PMEH: A Parallel and Write-Optimized Extendible Hashing for Persistent Memory

Jing Hu, Jianxi Chen, *Member, IEEE*, Yifeng Zhu, *Member, IEEE*, Qing Yang, Zhouxuan Peng, and Ya Yu

***Abstract*—Emerging persistent memory (PM) has the potential to substitute DRAM due to its near-DRAM performance and durability similar to disks. However, hash tables designed for DRAM cannot be directly adopted for PM. Moreover, prior studies on hash tables using Optane DC PM modules (DCPMMs) have shown suboptimal scalability and write performance due to expensive lock-based concurrent control and massive data movement caused by expansion. In this article, we propose an opportunistic lock-free parallel multisplit extendible hashing scheme (PMEH). First, PMEH achieves lock-free operations for evenly distributed data by partitioning the hash table into multiple zones and assigning each zone to one thread. Second, PMEH employs an opportunistic lock-free parallel scheme to effectively handle skewed data distribution, which maximizes the utilization of lock-free operations by enabling dynamic switching between lock-free and locking operations. Finally, PMEH uses multisplit with gradual splitting, instead of 2-split, to reduce the frequency of hash table expansion and, hence, reduce the data movement during expansion. The experimental results under the widely used YCSB workloads demonstrate that PMEH achieves excellent scalability regardless of data distribution. Moreover, PMEH significantly speeds up insertions by 1.44×–15.4×, and deletion by 2.04×–18.07× compared to other state-of-the-art hashing schemes. In addition, PMEH reduces at least 52% of extra writes while providing instant recovery.**

***Index Terms*—Data consistency, extendible hashing, instant recovery, opportunistic lock-free, persistent memory (PM).**

I. Introduction

EMERGING persistent memory (PM) technologies, such as phase change memory (PCM) [1], 3-D XPoint [2], and spin-transfer torque memory (STT-RAM) [3], offer several advantages, including byte addressability, high density, near-zero standby power consumption, and instant recovery. Therefore, PM has the potential to replace or supplement DRAM in certain applications. In 2019, Intel released the first commercial PM product, the Intel Optane DC PM module (DCPMM) [4]. DCPMM has three times the read latency and one-third the read bandwidth compared to traditional DRAM, but similar write latency and one-sixth the write bandwidth [5], [6], [7], [8], [9], [10].

As emerging PM technologies change the DRAM-dominated memory system, index structures originally designed for DRAM should be revisited due to their significant differences in characteristics. Consequently, many indexing structures have been proposed for PM, such as trees [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22] and hash tables [23], [24], [25], [26], [27]. Trees have an average lookup time complexity of $O(\log(N))$, where N is the size of the structure. Unlike tree-based index structures, hash tables, with a flat structure, have a constant-scale lookup time complexity of $O(1)$, making them widely used as fundamental components in main memory databases [28], [29], [30], [31], [32], [33] and in-memory key–value store [34], [35], [36], [37], [38], [39], [40]. However, maintaining hashing index structures in PM poses multiple nontrivial challenges that are not yet solved by existing research.

A. Low Scalability for Expensive Lock-Based Overhead or Inefficient Lock-Free Design

A scalable PM-based hashing scheme is necessary to efficiently utilize the hardware resources and provide high throughput since a server node may have tens of or even hundreds of threads. However, prior studies have utilized expensive locks for concurrent control. For instance, Level hashing [24] employs a slot lock for read/write operations, while dynamic hashing techniques like Dash [26] and CCEH [25] use locks for buckets and slots, respectively. Moreover, during directory expansion, both Dash and CCEH require a directory lock to update directory entries, further increasing the overhead. However, even with a lock-free design, CLevel hashing [27] demonstrates inferior scalability since it utilizes only a single thread for background resizing, leading to a bottleneck. Hence, existing hashing schemes suffer from low scalability.

B. Load Imbalance for Partition Design

Partition design, enabling each core to deal with different data, is usually adopted to avoid the use of lock and reduce the interthread interference [41], [42]. However, the partition design has an inherent issue of potentially causing imbalanced loads. Although hash functions can distribute inserted key–value pairs into a nearly uniform distribution for skewed data distribution, they cannot distribute key–value pairs evenly for

Manuscript received 28 October 2022; revised 10 March 2023; accepted 20 April 2023. Date of publication 28 April 2023; date of current version 20 October 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 61832020 and Grant 61821003; in part by the Open Project of Key Laboratory of Ministry of Industry and Information Technology for Basic Software and Hardware Performance and Reliability Measurement under Grant HK202201317; and in part by the Engineering Research Center of Data Storage Systems and Technology, Ministry of Education, China. This article was recommended by Associate Editor C. Yang. *(Corresponding author: Jianxi Chen.)*

Jing Hu, Jianxi Chen, Qing Yang, Zhouxuan Peng, and Ya Yu are with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: hujingreal@hust.edu.cn; chenjx@hust.edu.cn; oxygen@hust.edu.cn; hustpzx@hust.edu.cn; ayu@hust.edu.cn).

Yifeng Zhu is with the Department of Electrical and Computer Engineering, College of Engineering, University of Maine, Orono, ME 04469 USA (e-mail: yifeng.zhu@maine.edu).

Digital Object Identifier 10.1109/TCAD.2023.3271579

search operations since these operations target specific positions. Therefore, hash tables with partition design can suffer from load imbalance issues, leading to inferior performance and scalability.

C. Excessive Data Movement During Resizing

Memory writes in PM will consume limited bandwidth (1/6 of DRAM), as well as introduce extra memory barriers and cache lines flushes overhead. Hence, reduce write matters in PM. Existing hashing tables [24], [25], [26], [27] have taken measurements to reduce extra writes caused by dealing with hash collisions (different keys are mapped into the same location by hash functions), such as reducing the number of kicking out and using spare buckets instead of linked lists to store collision data. However, these hashing schemes do not reduce extra writes caused by data movement during hash table expansion.

To address these challenges, this article proposes PMEH, an opportunistic lock-free parallel multisplit extendible hashing index scheme designed for PM. Specifically, this article makes the following contributions.

Partition Design in Hash Table: PMEH achieves lock-free access to an extendible hash table through partitioning. In an extendible hash table, two components: 1) the directory and 2) the segment, are shared for thread-parallel access. To eliminate segment locks, PMEH partitions the hash table into different zones and binds each zone to a specific thread. This partitioning approach eliminates segments as critical areas for parallel access by multiple threads. Furthermore, on the basis of partition, PMEH eliminates directory lock during directory expansion by introducing an extra directory array that can temporarily accommodate multiple directories. This approach enables efficient and concurrent directory expansion without locking the directory and impacting thread-parallel access (Section III-B).

Opportunistic Lock-Free Parallel: PMEH employs an opportunistic lock-free parallel scheme to effectively handle load imbalances. This approach maximizes the utilization of lock-free operations by enabling dynamic switching between lock-free and locking operations. Since hash functions distribute data evenly, in the majority of cases, each zone is accessed only by its designated thread with lock-free operations. However, in rare cases of load imbalance, idle threads are assigned to handle access to the busy zones with lock-based concurrent control to improve performance and scalability (Sections III-C and III-D).

Multisplit With Gradual Splitting: PMEH uses a multisplit approach with gradual splitting, rather than the traditional 2-split method, to expand the hash table. By using multisplit, the frequency of hash table expansions is reduced, which leads to fewer data movements required during expansion. However, using multisplit may cause the load factor (the number of inserted items divided by the capacity) to decrease drastically since the hash table capacity increases x (x-split) times that of the original table. To address this issue, PMEH adopts gradual splitting to ensure a smoother split and avoid a significant decrease in the load factor (Section III-E). Additionally, PMEH achieves instant recovery by recording crash positions and states with minimal metadata (Section III-F).

Real Implementation and Evaluation: PMEH is implemented and evaluated on a Linux system with DCPMM. Experimental results show that PMEH increases the performance of insertion and deletion up to 1.44×–15.4× and 2.04×–18.07×, respectively, compared to state-of-the-art hashing schemes. Additionally, PMEH reduces extra writes by at least 52% and achieves instant recovery.

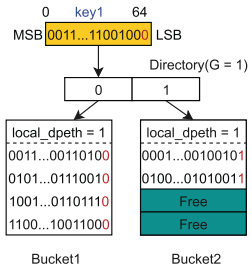

Fig. 1. Extendible hashing before inserting key1.

II. Background

A. Extendible Hashing

Extendible hashing is a dynamic hash table that can incrementally expand the hash table without requiring a full-table rehashing at once [43]. As shown in Fig. 1, extendible hashing consists of a set of buckets and an array called directory, which stores bucket addresses. Buckets can be added or deleted dynamically, and the directory is indexed using either the most significant bits (MSBs) or least significant bits (LSBs) of the hash values of keys. The number of entries in the directory is defined as global depth, denoted as G, and equals 2^G. The number of common bits in a bucket's hash value is defined as local depth, denoted as L. The number of directory entries pointing to a bucket can be calculated by 2^{G-L}. When a bucket becomes full, it is split to accommodate new key–value pairs. If multiple directory entries point to the split bucket, i.e., $L < G$, the bucket can be split into two buckets directly. Otherwise, the directory needs to be expanded first, and then the bucket is split.

As shown in Fig. 1, the directory in extendible hashing is indexed using the LSBs of the hash values, and each bucket can store up to four key–value pairs. When inserting the key1 into the hash table, it should be inserted into bucket1 since the LSB of the hash value of key1 is 0. However, since bucket1 is full and is only pointed by one directory entry, the current directory needs to be expanded to accommodate a new bucket for key1. Fig. 2 illustrates the expanded hash table with key1 inserted, where the global depth becomes two, and Bucket3, which is split from bucket 1, is now pointed to by a new directory entry. Subsequently, When inserting key2 with LSB equal to 01, it can be inserted directly into the bucket, which has two spare slots. Although the introduction of the directory in extendible hashing incurs additional access overhead, studies have shown it has little impact on performance compared to conventional hashing schemes [44], [45].

TABLE I
COMPARING PMEH WITH STATE-OF-THE-ART CONCURRENT HASHING SCHEMES DESIGNED FOR PM. (FOR MEMORY EFFICIENCY AND CRASH CONSISTENCY, "✓," "-," AND "X" INDICATE SUPERIOR, MODERATE, AND INFERIOR, RESPECTIVELY)

	Concurrency control			Reducing extra writes		Memory Efficiency	Crash Consistency
	Search	Insertion/Deletion	Resizing	Insertion	Resizing		
PMEH	Lock-free	Opportunistic Lock-free	Lock-free	✓	✓	✓	✓
Dash	Lock-free	Bucket writer lock	Global directory lock	✓	-	✓	✓
CLEVEL	Lock-free	Lock-free	Lock-free, single thread	✓	x	✓	✓
CCEH	Segment reader lock	Segment writer lock	Global directory lock	✓	-	✓	✓
LEVEL	Slot lock	Slot lock	Global metadata lock	✓	x	✓	✓

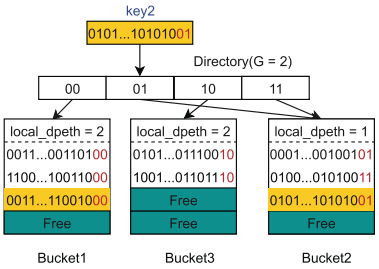

Fig. 2. Extendible hashing after inserting key1 and key2.

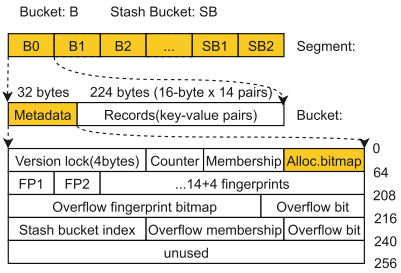

Fig. 3. Segment structure for dash.

B. Hashing Index Schemes on PM

Recently, various hashing index schemes have been proposed for emerging PM. These hashing schemes can be broadly classified into two categories: 1) static hashing, such as Level hashing and CLevel hashing and 2) dynamic hashing, such as CCEH and Dash. Static hashing can achieve constant lookup time with a flat structure. However, the size of hash tables must be estimated in advance [25]. Otherwise, the entire table or 1/3 of the table will need to be rehashed when the table is full, which can result in significant performance overhead. In contrast, dynamic hashing offers a robust ability for on-demand expansion and shrinking, which is particularly valuable for workloads that display high variability in data access patterns. When the hash table needs to be expanded, dynamic hashing only needs to rehash a small part of data rather than the entire table, which minimizes the performance overhead of resizing.

1) Static Hashing: Level hashing [24] contains two levels, where each level is an array of 4-slot buckets, and each slot can store one key–value pair. The capacity of the top level is twice that of the bottom level. To reduce extra writes, the top level is used to address items, while the bottom level is used to store records suffering from hash collisions. Moreover, only one kick-out operation is allowed to avoid cascading writes. A bit, used as a token, is associated with each slot to ensure data consistency. To improve memory efficiency, two hash functions are used to address buckets. Therefore, it has four buckets (16 slots in total for two levels) for data insertion. Additionally, only 1/3 of the buckets, instead of the entire table, need to be rehashed to expand the table. However, Level hashing incurs the same rehashing cost as other hashing schemes in practice [25]. For scalability, slot lock is imposed for search, insertion, and deletion, and global metadata lock is imposed for resizing, as shown in Table I. Hence, Level hashing has limited scalability since it suffers from severe lock contention overhead. Furthermore, while Level hashing significantly reduces the number of extra writes during the insertion, it does not reduce the number of extra writes during the resizing.

CLevel hashing [27] is a variant of Level hashing that uses a lock-free scheme to eliminate the expensive overhead of lock contention. To avoid blocking normal access, the rehashing process of Clevel is put in a background thread. Although Clevel eliminates the lock overhead, it still has limited scalability, as shown in Section IV-C. The reason is that only a single thread is used to perform the rehashing operation, which can easily create a performance bottleneck. In addition, Clevel may produce more duplicate keys due to background rehashing and crashes during insertion, requiring additional checks to prevent such duplicates. Furthermore, the number of extra writes during resizing is also not reduced in Clevel hashing.

2) Dynamic Hashing: CCEH [25], based on extendible hashing, adopts a cache line size bucket to improve cache efficiency. A set of buckets is organized into an intermediate level, called *segment*, to reduce the directory size. The crash consistency of CCEH is guaranteed by an atomic write to the 8-byte key. Only linear probing is used to deal with hash collision, and the linear probing distance is four buckets. Therefore, the space utilization of CCEH is inferior [26]. As summarized in Table I, the scalability of CCEH is limited by the severe overhead of lock contention. This is because the segment reader lock is used for search operations, and segment writer lock is used for insertion and deletion. Moreover, the resizing process locks the entire directory. When a segment is split in CCEH, the old segment's split data is not directly deleted. Instead, it will be overwritten by subsequent writes instead of deleted directly. Although this lazy deletion method reduces extra writes for deletion, CCEH does not reduce the number of data movements from the old segment to the new segment.

Dash [26] is another variant of extendible hashing designed for PM. Dash uses 256-byte buckets to match PM's access granularity and introduces an intermediate segment structure similar to CCEH, as shown in Fig. 3. Each Dash segment comprises a fixed number of normal buckets and two stash

buckets, with an identical structure used for both normal and stash buckets. Each bucket includes 32-byte metadata and 224-byte key–value records, with a fingerprint introduced for each record to reduce the probing distance and ensure that the average probing distance is 1. Dash also includes an allocation bitmap in each bucket to ensure data consistency by indicating whether corresponding records have been fully persisted. Balanced insertion, displacement, and stashing are used to deal with hash collisions and improve memory efficiency, with metadata recorded in the origin bucket. Hence, Dash has superior space utilization. As shown in Table I, Dash adopts a writer lock for insertion and deletion but does not require a lock for the search operation. Besides, the entire directory needs to be locked when it expands, leading to moderate scalability. Like CCEH, Dash also does not reduce the number of data movements from old to new segments.

Unlike existing hashing schemes, PMEH achieves excellent scalability and performance by minimizing lock overhead through its opportunistic lock-free parallel scheme. Additionally, PMEH reduces data movement during expansion by utilizing multisplit with gradual splitting.

C. Crash Consistency in Persistent Memory

To ensure crash consistency when storing data in PM, it is necessary to prevent data from being lost or being in a semi-updated status. Since the maximum size of an atomic write supported by the CPU is small or does not exceed the width of the memory bus, data larger than 8 bytes for a 64-bit CPU may be only partially updated when written to PM [46]. While the logging or COW (copy-on-write) method can ensure consistency, they also incur extra writes. Hence, Most hashing schemes based on PM do not use logging or similar methods to ensure data consistency. Instead, They generally ensure crash consistency by limiting the key to 8 bytes or adding a one-bit bitmap flag, since the key and the bitmap can be written atomically.

On the other hand, memory barrier instructions and cache-line flush instructions are necessary to enforce memory order since the CPU and the memory controller may reorder memory instructions to optimize performance. The CPU provides memory barrier instructions (e.g., *sfence*, *lfence*, and *mfence*) and cacheline flush instructions (e.g., *clflush*, *clflushopt*, and *clwb*) [47]. These memory barrier instructions can prevent instructions from being reordered crossing the barrier. For instance, *mfence* ensures that memory instructions in front of the barrier are executed before the memory those behind the barrier, while *clflush* invalidates and flushes the corresponding dirty cache line back to the PM. However, those instructions incur expensive overhead, so their use should be minimized. This article uses the memory barrier instructions and cache-line flush instructions provided by PMDK [48] to ensure crash consistency.

D. Compare and Swap Primitive

Compare and swap (CAS) is a synchronization primitive used to realize uninterrupted data exchange operations in multithread programming. It is available in most modern processors. When multiple threads attempt to modify the same data simultaneously, CAS can guarantee data consistency even in the face of the uncertainty of execution order or the unpredictability of interruption. CAS takes three parameters: memory address, expected value, and new value. CAS first checks whether the expected value matches the data stored in the memory address. If they are equal, the new value is written to the memory address, overwriting the old value. Otherwise, CAS reads the current value stored in the memory address is read or does nothing else.

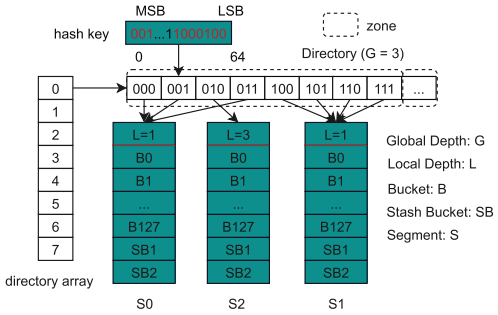

Fig. 4. PMEH overall architecture.

III. Design and Implementation

We propose PMEH, an opportunistic lock-free and multisplit hashing scheme, with excellent scalability and performance. In this section, we first present the overall structure of PMEH, which aims to deliver high scalability and reduce extra write (Section III-A). Next, we present a lock-free design for PMEH to eliminate the lock overhead for even data distribution (Section III-B). We then introduce lock-based concurrent control, which is used to assist the lock-free design to deal with skewed data distribution (Section III-C). After that, PMEH introduces an opportunistic lock-free scheme to maximize the use of lock-free operation by efficiently dynamically switching between lock-free and locking operation (Section III-D). We then present multisplit with gradual split to reduce extra write during hash table expansion (Section III-E). Finally, The crash recovery is present to guarantee data consistency on PM (Section III-F).

A. Structure of PMEH

PMEH consists of three main components: 1) the directory array; 2) directories; and 3) segments, as illustrated in Fig. 4. The directory array is an array of directory addresses that is used to avoid locking the directory during directory expansion (Section III-B). The directory itself is an array of segment addresses, and the segment is an intermediate layer that contains a fixed number of buckets. By using the MSBs to locate the segment and the LSBs to locate the bucket, PMEH reduces the overhead of the directory. The structure of the segment is illustrated in Fig. 5. The first 8 bytes of the segment represent its metadata. The first 6 bits of the metadata store the local depth (L), which records the number of common bits in the segment. The next 42 bits (pattern) record the value of

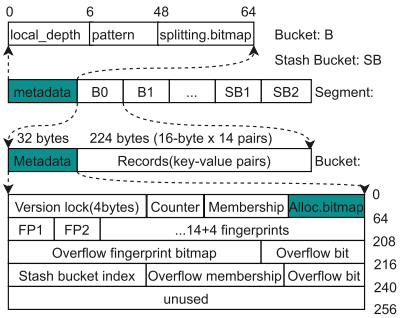

Fig. 5. Segment structure in PMEH.

common bits in the segment. The remaining 16 bits in the metadata represent the splitting bitmap. The splitting bitmap is used to track the newly allocated segments during segment gradual splitting. Specifically, each bit in the bitmap represents a new segment, and the bit is set to 1 if the corresponding segment has been split out during the splitting process (see Section III-E).

Furthermore, PMEH uses the same bucket structure and collision handling methods as Dash to achieve good load factor and search performance [26]. A bucket is composed of 32-byte metadata and 224-byte key–value records. A 4-bytes version lock is used for optimistic concurrent control, and a 4-bit counter is employed to record the number of records stored in the buckets. The membership indicates whether the record is hashed into this bucket or moved to this bucket by collision handling. The allocation bitmap indicates whether the slot holds a valid record by reserving one bit per slot. The fingerprints are used to speed up the search, with the front 14 fingerprints for normal records and the tail four fingerprints for overflowed records. When the records overflow, they are inserted into the stash buckets. The overflow fingerprint bitmap indicates whether the overflowed records are valid. The stash bucket index records which bucket the overflowed record is inserted into. The overflow membership field is similar to membership, indicating whether the record hashed into this bucket or moved to this bucket by collision handling. Finally, the overflow field represents the number of records that overflowed but do not record fingerprints.

B. Lock-Free Operations in PMEH

1) Lock-Free for Segments: PMEH eliminates lock overhead when each thread only deals with its own task. The hash table is divided into multiple zones, with each zone guaranteed to have at least one segment, as shown in Fig. 6. Each zone in PMEH is bound to a single thread, ensuring that each zone belongs to only one thread. The thread bound to a particular zone is referred to as the local thread for that zone, while all other threads are nonlocal. Therefore, before the hash table expands, each thread can access one segment in the zone bound to itself without a lock. This is because the segment is not the critical section for parallel access by multiple threads. The detail segment insert procedure is shown as Algorithm 1.

To ensure lock-free access even after the hash table expands, it is necessary to expand the range of each zone. As illustrated

Algorithm 1 Segment::Insert With Lock-Free

```
def Segment::InsertNoLock(key, value, bucket_id, finger):
    bucket=GetBucket(bucket_id)
    slot_id=bucket→GetEmptySlot()
    if slot_id<SlotNumPerBucket then
        slot=bucket[slot_id]
        slot.key=key
        slot.value=value
        Allocate::Persist(&bucket[slot_id],sizeof(bucket[slot_id])
        bucket.metadata.finger[slot_id]=finger
        bucket.metadata.bitmap=bucket.bitmap|(1≪slot_id)
        Allocate::Persist(&bucket.metadata,
        sizeof(bucket.metadata))
        return true
    else
        Normal bucket is full. Try to insert it into stash buckets
        if succeed in inserting into stash buckets then
            return true
        else
            # insert failure, need to expand the hash table
            return -1
        end if
    end if
```

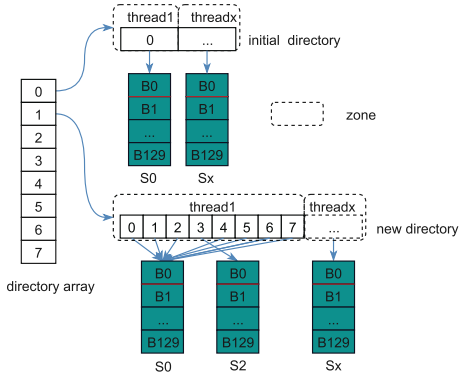

Fig. 6. Lock-free parallel accesses in PMEH.

in Fig. 6, directory 0 is the original directory and is indexed by the 1st position of the directory array. In directory 0, each segment is indexed by one directory entry, and each zone is bound to one thread. Therefore, when a segment becomes full, it needs to be split, resulting in the hash table expanding and the old directory expanding as well. To complete the segment splitting process, a new directory, directory 1, is allocated and indexed by the directory array's 2nd position. Meanwhile, the zone range must also be expanded to accommodate eight segments (8-split will split out seven new segments). This ensures that segment splitting only occurs in one zone, and ultimately eliminates the lock for a segment even if the hash table expands.

2) Lock-Free for the Directory: In order to achieve lock-free for the directory, PMEH allows for the existence of multiple directories temporarily. Although the lock for segments can be eliminated, the lock for the directory still exists. The reason is that generally, during the process of directory expansion, the entire old directory is required to be locked until the expansion process is finished. To eliminate this lock, PMEH uses an array to manage all directories and allows for the allocation of new directories directly without locking the old directory during expansion. Other threads can still access the old directory during this time. However, multiple threads may attempt to allocate a new directory simultaneously since they can access the old directory without blocking. To ensure

Algorithm 2 PMEH::Allocate New Directory With Lock-Free

```
def PMEH::AllocateDir(thread_id, dir_depth, dir_id):
    # allocate a larger directory atomically by PMDK
    Allocator::AllocDir(new_dir[thread_id], dir_depth)
    new_dir_id=dir_id + 1
    if CAS(dir[new_dir_id], nullptr, new_dir[thread_id]) then
        # succeed in adding new directory into directory array,
        # increase lastest_dir_id
        latest_dir_id=new_dir_id
    else
        # other thread has allocated and added a new directory,
        # free memory by the transaction of PMDK
        TX_BEGIN(Allocator::pm_pool){
            pmemobj_tx_add_range_direct(&new_dir[thread_id],
            sizeof(new_dir[thread_id]))
            Allocate::Free(new_dir[thread_id])
            new_dir[thread_id]=nullptr
        }
        TX_ONABORT
        TX_END
    end if
```

Algorithm 3 PMEH::Update Entries of New Directory

```
def PMEH::UpdateZoneEntries(zone_id, dir_id):
    tmp_id=dir_id - 1
    # let pre_dir point to the previous directory
    pre_dir=dir[tmp_id]
    # find a directory that points to a valid segment
    while pre_dir do
        # calculate the start entry id of the zone
        start_entry_id=GetZoneStartEntryID(pre_dir, zone_id)
        if pre_dir->_[start_entry_id] then
            break
        else
            tmp_id=tmp_id - 1
            pre_dir=dir[tmp_id]
        end if
    end while
    # current_dir point to the current directory that needs to update
    current_dir=dir[dir_id]
    start_entry_id=GetZoneStartEntryID(pre_dir, zone_id)
    end_entry_id=GetZoneEndEntryID(pre_dir, zone_id)
    start=start_entry_id
    while start<end_entry_id do
        update current_dir to point to the entry indexed by start
        in pre_dir, if the entries point to empty
        start=start + 1
    end while
    persist current_dir entries in zone[zone_id]
    # increase the number of having been updated zones
    old=current_dir→region_update_num
    new=old+1
    if CAS(&current_dir→region_update_num, old, new) then
        pememobj_persist(&current_dir→region_update_num,
        sizeof(current_dir→region_update_num));
        # if all entries have been updated, delete old directories
        if AllZoneUpdated(new) then
            delete previous directories before current_dir
        end if
    else
        new=old
    end if
```

Algorithm 4 PMEH::Insert With Lock-Free

```
def PMEH::InsertNoLock(key, value, segment, zone_id, dir_id,
segment_id, bucket_id, finger):
    if segment==nullptr then
        UpdateZoneEntries(zone_id, dir_id);
        segment=current_dir→_[segment_id]
    end if
    result=segment→InsertNoLock(key, value,
    bucket_id, finger)
    if result!=-1 then
        # insert success
        return result
    else
        if not all segment has been split out then
            do segment splitting
        else
            # allocate new directory,
            # expand_shift is split multiple
            dir_depth=current_dir→depth + expand_shift
            AllocateDir(thread_id, current_dir_id, expand_shift)
        end if
        return -1 # need retry
    end if
```

that only one thread successfully allocates the new directory, each thread uses the CAS primitive to add the new directory to the directory array. The CAS primitive ensures that only one thread can successfully add the new directory to the directory array, while the other threads that fail in the CAS operation need to free the allocated directory. This process is illustrated in Algorithm 2. In PMEH, the directory array size is set to 12. This is because only 43 bits are available (after subtracting the addressing bits of the initial directory (6 bits) and segment (7 bits), and 8 bits for the fingerprint) in the 64-bit space that can be used for directory expansion. In addition, PMEH reduces the number of required directories by using multisplit (Section III-E) and adopts 16-split as the optimal configuration (Section IV-B). As a result, a maximum of $\lceil 43/4 + 1 \rceil = 12$ directories can be generated.

To minimize the delay caused by expansion, PMEH amortized entries updated for the new directory into subsequent insertions rather than searches. This is because search operations are more sensitive to latency than insert operations. When inserting a key–value record into the hash table, the entries from the previous directory are searched to locate the valid segment if the indexed entry in the latest directory is null. Then, the directory entries in the latest directory are updated to index the valid segment. To improve performance, directory entries in the current zone, rather than the entire directory or a single directory entry, are updated to point to valid segments. Once all zones in the latest directory have been updated, all previous old directories are freed. In fact, since the hash function aims to evenly distribute the inserted key–value pairs, each segment is quickly indexed by the new directory, which results in the old directories being released quickly. The entries update process is illustrated in Algorithm 3. Algorithm 4 presents the entire lock-free insert algorithm.

The following is an example of how the hash table is expanded in the hash table. Initially, the directory 0 is indexed by the 1st position in the directory array, as shown in Fig. 6. Each segment is only indexed by one directory entry, and segment 0 belongs to thread 1 in directory 0. when segment 0 (S0) becomes full, thread 1 initiates the expansion process by allocating a new larger directory and adding it to the directory array using CAS primitive. The new directory, directory 1, is then indexed by the 2nd position in the directory array. When inserting a record into segment 0, Thread 1 will retrieve empty entries from directory 1. Therefore, thread 1 will first search directory 0 for a valid segment and update entries of the current zone in directory 1 to index valid segments. Similarly, entries of other zones in directory 1 will be updated after other threads finish their normal access. After all entries in directory 1 have been updated, directory 0 will be deleted.

Algorithm 5 PMEH::Get With Lock-Free

```
    def PMEH::Get(key):
 2:     hash_key=Hash(key)
        finger=GetFinger(hash_key)
 4:     bucket_id=GetBucketId(hash_key)
        RETRY:
 6:     # locate segment
        current_dir_id=latest_dir_id
 8:     current_dir=dir[current_dir_id]
        segment_id=GetSegmentId(hash_key,current_dir→depth)
10:     segment=current_dir→_[segment_id]
        while segment==nullptr do
12:         # find previous directory
            current_dir_id–
14:         current_dir=dir[current_dir_id]
            segment_id=GetSegmentId(hash_key,current_dir→depth)
16:         segment=current_dir→_[segment_id]
        end while
18:     result= segment→Get(key, finger, bucket_id)
        if result==-1 then
20:         # other thread change value
            goto RETRY
22:     else
            # other thread updated zone entries
24:         t=current_dir_id++
            while t<latest_dir_id do
26:             s=GetSegmentId(hash_key,dir[t]→depth)
                if dir[t]→_[s]==segment then
28:                 goto RETRY
                end if
30:         end while
            return result
32:     end if
```

Algorithm 6 PMEH::Insert With Lock

```
    def PMEH::InsertWithLock(key, value, segment, zone_id, dir_id,
    segment_id, bucket_id, finger):
 2:     if segment==nullptr then
            GetZoneLock(zone_id)
 4:         UpdateZoneEntries(zone_id, dir_id);
            segment=current_dir→_[segment_id]
 6:         ReleaseZoneLock(zone_id)
        end if
 8:     result=segment→InsertWithLock(key, value,
        bucket_id, finger)
        if result!=-1 then
10:         # insert success
            return result
12:     else
            if not all segment has been split out then
14:             GetZoneLock(zone_id)
                do segment splitting
16:             ReleaseZoneLock(zone_id)
            else
18:             # allocate new directory. expand_shift represent split factor
                # split multiple is equal to pow(2,expand_shift)
20:             dir_depth=current_dir→depth + expand_shift
                AllocateDir(thread_id, current_dir_id, expand_shift)
22:         end if
            return -1 # need retry
24:     end if
```

C. Lock in PMEH

1) Lock for Buckets: To handle skewed data access to the hash table, locks are used to ensure parallel access safety, except for search operations in the same zone. Each bucket is associated with a 32-bit version lock, with the highest bit used as the lock bit and the remaining bit used as the lock version. Before a record is written, the bucket is first locked. After the record is written, the bucket is unlocked, and the lock version is incremented.

The read operation for the hash table does not require a lock, but it needs to read the lock versions before and after the read operation for a bucket. These two versions are compared, and if they are not equal, the read operation is retried since there may have been parallel writes to the bucket. It is worth noting that the segment found by the search process may not be indexed by the latest directory since the read operation does not update entries of the new directory to reduce delay. If the old directory indexes the segment, it needs to check whether the newer directory has been updated to point to this segment after the search is complete. If the newer directory has been updated to point to this segment, the read operation needs to be repeated because the segment may have been split in the newer directory, and then the data may be missed or incorrect. Algorithm 5 shows the read process used in PMEH.

2) Lock for Zones: In addition, locks for zones are required to ensure data consistency during segment splitting and the updating of entries in the new directory. If segment splitting occurs while entries in the new directory are being updated in one zone, data loss may occur. This is because the latest directory entries may point to the old segment if entries have been updated before segment splitting. This can cause the new segment, which is split out by the old segment, not to be indexed by the new directory. Therefore, zone locks are necessary to ensure that segment splitting and directory entries update cannot occur simultaneously. It also avoids data loss between parallel segment splitting operations. If two threads perform segment splitting for the same segment simultaneously, the new segment produced by the one thread may overlap with the new segment produced by the other thread, leading to missing data. The insert algorithm with lock is shown in Algorithm 6.

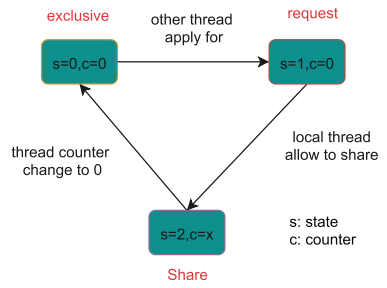

Fig. 7. State switch.

D. Opportunistic Lock-Free Parallel Operation in PMEH

To achieve good performance even with skewed data distribution, PMEH implements opportunistic lock-free parallel operation. It only needs to impose write locks for a small amount of write operation. The main idea is that PMEH first assigns every thread to handle its work and then assigns idle threads to assist busy threads. In other words. PMEH operates in a hybrid mode of lock-free and locking. To efficiently switch between lock-free and locking operations, PMEH introduces two variables for every zone: state and thread counter. The state variable represents the state of the zone, and the thread counter counts the number of threads that enter one zone. There are three states for one zone, exclusive state, request state, and shared state, as shown in Fig. 7. The initial state is exclusive, and each zone is only accessed by one thread. Thus, access to the hash table in the exclusive state is lock-free (Algorithm 7 lines 13–20). If some threads have completed their work and intend to help other threads, they

Algorithm 7 Insert With Opportunistic Lock-Free

```
    def PMEH::Insert(key, value, thread_id, is_local):
 2:     hash_key=Hash(key)
        zone_id=GetZoneId(hash_key)
 4:     finger=GetFinger(hash_key)
        RETRY:
 6:     current_dir_id=latest_dir_id
        current_dir=dir[current_dir_id]
 8:     segment_id=GetSegmentId(hash_key,current_dir->depth)
        bucket_id=GetBucketId(hash_key)
10:     segment=current_dir→_[segment_id]
        RETRY1:
12:     state=zone_state[zone_id] # get zone state
        if state==exclusive then
14:         result=InsertNoLock(key, value, segment, zone_id, current_dir_id,
            segment_id, bucket_id, finger)
            if result=-1 then
16:             goto RETRY
            else
18:             return result
            end if
20:     end if
        if state==request then
22:         if is_local then
                state=share
24:         else
                wait until state==share
26:         end if
            goto RETRY1
28:     end if
        if state==share then
30:         CAS increase zone[zone_id].thread_counter
            result=InsertWithLock(key, value, segment, zone_id, dir_id,
            segment_id, bucket_id, finger)
32:         CAS decrease zone[zone_id].thread_counter
            if zone[zone_id].thread_counter==0 then
34:             state=exclusive
            end if
36:         if result=-1 then
                goto RETRY
38:         else
                return result
40:         end if
        end if
```

first request to access the zone, which changes the state of the zone to the request state. If the local thread finds that the zone state has changed to the request state, it changes the state of the zone to the shared state (Algorithm 7 lines 21–28). After the state of the zone is changed to the shared state, subsequent access to the zone needs to be locked, and the lock operation is described in Section III-C. Additionally, the counter of threads for the zone needs to be incremented or decremented after each thread, including the local thread, enters or exits this zone in the shared state. To maintain the exclusive state as much as possible to improve parallelism, each thread checks whether the thread counter is equal to zero after exiting the accessed zone. If the counter of threads for the zone changes to zero, the state of the zone changes to the exclusive state (Algorithm 7 lines 29–40). PMEH achieves strong scalability and excellent performance by combining lock-free and locking operations, as demonstrated in Sections IV-C and IV-D.

E. Multiple Split

Expanding a hash table is necessary as data is continuously inserted into it. The delay is a significant consideration during the expansion process for the static hashing schemes. This is because rehashing the entire table is time-consuming and often blocks other access operations. In contrast, dynamic hashing tables do not have this issue since they do not need

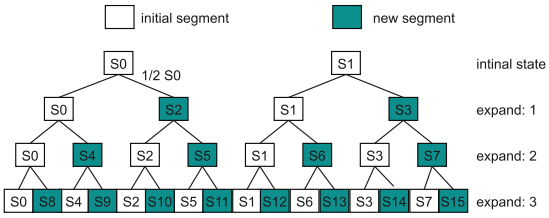

Fig. 8. Hash table expanded in 2-split.

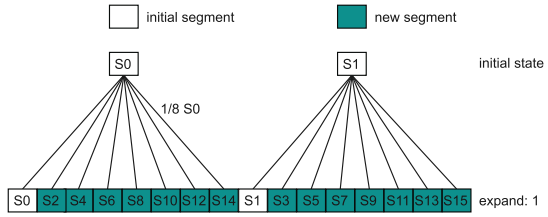

Fig. 9. Hash table expansion in 8-split without gradual splitting.

to rehash the entire hash table. During the expansion of a dynamic hashing table, the directory expansion is separated from the segment splitting operation. Only the directory expansion needs to be finished immediately, while the segment splitting operation is amortized into subsequent insertion operations. The directory size is minimal compared to the entire table and, hence, the overhead associated with directory expansion is also small. Apart from the delay, reducing the number of extra data movements during hash table expansion is also a problem worth considering. However, almost all previous hash tables have not taken this issue into account.

To reduce the number of extra data movements during hash table expansion, PMEH adopts multisplit instead of the traditional 2-split method. As depicted in Fig. 8, the original hash table comprises only two segments: 1) segment 0 (S0) and 2) segment 1 (S1), while the final hash table contains 16 segments to accommodate data. If PMEH uses the traditional 2-split method to expand the hash table, S0 and S1 would need to be split three times, and the new segments produced by the splitting operations would need to be split eight times. Hence, PMEH would have to perform 14 segment splitting operations in total to expand the original hash table to the final hash table. With the traditional 2-split method, approximately 1/2 of data from the old segment needs to be moved to the new segment for each segment splitting operation. This means that the traditional method would require moving approximately $1/2 * 14 = 7$ segments of data, which accounts for $7/16 \approx 0.44$ of the total data. Conversely, as illustrated in Fig. 9, if PMEH uses the 8-split method, both S0 and S1 would need to be split seven times, and the new segments produced by splitting would not require any further splitting. Although the total number of segment splitting is still 14, only around 1/8 of the data in the old segment needs to be moved to the new segment when using the 8-split method. In conclusion, the extra data movements required for the 8-split method are $1/8 * 14 = 7/4$ segments, accounting for approximately $7/64 \approx 0.11$ of the total data. Compared to the traditional 2-split method, the 8-split method reduces the

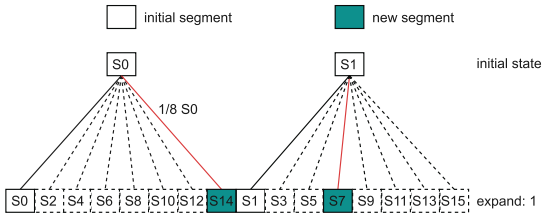

Fig. 10. Hash table expansion in 8-split with gradual splitting. Red solid lines indicate the newly split-out segments to accommodate the inserted value, while the black dashed lines indicate the nodes that will be split out once the current segment becomes full again.

number of extra data movements by about 33% of the entire data.

To maintain high space utilization and prevent a sharp increase in response latency caused by segment splitting, PMEH uses a gradual splitting method, as illustrated in Fig. 10. Instead of splitting a segment into multiple new segments, only one new segment is split out at a time for each segment splitting operation. To avoid repeatedly splitting out for the same new segment, a splitting bitmap is used to record which new segments have been split out. The bitmap preserves one bit for each new segment. Once a new segment has been indexed by the directory, the corresponding bit in the splitting bitmap of the origin segment is set. After the whole segments have been split out, the splitting bitmap is reset, and the local depth increases.

For instance, as depicted in Fig. 9, if seven new segments are split out from the old segment in a single splitting operation, the space utilization will decrease to 1/8 of the original space utilization. This is because the space increases by seven times compared with the previous one, but the number of records remains the same at this point. Conversely, the gradual splitting method, as shown in Fig. 10, expands the space only twice the original space. Therefore, the space utilization decreases to only 1/2 of the original space utilization, which is also demonstrated in Section IV-E. In addition, the latency incurred by the gradual splitting method is lower than complete splitting, as it has a lower overhead for memory allocation and data flushing.

F. Instant Recovery

Unlike DRAM, data structures designed for PM must ensure crash consistency. PMEH uses PMDK [48] for data persistence and memory allocation. Hence, PMEH only needs to consider the crash recovery for its own program since PMDK has guaranteed its data consistency. For PMEH, there are five types of crash recovery.

1) The first case is when a crash occurs during the insertion, and there is no segment splitting and directory expansion operation. In this scenario, the data consistency is not affected by a partial update for the record. This is because there is a bitmap that preserves one bit for one slot in a bucket to record whether one record is inserted completely or not.
2) The second case is when a crash occurs during segment splitting, and there is no directory expansion. In this situation, PMEH needs to record the splitting metadata of the segment, which includes (*directory_id*, *old_entry_id*, *new_entry_id*, *flag*) to ensure crash consistency. The directory_id represents the directory in which the old segment is. The old_entry_id indicates the index of the old segment in the directory. The new_entry_id is the index of the new segment in the directory. The flag is used to determine whether the new segment has been successfully inserted into the hash table or not. During recovery, each thread first checks the flag to determine whether the segment splitting has been completed or not. If the new segment has not been inserted successfully, the thread redoes the segment splitting operations according to the splitting metadata. Of course, the segment splitting operation can be undone.
3) The third case is a crash that occurs during directory expansion, with no ongoing segment splitting. To ensure correct recovery of the hash table, each thread has a directory pointer to record the latest allocated directory. When recovering, each thread checks whether its directory pointer points to a new or empty directory. If it is empty, it is not processed. Otherwise, PMEH traverses all threads' directory pointers to check whether a larger directory than the latest directory exists. If a larger directory is found, it is added to the directory array using CAS primitive. The update of entries in the new directory is amortized to subsequent insertion operations to speed up the recovery process.
4) The fourth case is when a crash occurs while segment splitting and directory expansion occur. First, PMEH recovers the segment-splitting operations according to the second crash type. Then, the directory expansion operations are recovered according to the third crash type. Finally, PMEH can guarantee that the new segment produced by segment splitting is indexed by the new directory allocated by directory expansion.
5) The fifth case is that locks are set when a crash occurs. The recovery process for locks simply involves resetting them. The recovery of locks in segments occurs when the segment is first accessed, and the one-bit flag is used to indicate whether the lock is recovered or not. The zone locks are recovered during the recovery process, as only a small number of them exist.

During the recovery process of PMEH, only the segment splitting metadata and thread pointer needs to be checked, rather than the entire table. Moreover, the recovery process is executed by multiple threads in parallel, and the update of entries is amortized to subsequent insertion operations. Therefore, PMEH is able to achieve instant recovery.

IV. Performance Evaluation

A. Experimental Setup

Our evaluations are conducted on a server equipped with the Intel Optane DCPMM in APP DIRECT Mode. The server runs the Linux operating system with kernel version 5.4.0 and ext4-DAX file system. The capacities of the Optane PM (DCPMM)

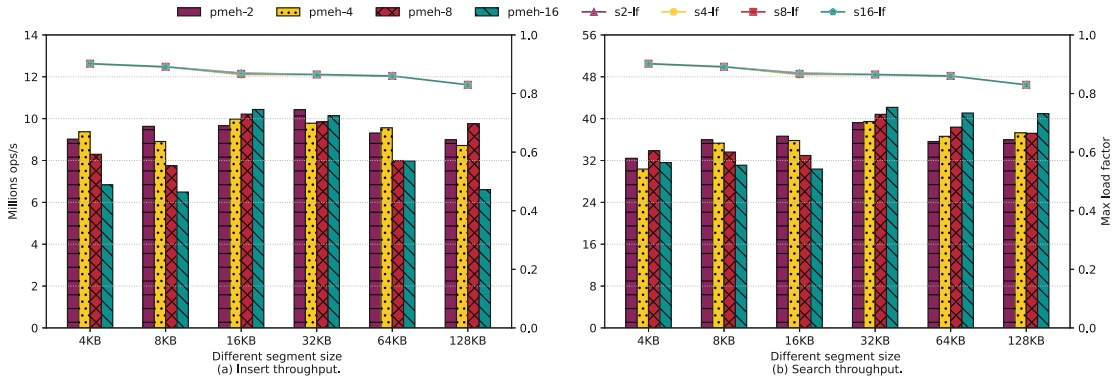

Fig. 11. Throughput of PMEH under different split multiples and different segment sizes (s*n*: *n*-split, s*n*-lf: load factor under *n*-split).

and DRAM are 512 (4 x 128) and 128 GB (4 x 32), respectively. The server was powered by two Intel Xeon Gold 6240 CPU @ 2.60 GHz, with 18 cores per CPU. Each core has a 32-kB L1d cache and a 32-kB L1i cache, while the L2 cache is 1 MB per core, and the shared L3 cache is 25 MB.

PMEH is compared with state-of-the-art hashing schemes, including Dash [26], CCEH [25], CLevel hashing [27], and Level hashing [24]. All these schemes use the synchronization primitives and memory management provided by PMDK [48] to guarantee crash consistency. Besides, an epoch-based reclamation mechanism is used to reduce overhead [26]. Dash and CCEH use the same parameters as those in their original paper to ensure a fair comparison. Dash adopts 16-kB segments and 256-byte (four cachelines) buckets, with each segment having two stash buckets for handling hashing collision. CCEH uses 64-byte buckets (one cacheline) and 16-kB segments, with a linear probing distance of four buckets. CLevel hashing is modified to store records directly rather than storing pointers to records in the hash table, and Level hashing is modified to adopt bucket locks, like Dash and CCEH, instead of slot locks. Both CLevel hashing and Level hashing use 128-byte buckets (two cachelines). Besides, the widely used YCSB [49] is adapted to generate the workloads, with both key and value being a fixed size of 8 bytes. The size of the initial directory for the hashing tables is set to 64, and the initial size for the hashing tables is 1 MB.

B. Sensitivity Analysis of PMEH Design

PMEH has two sensitivity parameters: split multiple and segment size. To determine the optimal values for these parameters, we evaluated the impact of different split multiples ranging from 2 to 16 and different segment sizes ranging from 4 to 128 kB. In PMEH, the bucket size is set to 256B (four cachelines) to match the PM access granularity, and each segment has two stash buckets. For the sensitivity test, 200 million records are first inserted into the hash table to evaluate the insertion performance and calculate the load factor. Then, 200 million records are read from the hash table to measure the search performance.

The results of the sensitivity are presented in Fig. 11. Fig. 11(a) and (b) illustrate the insertion throughput with load factor and the search throughput with load factor, respectively. Under the same split multiple, both insertion and search throughput exhibit two trends as the segment size increases gradually from 4 to 128 kB: 1) increasing first and then decreasing; and 2) decreasing first and then increasing. The reasons behind these trends are as follows. For insertion operation, frequent segment splitting occurs, leading to higher overall overhead if the size of segments is too small, although the delay for segment splitting is low. Conversely, if the size of segments is too large, the delay for segment splitting is long, even though segments are not split frequently. For search operations, varying segment sizes lead to a different distribution of records in the segment, resulting in changes in probing distances for records. Additionally, the insertion and search throughput vary for different splitting multiples under the same segment size. This is because the splitting multiple can affect the frequency of segment splitting, the number of data movements, and the distribution of records. Moreover, It is also observed that the load factor decreases as the segment size increases. This is because a segment is split once one bucket is full. The larger the segment, the fewer buckets become full when the segment is splitting.

The results from the sensitivity test indicate that the largest insertion throughput is obtained when the segment size is 16 or 32 kB, as shown in Fig. 11. Meanwhile, the largest search throughput is achieved when the segment size is 32 kB. It is worth noting that the load factors for the 32 and 16-kB segments are the same. Besides, a larger split multiple can reduce extra write (Srction IV-F). Hence, PMEH adopts the 32-kB segment and a split multiple of 16 as the optimal configuration. All subsequent experiments use this configuration.

C. Scalability

To test scalability under uniform workloads, the YCSB benchmark is used to generate five uniform workloads: 100% insert, 100% positive search, 100% negative search, 50% insert and 50% search, and 100% delete. For insert-only operations and mixed operations of 50% insertion and 50% search, ten million records are first preloaded into the hash table, followed by 190 million insertion or search operations. For search or deletion operations, 200 million records are preloaded into the hash table, and then 200 million search or delete operations are executed.

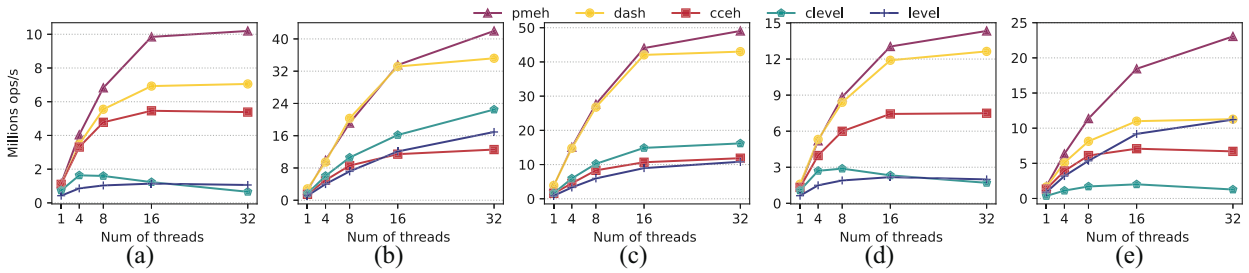

Fig. 12. Scalability(uniform) for different hashing schemes on PM. (a) 100% insert, (b) 100% positive search, (c) 100% negative search, (d) 50% insert and 50% search, and (e) 100% delete.

The experimental results of 100% insertion operations are presented in Fig. 12(a). The scalability of PMEH, Dash, CCEH and Level hashing is almost linear until the number of threads increases to 8. All hashing schemes achieve the maximum insert throughput when the number of threads is equal to or greater than 16. However, CLevel hashing, which adopts a lock-free mechanism, shows weaker scalability for insertion. This is because CLevel hashing only uses a single thread to perform the expanding process, causing extra multiple checks during expansion. In addition, CLevel and Level hashing need to check four buckets to find whether a key exists for a unique insertion, resulting in inferior insert throughput. On the Contrary, Dash shows good scalability since it uses fingerprints to reduce the number of probes. PMEH outperforms Dash as the number of threads increases due to eliminating most write locks and reducing data movement. Under 32 threads, PMEH outperforms Dash, CCEH, CLevel hashing, and Level hashing by 1.44×, 1.89×, 15.4×, and 9.6× for insert, respectively.

The experimental results of 100% positive search operations are shown in Fig. 12(b). All hashing schemes exhibit nearly linear scalability when the number of threads is less than 16, although there are performance gaps among them. PMEH shows a similar search performance to Dash since it also adopts a lock-free way to search records. However, PMEH surpasses Dash under 32 threads, due to its use of 32-kB segments instead of 16-kB segments, which leads to improved spatial locality. Additionally, PMEH does not amortize entries update to read operations. Under 32 threads, PMEH outperforms Dash, CCEH, CLevel hashing, and Level hashing by 1.19×, 3.34×, 1.86×, and 2.47×, respectively.

Fig. 12(c) presents the experimental results of 100% negative search operations. Compared with the test of 100% positive search, all hashing schemes exhibit similar scalability, except that PMEH and Dash show higher performance, while other hashing schemes exhibit lower performance. PMEH and Dash perform better than the other hashing schemes because they introduce one fingerprint for every record, thereby reducing the number of probes. Furthermore, in negative search, PMEH and Dash almost do not need to read keys for comparing, except for the fingerprints. Hence, they show better performance than their counterparts in positive search. Under 32 threads, PMEH outperforms Dash/CCEH/CLevel hashing/Level hashing by 1.14×/4.13×/3.03×/4.53× for negative search.

The experimental results for 50% insertion and 50% search operations are shown in Fig. 12(d). With the exception of CLevel, all hashing schemes demonstrate near-linear scalability at smaller or equal to 16 threads. This is because CLevel only adopts a single thread for resizing, leading to a performance bottleneck. The performance of PMEH is better than Dash's since PMEH has better insertion and read performance than Dash. For the mixed workload, PMEH outperforms Dash/CCEH/CLevel hashing/Level hashing by 1.13×/1.91×/8.41×/7.21× under 32 threads.

The experimental results of 100% deletion operations are presented in Fig. 12(e). When the number of threads is smaller or equal to 16, PMEH and Level hashing show near-linear scalability. This is because PMEH uses opportunistic lock-free for write. Level hashing has a similar deletion performance to Dash under 32 threads, as it has been modified to implement bucket locks like Dash. On the other hand, CCEH shows worse deletion performance than PMEH, Dash, and Level hashing because it adopts segment locks. CLevel hashing exhibits the worst deletion performance because it only adopts a single thread for background expansion and requires checking multiple levels to search records. In terms of deletion performance, PMEH outperforms Dash, CCEH, CLevel hashing, and Level hashing by 2.04×, 3.44×, 18.07×, and 2.05×, respectively, under 32 threads.

D. Skew Distribution

To comprehensively evaluate the performance of PMEH, five kinds of skewed (Zipfian) workloads are generated by the YCSB benchmark: 100% insert, 100% positive search, 100% negative search, 50% insert, and 50% search, and 100% delete. The experimental results for the skew workload are presented in Fig. 13. The insert throughput in the skewed workload, as shown in Fig. 13(a), is similar to that in the uniform workload. This is because PMEH uses efficient opportunistic lock-free parallel operations to deal with skew records. The positive search trends are similar, and the performance is better compared to the counterparts in the uniform workload, as shown in Fig. 13(b). The reason is that skewed data degrade the cache missing rate, and data can be accessed directly from the cache. However, the negative search performance does not improve, as shown in Fig. 13(c), since the cache does not cache any records useful for subsequent access. Fig. 13(d) shows similar trends and slightly higher performance for 50% insert and 50% search compared to the counterpart in uniform workload. This is because the insert performance is inferior to the search operation and, hence, the performance of 50% insert and 50%

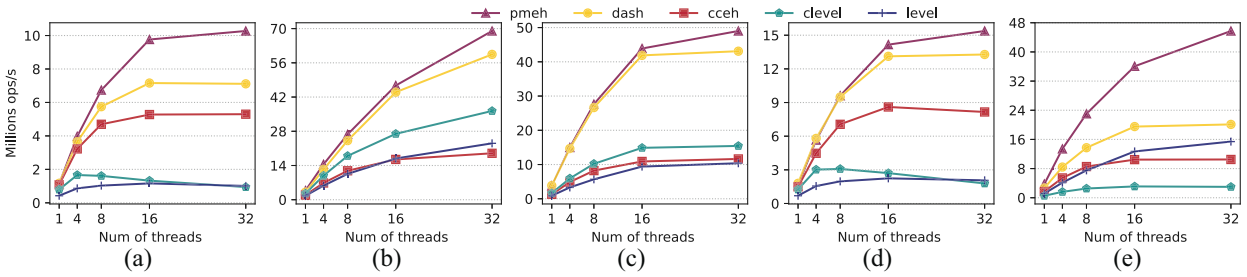

Fig. 13. Scalability(skew) for different hashing schemes on PM. (a) 100% insert, (b) 100% positive search, (c) 100% negative search, (d) 50% insert and 50% search, and (e) 100% delete.

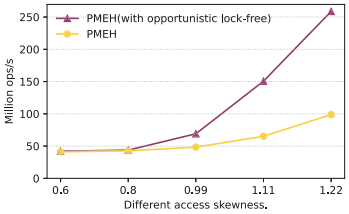

Fig. 14. Throughput of different access skewness under 32 threads.

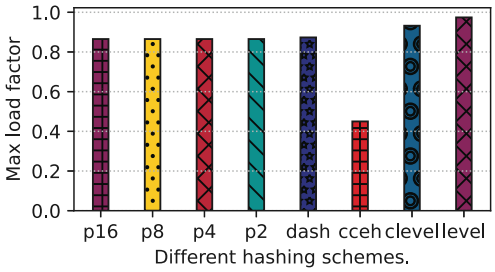

Fig. 15. Load factor for different hashing schemes (P16: PMEH under 16-split, et al.)

search depends on the insert performance. The experimental results of 100% deletion are presented in Fig. 13(e), which has similar trends and higher performance than its counterpart in the uniform workload. Moreover, the deletion performance for PMEH is similar to negative search since many duplicate records are presented in the skew workload. The subsequent delete operation is similar to the negative search when one record is deleted. On the other hand, Dash improves less than PMEH due to the expensive lock overhead for deletion.

Moreover, we conducted an experiment to compare PMEH with and without opportunistic lock-free parallel operation for search operations. This is because while the hash function can distribute inserted key–value pairs into a nearly uniform distribution for skewed data distribution, they cannot distribute key–value pairs evenly for search operations as these operations target specific positions. First, We generated different Zipfian distributions with varying θ parameter ranging from 0.6 to 1.22 [50]. A larger value of the parameter θ indicates a more heavily skewed data distribution. Then, we tested the throughput of PMEH with and without opportunistic lock-free parallel schemes. The experimental results are presented in Fig. 14. When the θ value is 0.6 and 0.8, which represent evenly distributed datasets, PMEH with and without opportunistic lock-free parallel schemes exhibit similar throughput. However, as θ increases from 0.8, PMEH with opportunistic lock-free parallel schemes shows excellent scalability and performance, while PMEH without it shows inferior scalability and performance. Therefore, our opportunistic lock-free schemes achieve excellent scalability and performance regardless of the data distribution.

E. Load Factor

200 million records are inserted into empty hashing tables using a single thread to evaluate the maximum load factor. Specifically, PMEH is evaluated by varying the split multiple from 2 to 16. The load factor is measured for every insertion operation. As shown in Fig. 15, PMEH maintains a consistent maximum load factor regardless of the split multiple used. This is because PMEH uses gradual splitting to split segments. In addition, PMEH and Dash exhibit similar load factors due to their similar segment organization structure. CCEH has the worst load factor since it only uses linear probing to deal with collision records and the probing distance is limited to four buckets to obtain good search performance. Level hashing and CLevel hashing have a similar organizational structure, resulting in comparable load factors. However, CLevel hashing is slightly inferior to Level hashing because it does not move items and has a slightly larger slot number than Level hashing. Furthermore, Level hashing and CLevel hashing have slightly better load factors than PMEH and Dash due to their shared-based two-level structure.

F. Extra Data Movements

To evaluate the number of extra data movements for various hashing schemes, a total of 200 million records are inserted into empty hash tables with a single thread, including PMEH, Dash, CCEH, CLevel hashing, and Level hashing. PMEH is evaluated while varying split multiple from 2 to 16. The experimental results are shown in Fig. 16. PMEH has a similar number of extra data movements as Dash under 2-split, which can be attributed to the fact that Dash has a similar segment structure as PMEH and uses 2-split to expand the hash table. The number of extra data movements in PMEH under 2-split is slightly less than in Dash since PMEH uses 32-kB segments to perform segment splitting, resulting in fewer segment splits. The number of extra data movements in PMEH increases as the split multiple decreases. This is because a smaller split multiple leads to more hash table expansion, which increases the number of extra data movements. Compared with Dash, PMEH's 16-split, 8-split, 4-split, and 2-split reduce the number of extra data movements by 52%, 31%, 9%, and 4%, respectively. CCEH exhibits the

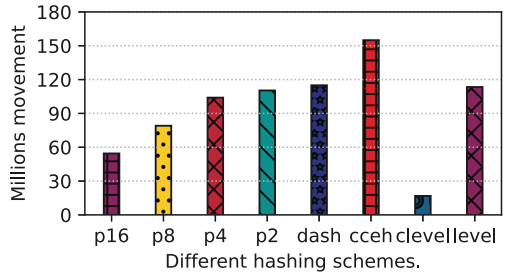

Fig. 16. Extra data movements for different hashing schemes (P16: PMEH under 16-split, et al.)

TABLE II
RECOVERY TIME VERSUS DATA SIZE

	Number of indexed records (million)					
records (million)	1	10	100	200	300	400
time (ms)	1.8	1.8	1.9	1.6	1.5	1.9

most significant amount of extra data movement since it has the worst load factor, which results in frequent segment splits. On the other hand, Clevel hashing shows the least amount of extra data movement, since it only adopts a single thread to expand the hash table in the background. Therefore, many data movements are still in process, but Clevel hashing does not count them. Besides, Level hashing shows the same amount of extra data movement as dash and PMEH under 2-split, which can be attributed to the fact that the expansion for level hashing is double the size of hash table [26].

G. Recovery

For the recovery test, failures are deliberately introduced into the code. The recovery time of different data sizes, ranging from 1 million to 400 million records, is measured. The results are shown in Table II. It can be observed that the recovery time of PMEH remains consistently low, at around 2 ms, regardless of the amount of data. This is because PMEH adopts multiple threads to process the recovery simultaneously, and each thread only processes a small amount of metadata related to crashes. Besides, PMEH amortizes the heavy update of new directory entries to subsequent insertion operations, enabling it to achieve instant recovery.

V. CONCLUSION

This article introduces PMEH, a novel variant of extendible hashing based on PM that offers excellent scalability and write-optimized functionality. PMEH improves the scalability and write performance of extendible hashing by incorporating fine-grained and lock-free structures that support a significant degree of parallelism and reduce extra write. PMEH divides the directory into multiple zones, each associated with a single thread, thereby eliminating the need to share segments among multiple threads. The use of multiple directories allows for lock-free updates during directory expansion, avoiding blocking other threads. To ensure good performance even in the presence of skewed data distribution, PMEH adopts an opportunistic lock-free scheme that switches dynamically between lock-free and locking operation. Furthermore, PMEH employs a multisplit with gradual splitting mechanism that minimizes data movements during expansion, leading to fewer writes to PM. Finally, PMEH offers instant recovery by recording minimal metadata. Experimental results demonstrate that, under the YCSB workloads, PMEH outperforms the existing state-of-the-art hashing mechanisms, including Dash, CCEH, CLevel hashing, and Level hashing in terms of scalability, performance, and PM writes.

REFERENCES

[1] S. Raoux et al., "Phase-change random access memory: A scalable technology," *IBM J. Res. Develop.*, vol. 52, nos. 4–5, pp. 465–479, Jul. 2008.
[2] "Intel and micron produce breakthrough memory technology." Intel and Micron. 2015. [Online]. Available: http://newsroom.intel.com/news-releases
[3] T. Kawahara, "Scalable spin-transfer torque RAM technology for normally-off computing," *IEEE Design Test Comput.*, vol. 28, no. 1, pp. 52–63, Jan./Feb. 2011.
[4] "Intel® Optane™ DC persistent memory." Intel. 2019. [Online]. Available: https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html
[5] J. Izraelevitz et al., "Basic performance measurements of the Intel Optane DC persistent memory module," 2019, *arXiv:1903.05714*.
[6] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "SplitFS: Reducing software overhead in file systems for persistent memory," in *Proc. 27th ACM Symp. Oper. Syst. Principles*, 2019, pp. 494–508.
[7] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST)*, 2020, pp. 169–182.
[8] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: Converting concurrent dram indexes to persistent-memory indexes," in *Proc. 27th ACM Symp. Oper. Syst. Principles*, 2019, pp. 462–477.
[9] Q. Wang, Y. Lu, J. Li, and J. Shu, "Nap: A black-box approach to NUMA-aware persistent memory indexes," in *Proc. 15th USENIX Symp. Oper. Syst. Des. Implement. (OSDI)*, 2021, pp. 93–111.
[10] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proc. VLDB Endowment*, vol. 13, no. 4, pp. 574–587, 2019.
[11] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Proc. Int. Conf. Manag. Data*, 2016, pp. 371–386.
[12] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-tree: Reducing consistency cost for NVM-based single level systems," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 167–181.
[13] S. Chen and Q. Jin, "Persistent B$^+$-trees in non-volatile main memory," *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.
[14] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proc. FAST*, vol. 11, 2011, pp. 61–75.
[15] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write optimal radix tree for persistent memory storage systems," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 257–270.
[16] S. Ma et al., "ROART: Range-query optimized persistent ART," in *Proc. 19th USENIX Conf. File Storage Technol. (FAST)*, 2021, pp. 1–16.
[17] J. Liu, S. Chen, and L. Wang, "LB+Trees: Optimizing persistent index performance on 3DXPoint memory," *Proc. VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, 2020.
[18] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen, "DPTree: Differential indexing for persistent memory," *Proc. VLDB Endowment*, vol. 13, no. 4, pp. 421–434, 2019.
[19] L. Benson, H. Makait, and T. Rabl, "Viper: An efficient hybrid PMem-DRAM key-value store," *Proc. VLDB Endowment*, vol. 14, no. 9, pp. 1544–1556, 2021.
[20] Y. Luo, P. Jin, Q. Zhang, and B. Cheng, "TLBtree: A read/write-optimized tree index for non-volatile memory," in *Proc. IEEE 37th Int. Conf. Data Eng. (ICDE)*, 2021, pp. 1889–1894.
[21] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B+-tree," in *Proc. 16th USENIX Conf. File Storage Technol. (FAST)*, 2018, pp. 187–200.

[22] H. Cha, M. Nam, K. Jin, J. Seo, and B. Nam, "B^3-tree: Byte-addressable binary B-tree for persistent memory," *ACM Trans. Storage*, vol. 16, no. 3, pp. 1–27, 2020.
[23] P. Zuo and Y. Hua, "A write-friendly and cache-optimized hashing scheme for non-volatile memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 5, pp. 985–998, May 2018.
[24] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implement. (OSDI)*, 2018, pp. 461–476.
[25] M. Nam, H. Cha, Y.-R. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST)*, 2019, pp. 31–44.
[26] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," 2020, *arXiv:2003.07302*.
[27] Z. Chen, Y. Huang, B. Ding, and P. Zuo, "Lock-free concurrent level hashing for persistent memory," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2020, pp. 799–812.
[28] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 1984, pp. 1–8.
[29] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Trans. Knowl. Data Eng.*, vol. 4, no. 6, pp. 509–516, Dec. 1992.
[30] "PostgreSQL." PostgreSQL Global Development Group. 2019. [Online]. Available: https://www.postgresql.org/
[31] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: Dependably fast multi-core in-memory transactions," in *Proc. ACM Int. Conf. Manag. Data*, 2017, pp. 21–35.
[32] "MySQL." Oracle. 2019. [Online]. Available: https://www.mysql.com/
[33] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers accelerating index traversals for in-memory databases," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2013, pp. 468–479.
[34] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "FASTER: A concurrent key-value store with in-place updates," in *Proc. Int. Conf. Manag. Data*, 2018, pp. 275–290.
[35] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing," in *Proc. 10th USENIX Symp. Netw. Syst. Des. Implement. (NSDI)*, 2013, pp. 371–384.
[36] S. Ghemawat and J. Dean. "LevelDB." 2019. [Online]. Available: https://github.com/google/leveldb
[37] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-R. Choi, "SLM-DB: Single-level key-value store with persistent memory," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST)*, 2019, pp. 191–205.
[38] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, pp. 1–28, 2017.
[39] "Redis." Redis Labs. 2016. [Online]. Available: https://redis.io
[40] S. Xu et al., "BlueCache: A scalable distributed flash-based key-value store," Ph.D. dissertation, Dept. Electr. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2016.
[41] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Oper. Syst. Des. Implement.*, 2006, pp. 307–320.
[42] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implement. (NSDI)*, 2014, pp. 429–444.
[43] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing—A fast access method for dynamic files," *ACM Trans. Database Syst.*, vol. 4, no. 3, pp. 315–344, 1979.
[44] H. Mendelson, "Analysis of extendible hashing," *IEEE Trans. Softw. Eng.*, vol. SE-8, no. 6, pp. 611–619, Nov. 1982.
[45] H. Korth, A. Silberschatz, and S. Sudarshan, *Database Systems Concepts*. New York, NY, USA: McGraw-Hill, 2005.
[46] *Intel 64 and IA-32 Architectures Software Developer's Manual*, vol. 3A, Intel, Santa Clara, CA, USA, 2014.
[47] "Intel® architecture instruction set extensions programming reference." Intel. 2018. [Online]. Available: https://software.intel.com/en-us/isaextensions
[48] "Persistent memory development kit." Intel. 2018. [Online]. Available: https://pmem.io/pmdk/libpmem/
[49] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[50] J. Chen et al., "HotRing: A hotspot-aware in-memory key-value store," in *Proc. FAST*, 2020, pp. 239–252.

Jing Hu received the B.E. degree in computer science and technology from the China University of Geosciences, Wuhan, China, in 2017. He is currently pursuing the Ph.D. degree with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan.

His current research interests include nonvolatile memory technologies, key-value stores, and parallel and distributed systems.

Jianxi Chen (Member, IEEE) received the B.E. degree from Nanjing University, Nanjing, China, in 1999, and the M.S. and Ph.D. degrees in computer architecture from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2002 and 2006, respectively.

He is currently an Associate Professor with HUST. He has published over 20 papers in major journals and conferences. His research interests include computer architecture and massive storage systems.

Dr. Chen is a member of CCF.

Yifeng Zhu (Member, IEEE) received the B.S. degree from the Huazhong University of Science and Technology, Wuhan, China, in 1998, and the M.S. and Ph.D. degrees from the University of Nebraska, Lincoln, NE, USA, in 2002 and 2005, respectively.

He is an Associate Professor with the University of Maine, Orono, ME, USA. His research interests include parallel I/O storage systems and energy-aware memory systems.

Dr. Zhu received the Best Paper Award at IEEE CLUSTER 07. He served for the Program Committee of international conferences, including ICDCS and ICPP. He is a member of ACM and Francis Crowe Society.

Qing Yang received the B.S. and M.S. degrees in computer science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2018 and 2021, respectively.

His current research interests include data compression and deduplication, nonvolatile memory, and memory security.

Zhouxuan Peng received the B.E. degree in computer science and technology from the Huazhong Universality of Science and Technology, Wuhan, China, in 2018, where he is currently pursuing the Ph.D. degree in computer system architecture.

His current research interests include hybrid/heterogeneous and low-power-embedded memory systems.

Ya Yu received the B.E. degree in computer science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2009, and the M.S. degree from Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology in 2021.

His research interests include persistent memory file systems and databases.