we first warm up the hash table with 10 millions of KV items, followed by executing a total number of 200 millions of operations unless otherwise stated. Particularly, workloads composed of a single type of operations are evaluated in §4.2.1~§4.2.3 and workloads mixed with multiple types of operations are used in §4.2.4, where all these workloads are generated by YCSB [11] in Zipf distribution with 0.99 skewness. As for the macro-benchmarks presented in §4.3, we use the real-world workloads from YCSB [11]. Particularly, in the load phase, we populate 64 millions of KV items, following Clevel Hashing [10]; then, the standard YCSB workloads A, B, C, D, and F are conducted with 48 threads. Notably, the standard YCSB workload E is not evaluated since none of the hashing schemes optimizes the range query performance. Besides, the lengths of key and value are both set to 16 bytes since it is widely used [6], and the KV items are pre-generated before the testing as [33].

4.2 Micro Benchmark

4.2.1 Performance Efficiency and Predictability

To analyze the advantages of SEPH, we first focus on the insertion performance. Figure 7 plots the real-time insertion throughput of different hash tables under 48 threads. It can be clearly observed that SEPH outperforms all the other schemes

in terms of performance efficiency. Specifically, SEPH outperforms Dash, CCEH, and CCEH-C by 2.12×, 4.31×, and 4.46× for average insertion throughput respectively, and achieves at least 8.27× higher average throughput compared with Level-based hashing schemes and PCLHT.

As for the performance predictability, SEPH demonstrates the most superior worst-case real-time throughput (i.e., the minimal real-time throughput) than all the other evaluated hashing schemes, as revealed by Figure 7. Specifically, the minimal real-time throughput of SEPH is higher than that of Clevel, CCEH, CCEH-C, and Dash by 9.34×, 4.40×, 4.74× and 5.23× respectively, while PCLHT and Level Hashing even suffer zero real-time throughput because their full-table resizing are conducted in a blocking manner. Moreover, it is also worth noting that the minimal real-time throughput of SEPH is even higher than the maximal real-time throughput of all the other evaluated hashing schemes by from 1.06× to 5.76×. This reveals that SEPH achieves remarkable performance predictability by delivering an excellent worst-case real-time throughput under the insertion-intensive workload.

The tail latency of different percentiles is another perspective to show the performance by reflecting the response time. A design with good performance predictability requires a low bound of the latency on high percentiles (i.e., tail latency). Figure 8 shows the evaluation of the insertion latency at different percentiles. In general, SEPH significantly cuts down the 100th-percentile insertion latency compared with PCLHT/Level-based hashing schemes (by 3 ~ 4 orders of magnitude) and is superior to all the EH-based hashing schemes on every percentile of insertion latency. Especially, it can be noticed that in contrast to EH-based hashing schemes (i.e., CCEH, CCEH-C and Dash) that have a sharp raising of insertion latency at the 99.9th percentile, the insertion latency of SEPH rises at the 99.99th percentile. The rationale behind this is that SEPH triggers a less number of split operations than EH-based hashing schemes, since the size of LS in SEPH (i.e., 24 KB) is larger than the segment size (i.e., 16 KB) of EH-based hashing schemes. Despite this, SEPH still achieves 9.75× ~ 11.36× lower latency at both 99.99th and 99.999th percentiles than EH-based hashing schemes; Also, the 100th-percentile latency of SEPH is lower than that of EH-based hashing schemes by from 3.62× to 5.86× because SEPH offloads the directory doubling to the background thread.

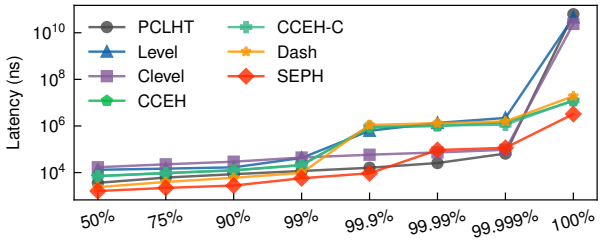


Figure 8: Latency at Different Percentiles.

Figures 9(a) and 9(b) further disclose the key reasons behind the improvements in performance efficiency and predictability achieved by SEPH. On the one hand, thanks to the one-third splitting and the dereference-free rehashing, SEPH introduces 8.11 ~ 43.78× less time for resizing than all the other evaluated hashing schemes as shown in Figure 9(a). On the other hand, Figure 9(b) validates the efficacy of the semi-lock-free concurrency control in minimizing the PM writes. Specifically, SEPH significantly reduces the PM writes by 2 ~ 3.33× and nearly approaches the expected, optimal amount of PM writes.

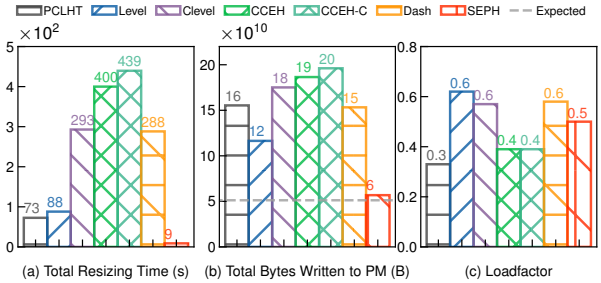


Figure 9: Resizing Time, Total Bytes Written, and Load Factor.

Load Factor. Figure 9(c) further depicts the result of average load factors. Considering that Dash leverages non-trivial memory PM (i.e., 12.5%) to store the metadata to improve the performance, the calculation of the load factor of Dash excludes this amount of memory. The average load factor of Level Hashing and Dash are both up to 59% because they adopt multiple optimizations such as one movement, balancing load, stash buckets, etc. The average load factor of SEPH is 50%, which is because SEPH does not adopt the optimization that improves the load factor at the expense of introducing more operational overhead. On the other hand, our design shows higher load factor compared to CCEH and CCEH-C (both 39%) since SEPH can accommodate more KV items in a bucket (i.e., 32 for SEPH).

4.2.2 Performance Breakdown

To investigate how the key techniques of SEPH introduced in §3.2 and §3.3 contribute to the overall performance improvement, Figure 10 shows the experiment results of inserting 200 millions of KV items using 48 threads on different variants of SEPH (see Table 2 for their detailed configurations).

Table 2: Configuration of Different SEPH Variants.

SEPH Variants	Semi Lock-Free	One-Third Splitting	Dereference-Free Rehashing
SEPH-Base	×	×	×
SEPH-S	✓	×	×
SEPH-SO	✓	✓	×
SEPH-SOD	✓	✓	✓

First of all, SEPH-Base is considered as a baseline design of SEPH which only keeps the level segment based hash table of SEPH (presented in §3.1) but does not equip with any low-overhead split techniques (proposed in §3.2) and even adopts the lock-based scheme of Dash [33]. It can be clearly observed from Figure 10(a) that compared with SEPH-Base, SEPH-S (which adopts only the proposed semi lock-free concurrency control) significantly lifts up the average

and highest throughputs by 56.72% and 70.20% respectively. The rationale behind this is that the proposed semi lock-free concurrency control effectively avoids the PM writes required to manipulate locks (as validated by Figure 10(b)), resulting in better performance efficiency and scalability. Secondly, compared with SEPH-S, SEPH-SO further demonstrates the efficacy of the proposed one-third splitting in raising the worst-case throughput by 71.46% and reducing the total resizing time by 51.04% (as shown in Figure 10(c)). Finally, compared with SEPH-SO, SEPH-SOD (i.e., the complete design of SEPH) ultimately shows how the proposed dereference-free rehashing can amazingly minimize the total resizing overhead by 92.50%, resulting in a further improvement in the worst-case throughput (i.e., performance predictability) by 55.85%.

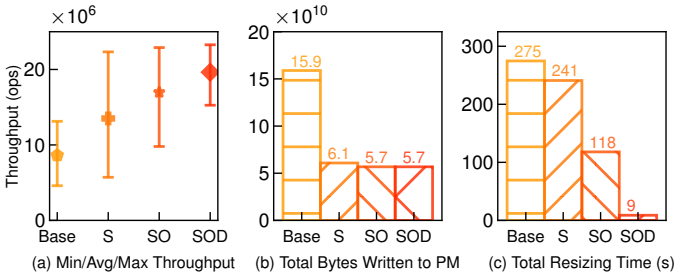


Figure 10: Breakdown of Different SEPH Variants.

4.2.3 Performance Scalability

Insertion. Figure 11(a) shows the insertion throughput under different number of threads. SEPH speeds up the insertion performance by 2.2× under 48 threads and by 2× under the other thread numbers compared with Dash. The main reason is that the write bandwidth of PM is considered a common bottleneck, and SEPH completes the insertions with less consumption of write bandwidth. By contrast, PCLHT and Level Hashing show poor scalability owing to the blocking resizing, while the low scalability of Clevel Hashing is due to the high consumption of read/write bandwidth for the full lock-free design.

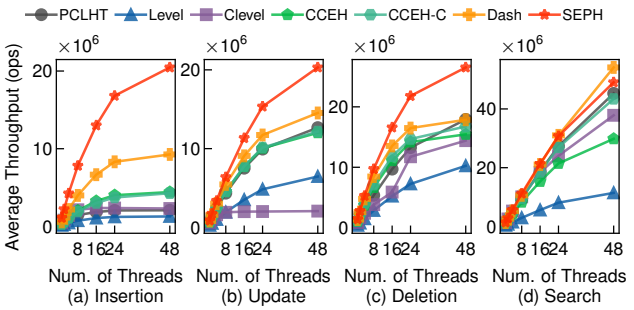


Figure 11: Scalability.

Update and Deletion. Figures 11(b) and 11(c) present the update and deletion performance of various hashing schemes under different numbers of thread. Although SEPH shows similar performance on update operations with Dash when the number of the threads is less than 16, SEPH outperforms Dash by 30% at 24 threads and 39% at 48 threads, demonstrating higher performance scalability. This is because the lock-free design of SEPH provides a more scalable update performance by reducing a write for the lock to avoid hitting the limit of the write bandwidth at lower concurrent scenarios. To test the scalability of delete operations, we run 10 millions of delete operations to delete all the pre-loaded 10 millions of KV items. As shown in Figure 11(c), SEPH also surpasses all the other designs at 48 threads in delivering high performance scalability of deletion, thanks to the reduction in PM write achieved by the semi lock-free concurrency control.

Search. As shown in Figure 11(d), Dash, SEPH and PCLHT show good scalability on search performance, thanks to the low-overhead search operations of these designs and the high bandwidth of PM read (compared with write bandwidth). The search throughput of SEPH is 9.1% lower than that of Dash with 48 threads because SEPH needs to access two candidate buckets by two random reads, yet Dash accesses two candidate buckets by two sequential reads.

4.2.4 Mixed Workload

In order to evaluate the performance behavior of the different hashing schemes under the realistic mixed workload, we conduct the experiments with the mixed requests of different search/insertion ratio generated by YCSB in the zipfian distribution (0.99 skewness).

Figure 12 shows the real-time throughput of different hashing schemes with different mixed workloads under 48 threads. SEPH performs the mostly-highest and the least-fluctuated performance in these workloads, which demonstrates SEPH can achieve good performance efficiency and good performance predictability at the same time under the evaluated workloads mixed with different percentages of search operations (denoted as "S") and insert operations (denoted as "I").

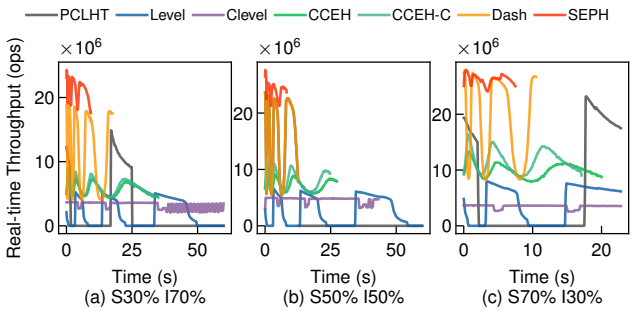


Figure 12: Real-Time Performance under YCSB Workloads with Different Search/Insert Ratio (%).

Figure 13 shows the results of the scalability of the hash tables under different mixed workloads. It can be observed that SEPH delivers better performance scalability than any other evaluated hashing schemes, even under the workload mixed with a high percentage of insertions (i.e., 70% of insert operations and 30% of search operations shown in Figure 13(a)). This is because the proposed semi lock-free

concurrency control entails a nearly-minimal amount of PM writes to handle an insertion operation.

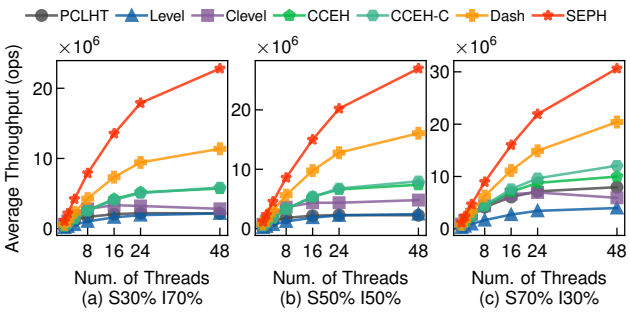


Figure 13: Performance Scalability under YCSB Workloads with Different Search/Insert Ratio (%).

4.3 Macro Benchmark

Figure 14 shows the performance results of executing the standard YCSB workloads in terms of the minimal, average, and maximal throughputs. First of all, under the workload Load (100% insertion) shown in Figure 14(a), SEPH performs 2.63× ~ 19.59× higher average throughput and at least 4.52× higher minimal throughput than any other evaluated hashing schemes. This demonstrates SEPH can deliver the most superior performance efficiency and predictability under the insertion-intensive workload, with the help of the proposed low-overhead splitting and semi lock-free concurrency control.

However, under the workload C (100% search) shown in Figure 14(d), the average throughput of SEPH is 12.83% lower than that of Dash, since SEPH is not optimized for search operations (as discussed in §4.2.3). But it is encouraging to see that with the increasing of update operations, the performance gap between the average throughputs of SEPH and Dash reduces and even reverses, because the proposed semi lock-free concurrency control avoids the PM writes to manipulate locks for update operations. Particularly, under the workload B (95% search & 5% update) and the workload F (95% search & 5% read-modify-write (RMW)), the average throughput of SEPH only falls behind that of Dash by 10.9% and 5.6%, as shown in Figures 14(c) and 14(f), respectively. On the contrary, under the workload A (50% search & 50% update), the average throughput of SEPH overtakes that of Dash by 6.9% as in Figure 14(f).

More importantly, under the workload D (95% search & 5% insertion) shown in Figure 14(e), even though SEPH does not achieve the best average throughput (due to the high portion of search operations) among all the evaluated hashing schemes, SEPH demonstrates the best performance predictability (i.e., improving the minimal throughput by at least 39.50%). This reveals the value of the proposed low-overhead splitting in reducing the resizing overhead, even if there are only 5% of insertions. Figure 15 further shows the operation latency of the evaluated hashing schemes at different percentiles under the workload D. It can be observed that SEPH outperforms

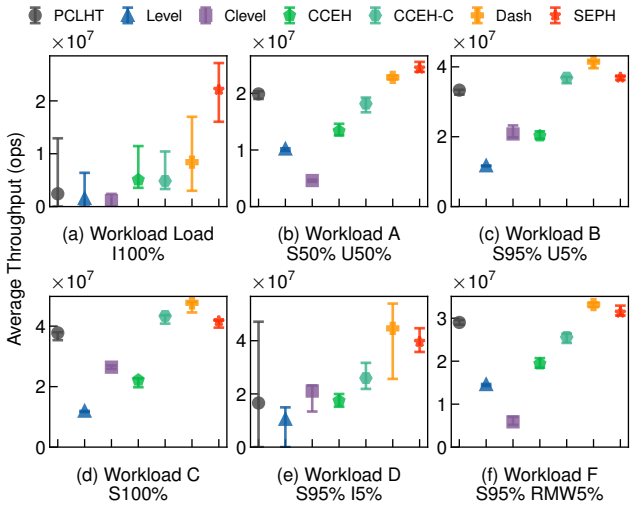


Figure 14: The Minimal, Average, and Maximal Throughputs under Standard YCSB Workloads.

EH-based designs by at least 11× and 1.82× for the 99.999th and 100th percentile latency respectively; additionally, SEPH greatly surpasses Level-based designs and PCLHT by at least 2792× for the 100th percentile latency.

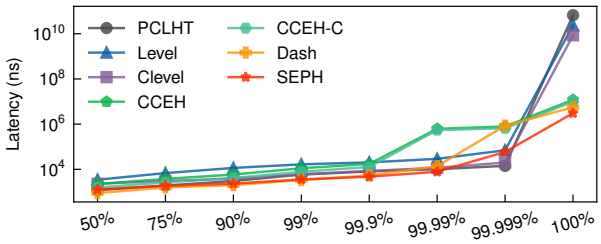


Figure 15: Latency at Different Percentiles (Workload D).

5 Conclusion

This paper presents SEPH, a Scalable, Efficient, and Predictable Hashing for PM. To break the dilemma between efficiency and predictability, SEPH introduces a new structure called level segment (LS) to build the hash table with a unique indexing mechanism. SEPH further enables a low-overhead split operation to significantly suppress the resizing-incurred performance unpredictability, and puts forward a semi lock-free concurrency control that requires a nearly-minimal amount of writes to handle an insertion operation for achieving ever-higher efficiency and scalability while ensuring the correctness and crash consistency. Our results reveal that SEPH achieves higher efficiency, better scalability, and more reliable predictability when compared with state-of-the-art hashing schemes for PM.

Acknowledgements

We thank our shepherd, Ashvin Goel, and all the anonymous reviewers for their valuable suggestions. This work is supported in part by The Research Grants Council of Hong Kong SAR (Project Nos. CUHK14210320 and CUHK14208521).

References

[1] Extended asynchronous dram refresh (eadr), https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[2] Intel corporation. intel 64 and ia-32 architectures software developer's manual, https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[3] Intel optane dc persistent memory module, https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html, 2019.

[4] Intel architecture instruction set extensions programming reference, https://www.intel.com/content/www/us/en/developer/overview.html#gs.jevnd1, 2021.

[5] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment*, 11(5):553–565, 2018.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[7] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. *Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated*, page 487–498. Association for Computing Machinery, New York, NY, USA, 2011.

[8] Daniel Bittman, Darrell DE 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)*, pages 17–30, 2019.

[9] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[10] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. Lock-free concurrent level hashing for persistent memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 799–812. USENIX Association, July 2020.

[11] 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*, pages 143–154, 2010.

[12] Biplob Debnath, Alireza Haghdoost, Asim Kadav, Mohammed G. Khatib, and Cristian Ungureanu. Revisiting hash table design for phase change memory. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Carla Schlatter Ellis. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 106–116, 1983.

[14] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 371–384, 2013.

[15] Philippe Flajolet, Patricio Poblete, and Alfredo Viola. On the analysis of linear probing hashing. *Algorithmica*, 22(4):490–515, 1998.

[16] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[17] H. Gao and W.H. Hesselink. A general lock-free algorithm using compare-and-swap. *Information and Computation*, 205(2):225–241, 2007.

[18] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, dec 1992.

[19] 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)*, pages 187–200, Oakland, CA, February 2018. USENIX Association.

[20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman 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.

[21] 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, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Giorgos Kappes and Stergios V. Anastasiadis. A lock-free relaxed concurrent queue for fast work distribution. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 454–456, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers accelerating index traversals for in-memory databases. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 468–479, 2013.

[24] Per-Ake Larson. Dynamic hash tables. *Commun. ACM*, 31(4):446–457, apr 1988.

[25] 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)*, pages 257–270, Santa Clara, CA, February 2017. USENIX Association.

[26] 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, page 462–477, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. Loda: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 263–279, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Junkai Liang and Yunpeng Chai. Cruisedb: An lsm-tree key-value store with both better tail throughput and tail latency. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1032–1043, 2021.

[29] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 21–35, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Witold Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the Sixth International Conference on Very Large Data Bases - Volume 6*, VLDB '80, page 212–223. VLDB Endowment, 1980.

[31] Jihang Liu, Shimin Chen, and Lujun Wang. Lb+trees: Optimizing persistent index performance on 3dxpoint memory. *Proc. VLDB Endow.*, 13(7):1078–1090, March 2020.

[32] Yujie Liu and Michael Spear. A lock-free, array-based priority queue. *ACM SIGPLAN Notices*, 47(8):323–324, 2012.

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

[34] Chen Luo and Michael J. Carey. On performance stability in lsm-based storage systems. *Proc. VLDB Endow.*, 13(4):449–462, dec 2019.

[35] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. {ROART}: Range-query optimized persistent {ART}. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, pages 1–16, 2021.

[36] 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)*, pages 31–44, Boston, MA, February 2019. USENIX Association.

[37] ORACLE. Architectural overview of the oracle zfs storage appliance, https://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/o14-001-architecture-overviewzfsa-2099942.pdf, 2018.

[38] Swapnil Patil and Garth Gibson. Scale and concurrency of giga+: File system directories with millions of files. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, San Jose, CA, February 2011. USENIX Association.

[39] Yaqiong Peng and Zhiyu Hao. Fa-stack: A fast array-based stack with wait-free progress guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):843–857, 2018.

[40] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, page 19–es, USA, 2002. USENIX Association.

[41] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, may 2006.

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

[43] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508, 2020.

[44] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery.

[45] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[46] 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 *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, February 2015. USENIX Association.

[47] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, nov 2014.

[48] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 897–912, 2019.

[49] Pengfei Zuo and Yu Hua. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):985–998, 2018.

[50] 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)*, pages 461–476, Carlsbad, CA, October 2018. USENIX Association.