Accelerating Persistent Hash Indexes via Reducing Negative Searches

Renzhi Xiao*, Hong Jiang‡, Dan Feng*†, Yuchong Hu†, Wei Tong*, Kang Liu†, Yucheng Zhang*, Xueliang Wei†, Zhengtao Li*

*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
†School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
‡Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, USA
Correspongding Author: dfeng@hust.edu.cn
{rzxiao,dfeng,yuchonghu,tongwei,kangliu,xueliang_wei,lizhengtao10}@hust.edu.cn,hong.jiang@uta.edu,zhangyc_hust@126.com

***Abstract*—Hashing is a widely used and efficient indexing mechanism for key-value storage. Persistent memory (PM) has attracted extensive attention in research due to its non-volatility and DRAM-like performance. Intel DCPMM, as a PM, can provide large capacity and low total cost of ownership, further promoting the research of PM-based hash index. However, based on real-world workloads, we found that negative searches of existing PM-based hash indexes significantly degrade system performance. A direct method to solve this problem is to use a PM-based Bloom filter to reduce negative searches, but at the cost of the decreased lifespan of PM due to extra PM writes. An alternative method is to use a DRAM-based Bloom filter, but it still faces increased multi-threaded insertion/deletion/positive-search scalability overhead as well as increased data consistency and recovery overhead.**

In this paper, we propose SmartHT, a small-size DRAM-based Bloom filter to accelerate hash table operations for PM while solving the aforementioned problems. SmartHT uses efficient merge write optimization with head insertion, lazy deletion, and shortened average chained length of head-bucket to provide high insertion/deletion/positive-search scalability, respectively. On the other hand, it utilizes a merged-flush mechanism based on an 8-byte failure-atomic write method to reduce flush instructions and extra PM writes to achieve low data consistency overhead. Experimental results on Intel Optane DCPMM show that, compared with the state-of-the-art persistent hash indexes, SmartHT improves multi-threaded negative queries under uniform and skewed distributions by 4.61x-13.86x and 2.76x-12.99x respectively, achieves high multi-threaded scalability and low data consistency overhead, at the modest cost of recovery time overhead.

***Index Terms*—persistent memory, PM-based hash index, negative research, multi-threaded scalability**

I. Introduction

Persistent memory (PM) provides a new and promising path of building large-scale and low-latency memory and storage systems. PM, such as phase-change memory [1] and Intel Optane DC Persistent Memory Module (DCPMM) [2], offers desirable properties such as large capacity, high performance, non-volatility, and byte addressability. PM is expected to mix with DRAM as a hybrid memory system, bridging the performance gap between HDD and DRAM and enriching the storage layer. The DCPMM released by Intel has a single-device capacity of up to 256GB, and the maximum single-machine capacity is 8TB (512GB/DIMM * 16DIMMs), making it capable of providing a real-time processing system with high throughput, low latency, and low total cost of ownership (TCO) for a wide range of applications.

Hash indexes are widely used in-memory indexing structures for key-value storage systems, such as Memcached and Redis, because of their constant-level fast lookup performance. If multiple keys are mapped to the same location (i.e., hash conflicts), they need to be rehashed or resized if the hash table is full. With the development of PM, numerous researchers have been focusing on designing efficient PM-based persistent hash indexes, such as Level hashing [3], CCEH [4], PCLHT [5], SOFT [6], Clevel [7], and Dash [8].

A positive search looks up an existing element in the hash table, whereas a negative search looks up a non-existing element in the hash table. However, the aforementioned PM-based hash indexes still suffer from poor negative search performance. Our study on negative search analysis for them in Section II-C suggests that the negative query throughput of persistent hash indexes is only 45.1%-79.8% of their positive query throughput. Negative queries in a PM-based hash index system result in a large number of PM accesses, which significantly degrade system performance, particularly when handling workloads that frequently query non-existing elements, making it necessary and important to improve negative query performance. Bloom filter can effectively address the negative-search problem. Directly reducing negative queries using PM-based Bloom filters has limitations, including reduced system performance due to DCPMM-based PMs' 2-3 times higher read latency than DRAM [2] and potential damage to PM durability from extra PM writes. Additionally, while using a DRAM-based Bloom filter to optimize a PM-based hash index can avoid the issues introduced by a PM-based Bloom filter, it faces two challenges: (1) increased scalability

This work was supported by NSFC (No. 61821003, U22A2027, 61832020, 62262042, 62202190), Key Research and Development Program of Hubei Province (No. 2021BAA189), Hubei Province Science and Technology Research Project (No. 2023BAA018), Hubei Province Natural Science Foundation (No.2023AFB237), the Open Project Program of Wuhan National Laboratory for Optoelectronics (No. 2021WNLOKF012), and Key Laboratory of Information Storage System Ministry of Education of China.

2576-6996/23/$31.00 ©2023 IEEE
DOI 10.1109/ICCD58817.2023.00035

overhead for multi-threaded insertion/deletion/positive-search operations, and (2) data consistency and recovery overhead.

Motivated by the above analysis, we propose SmartHT, a small-sized DRAM-based Bloom filter to accelerate hash table operations for persistent memory by reducing negative searches. SmartHT adopts several techniques to tackle the two challenges faced by a DRAM-based Bloom filter. It merges small writes with head insertion to enhance multi-threaded insertion scalability, adopts a lazy deletion approach to improve multi-threaded deletion scalability, and shortens the average chained length of head-bucket to reduce the positive search path length and improve the positive search scalability. To guarantee low data consistency overhead, SmartHT minimizes costly PM flushes through one 64-byte merged-flush technique based on 8-byte failure-atomic writes. Experimental results on Intel Optane DCPMM show that SmartHT outperforms state-of-the-art persistent hash indexes in terms of the negative query, achieving 4.61x-13.86x and 2.76x-12.99x improvements under uniform and skewed workloads generated by PiBench [9], respectively. Furthermore, SmartHT achieves high multi-threaded scalability, and low data consistency overhead, at the modest cost of recovery time.

II. Background and Motivation

A. Persistent Hash Indexes

PM-based persistent hash indexes such as Level hashing [3], Clevel [7], CCEH [4], Dash [8], PCLHT [5] and SOFT [6], are cutting-edge data structures that overcome the limitations of traditional memory hash indexes, including capacity limitations and data loss. Level hashing [3] is a hash table structure specifically designed for persistent memory that achieves cost-effective resizing and low-overhead data consistency through log-free failure atomicity operations. Clevel [7] was developed as an enhancement to Level hashing by replacing the slot lock with lock-free concurrency through atomic compare-and-swap functions, and utilizing a background thread for asynchronous resizing that does not block reads. CCEH [4] is a PM-based extendible hashing which utilizes dynamic capacity expansion through segment splitting without expensive full-table resizing, but suffers a low load factor. Dash [8] addresses the issue of frequent segment splitting with balanced insert, replacement, and stashing techniques to improve load factor at the cost of more PM reads. PCLHT [5] uses a lock-free mode to enhance read concurrency and bucket locks for multi-threaded write concurrency correctness. SOFT [6] is a hash table optimized for hybrid DRAM and PM usage, utilizing volatile nodes (VNodes) in DRAM and persistent nodes in PM.

Unlike traditional memory hash indexes, which face constraints on size and risk data loss in cases of failure, PM-based persistent hash indexes offer faster read and write speeds approaching those of memory-based tables and support larger data sizes, up to several terabytes. Importantly, they ensure data persistence, even in the face of system failure, making them highly reliable for scenarios that require fast and consistent data storage, such as high-speed caching, real-time analysis, and high-frequency data access operations.

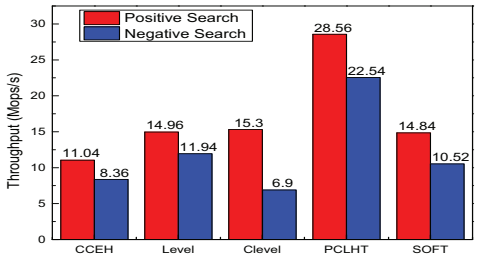

Fig. 1. 32-threaded positive vs negative search throughput for various persistent hashes with uniform distribution.

B. Bloom Filter

The Bloom filter is a popular and space-efficient probabilistic data structure known for its exceptional negative query performance [10], [11]. By distinctly identifying the non-membership of an element in a set, the Bloom filter eliminates the potential for false negatives. However, it can exhibit false positives, leading to a requirement for additional confirmation of an element's membership after a positive result. The false-positive rate in a Bloom filter can be diminished by increasing the number of bits and hash functions. Hence, achieving an acceptable false-positive rate while maintaining low space and time overheads is a crucial endeavor when designing a Bloom filter.

C. Negative Search Analysis

Negative searches in a key-value storage system indicate the non-existence of the sought-after key-value pair. Such negative queries are extensively employed in PM-based persistent hash indexes [3], [12]. Nevertheless, negative searches in a persistent hash index system result in a considerable number of PM accesses, causing substantial performance degradation, mainly when dealing with workloads that frequently query non-existent elements. As illustrated in Figure 1, experimental results on Intel Optane DCPMM using popular benchmark PiBench [9] demonstrate that the negative query throughput of existing persistent hash indexes ranges from 45.1% to 79.8% of their positive query throughput.

Since Bloom filters have been shown to significantly reduce negative queries, introducing them to persistent hash indexes has the potential to substantially optimize negative query performance. One straightforward approach is to use a PM-based Bloom filter, which, however, incurs additional PM writes, thereby decreasing the lifespan of PM. In addition, the access latency of DRAM is lower than that of PM. Previous work [2] shows that the sequential read and random read latency of DCPMM are 2 and 3 times that of DRAM, respectively, and the read and write bandwidth of DCPMM is 1/3 and 1/6 of that of DRAM respectively. Based on the above analysis, a DRAM-based Bloom filter is more conducive to improving performance compared to a PM-based one.

D. Challenges and Motivation

Although a DRAM-based Bloom filter offers more advantages than a PM-based Bloom filter, introducing it into persistent hash indexes still faces the following challenges.

Challenge 1. Scalability overhead in integrating DRAM-based Bloom filter into persistent hash indexes. Incorporating

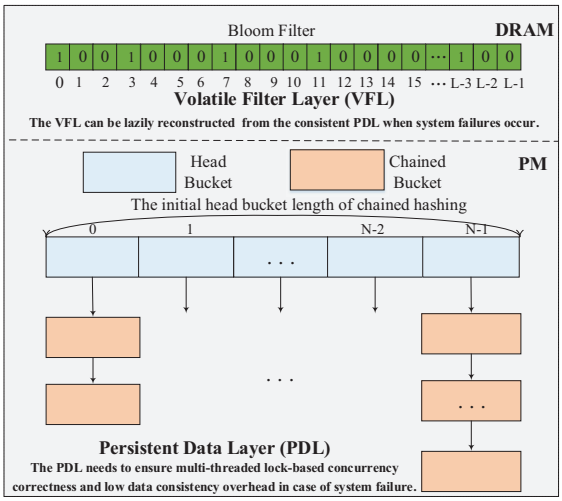

Fig. 2. The architecture of SmartHT.

a DRAM-based Bloom filter in a persistent hash index that uses Persistent Memory (PM) can result in scalability overhead due to increased search costs during positive queries. This can lead to reduced query efficiency, as finding Bloom filters during insertion, deletion, and metadata insertion requires additional steps. Moreover, in a multi-threaded environment, a DRAM-based Bloom filter in persistent hash indexes can negatively impact scalability and efficiency in inserts, positive queries, and deletions.

Challenge 2. Consistency and recovery overhead in integrating DRAM-based Bloom filter into persistent hash indexes. Incorporating a DRAM-based Bloom filter as an optimization strategy for persistent hash indexes can cause data consistency challenges in case of system failures, as the Bloom filter metadata in the DRAM may be lost. In such cases, restoring the Bloom filter metadata and repairing the filter creates an increased consistency and recovery overhead, which negatively impacts the data management process.

Since reducing negative search using a PM-based Bloom filter can decrease the lifespan of PM and increase costly PM reads, in this paper, we focus on using a DRAM-based Bloom filter while addressing the challenges it presents.

III. The Design of SmartHT

In this section, we present the design details of SmartHT, a holistic scheme using a small DRAM-based Bloom filter to accelerate hash table operations for PM by mitigating negative searches to the hash table in PM. The design goals of SmartHT are as follows:

- **Reducing Negative Searches.** SmartHT should efficiently reduce negative searches.
- **High Multi-threaded Scalability.** SmartHT should scalable to multiple threads.
- **Low Data Consistency Overhead.** SmartHT should achieve low data consistency overhead while providing accurate recovery after a system failure.

A. SmartHT Overview

SmartHT is a persistent hash index optimized by a DRAM-based Bloom filter that reduces negative searches and has high multi-threaded scalability and low data consistency overhead. SmartHT employs a DRAM-based Bloom filter to enhance negative search performance in Section III-B1. A DCPMM-aware design allows SmartHT to read only the DRAM-based Bloom filter during negative searches, which avoids costly PM accesses to the hash table located in PM. Under lock-based concurrency control, SmartHT uses efficient merge small writes with head insertion, lazy deletion, and a shortened average chained length of head-bucket to achieve high multi-threaded insertion/deletion/positive-search scalability, respectively. To minimize the data consistency overhead, SmartHT reduces the costly cacheline flush instructions and extra PM writes by employing a merged flush in one cacheline-sized bucket based on an 8-byte failure-atomic write.

Figure 2 shows an overview of the SmartHT architecture in hybrid DRAM-PM memory. Due to the significant performance differences between DRAM and DCPMM-based PM, SmartHT leverages the strengths of both memory types to minimize their respective disadvantages, details in Section III-B. SmartHT adopts a cacheline-sized bucket and traditional chained list to handle hash conflicts and obtain high space utilization. SmartHT is composed of a Volatile Filter Layer (VFL) and a Persistent Data Layer (PDL), where VFL is a volatile Bloom filter in DRAM used for fast negative searches, and PDL is a persistent chained hashing in PM used for persistence. After system failures, VFL does not need to guarantee data consistency because the consistent PDL can lazily reconstruct it.

B. Hybrid Memory Architecture

*1) **Volatile Fitler Layer (VFL):*** SmartHT's VFL is a DRAM-based Bloom filter primarily used to reduce negative searches for a persistent chained hash in PDL, which can significantly improve the negative search performance and decrease costly PM accesses (DCPMM-based PM read latency is 2-3 times that of DRAM [2]). We can calculate the minimum number of bits required for three hash functions according to the formula for Bloom Filters as follows: $n = -(m * ln(p))/(ln(2))^2$, where n represents the required number of bits, m represents the number of key-value entries stored, and p represents the false positive rate. We set a low false-positive rate of 0.0001 for the Bloom filter considering that a false positive can result in significant search overhead. According to the formula, a Bloom filter that stores 200 million key-value pairs requires at least 288 million bits, approximately 34.64MB. In practice, additional space can increase the accuracy rate.

*2) **Persistent Data Layer (PDL)):*** SmartHT's PDL utilizes a persistent chained hashing method in PM, as depicted in Figure 2. The hash table of SmartHT in the PDL uses chained lists to resolve hash conflicts. Therefore, each head bucket may have multiple chained buckets. SmartHT aims to decrease the average search path length by utilizing a shortened average

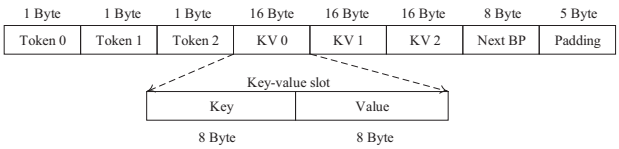

Fig. 3. The 64-byte bucket structure for head and chained buckets.

chained length of head buckets. Therefore, SmartHT sets the length N of the head bucket array for the chained hashing to 2^{24}. Each bucket size in PDL is 64 bytes and comprises three one-byte tokens, three 16-byte key-value slots, one 8-byte next bucket pointer (BP), and a five-byte padding, as shown in Figure 3.

SmartHT utilizes a shared-read and exclusive-write mutex for lock-based concurrency control to achieve multi-threaded correctness. To reduce the space and operation overhead of the locks, SmartHT divides chained hashing into many lock areas with a lock size, or lock stripe, of 256 head buckets, as depicted in Figure 4. Despite the limitations of a coarse-grained lock in terms of limiting the concurrency of key insertion within a single lock area, SmartHT leverages several optimizations to maintain high scalability for this critical operation, including merging small writes with head insertion and shortening the average chained length of the head bucket. To improve read scalability, SmartHT employs a shared-lock method, enabling it to achieve both insertion and read parallelism for different lock areas and read concurrency for the same lock area.

SmartHT prioritizes data consistency and adopts a two-stage insertion process. First, key-value entries are inserted into PM-based chained hashing in PDL, followed by metadata insertion into the DRAM-based Bloom filter in VFL. To minimize the overhead incurred by flush instructions, SmartHT uses a 64-byte bucket to combine small writes into a single cacheline flush, which employs an 8-byte failure-atomic write for merged flushing. As a result, SmartHT achieves low data consistency overhead, which is especially crucial for PM-based key-value store systems.

While SmartHT only guarantees the data consistency of PDL in PM, there is a chance of losing metadata in the Bloom filter of VFL located in DRAM during system failures. To address this issue, SmartHT recovers the metadata in the DRAM-based Bloom filter to establish a key's membership via SmartHT's consistent PDL after a system failure. SmartHT has the capability of lazy recovery, whereby the Bloom filter can be recovered without compromising any of the actual data stored in the PDL in case of system failure.

C. SmartHT Operations

1) ***Insert****:* The SmartHT solution effectively reduces write and consistency costs by combining value, key, and its corresponding token of the key-value item and then flushing an entire cache line back to PM. To minimize unnecessary search paths and costly DCPMM accesses, SmartHT uses head insertion instead of tail insertion for chained hashing in PDL, where a head bucket may have long insertion chains.

Algorithm 1: Insert

```
1  Calculate the hash value (hashVal) and specific head
     bucket number (HBN) according to the given key;
2  curr_bucket_ptr = &Buckets[HBN];
3  if Is_Existed_in_SmartHT(key) then
4  |   return false;
5  end
6  // Head insert the key-value pair into SmartHT if the
     pair does not exist
7  (curr_bucket_ptr, empty_slot_num) = Find_Empty_Slot
     (Head_Bucket or First_Chained_Bucket);
8  set unique_lock Locks[hashVal/locksize] is true;
9  if curr_bucket_ptr != NULL and
     Is_Valid(empty_slot_num) then
10 |   curr_bucket_ptr→slot[empty_slot_num].value=value;
11 |   curr_bucket_ptr→slot[empty_slot_num].key = key;
12 |   curr_bucket_ptr→token[empty_slot_num] = 1;
13 |   Use CLFLUSH to persist the bucket to PM;
14 |   return true;
15 end
16 curr_bucket_ptr = &Buckets[HBN];
17 tmpBucket_ptr = Allocate_Bucket_From_PM();
18 if tmpBucket_ptr == NULL then
19 |   print new temporary bucket failed and return false;
20 end
21 tmpBucket_prt→slot[0].value = value;
22 tmpBucket_ptr→slot[0].key=key;
23 tmpBucket_ptr→token[0]=1;
24 tmpBucket_ptr→next =curr_bucket_ptr→next;
25 Use CLFLUSH to persist the bucket pointed by
     tmpBucket_ptr to PM;
26 Use memory fence (MFENCE) to Control PM
     Persistence Order;
27 curr_bucket_ptr→next = tmpBucket_ptr;
28 Use CLFLUSH to persist the head bucket pointed by
     curr_bucket_ptr to PM;
29 Use MFENCE to Control PM Persistence Order;
30 Insert the key into Bloom filter in VFL;
31 return true;
```

SmartHT employs an exclusive mutex (i.e., unique_lock) for insert operations. Algorithm 1 outlines the pseudo-code of the insert operation for SmartHT.

2) ***Search****:* SmartHT employs a shared-read lock approach for reading, with the aim of eliminating the impact of lock granularity and ensuring read parallelism under the same lock size. In the event that the Bloom filter in VFL deduces the nonexistence of a given key, SmartHT returns NONE to avoid excessive high-latency accesses to DCPMM over DRAM. On the other hand, if the key exists, a search of the chained hashing in PDL is needed to verify the existence of the key to avoid false positives. Consequently, SmartHT leverages a DRAM-based Bloom filter to accelerate the performance of negative queries. For positive queries, however, SmartHT still

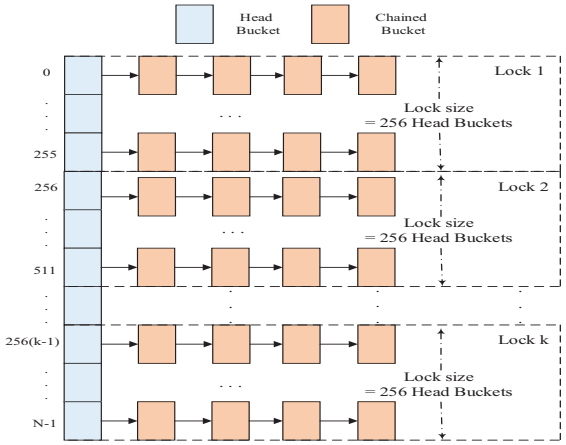

Fig. 4. The shared-read and exclusive-write lock-based concurrency control and each lock contain 256 64-byte head bucket range.

Algorithm 2: Search

```
1  if the Bloom filter (BF) contains the key then
2      Calculate hashVal and HBN according to the given
         key;
3      bucket_ptr = &Buckets[HBN];
4      while bucket_ptr != NULL do
5          set shared_lock Locks[hashVal/locksize] is true;
6          for i ← 0 to 2 do
7              if bucket_ptr→token[i] ==1 and
                 bucket_ptr→slot[i].key == key then
8                  return bucket_ptr→slot[i].value;
9              end
10         end
11         bucket_ptr = bucket_ptr→next;
12     end
13 end
14 return NONE;
```

needs to search the chained hashing of PDL in PM. Algorithm 2 displays the pseudo-code of SmartHT's search operation.

*3) **Delete**:* Similar to the search operation, SmartHT leverages the Bloom filter to optimize the deletion of non-existent key-value pairs. Moreover, it enhances deletion scalability by invalidating tokens (i.e., setting them to 0 and persisting them to PM), thereby implementing lazy deletion. SmartHT makes sure that parallel deletions are correct by utilizing exclusive write locks.

D. Recovery

SmartHT may lose the DRAM-based Bloom filter located in the volatile filter layer (VFL) after regular system shutdowns or failures. SmartHT can lazily recover the Bloom filter after a system failure, as all data exist in the hash table located in the persistent data layer (PDL) after the system failure.

Recovery following a normal system shutdown: When a system shutdown is regular, the recovery method is relatively simple. SmartHT initially persists VFL to a reserved position in PM and sets the valid flag (e.g., nonCrashed) to 1 to denote a normal shutdown. As all PDL data stored in PM is consistent, SmartHT can easily recover the small-sized VFL from the appropriate copy in PM. SmartHT then changes the nonCrashed flag to 0 upon successful recovery from a normal shutdown.

Recovery following a system failure: Recovering from unexpected system failures in SmartHT is more complex than normal system shutdowns. In such cases, the Bloom filter of VFL must be rebuilt by traversing through all keys in a consistent chained hashing of PDL in PM. SmartHT utilizes a token to indicate the validity of the key-value pair, where a token's value of one denotes a valid slot, and any other value implies an invalid one. SmartHT initializes a DRAM-based Bloom filter in VFL during the recovery process after a system failure. It then iteratively scans the key-value pairs in the PM-based chained hashing and checks their corresponding token values. When encountering an item with an invalid token, SmartHT skips it, and for an item with a token's value of "1", SmartHT reinserts it into VFL.

IV. Performance Evaluation

A. Experiment Setup

Hardware Configuration: Our experiment runs on a server environment equipped with four Optane DCPMMs of 128 GB each, resulting in a total capacity of 512 GB available in the App Direct mode. This setup has 128 GB of DRAM and runs on two Intel Xeon Gold 6240 CPUs. Each CPU has 18 cores, i.e., 36 hyper threads, and has an L1 cache of 1152 KB, an L2 cache of 18 MB, and an L3 cache of 24.75 MB. The server runs on Linux kernel version 5.15.0. We bind all Optane DCPMMs on one CPU to avoid the impact of NUMA. The multi-threaded environment utilizes two CPUs to evaluate the scalability of different PM-based hash indexes.

Implementation: To ensure a fair comparison, we adopted the evaluation technique proposed by HashEvaluation [12] and integrated SmartHT into the evaluation platform. We utilized the modified Level hashing version in HashEvaluation. In this version, the key and value are each set to 8 bytes, and each bucket has a size of 64 bytes and accommodates four key-value pairs. For PM management in SmartHT, CCEH [4], Level hashing [3], PCLHT [5], and SOFT [6], we leveraged the libvmem library available in PMDK [13]. While Dash [8] adopts libpmem and libpmemobj from PMDK to create a PM space, Clevel [7] uses the original implementation of libpmemobj c++ binding. To optimize performance, we utilized clwb instructions for all cacheline flush operations [5], while the hash function used the high-performance GCC std::Hash_Bytes, which has been commonly used in prior studies [3], [4], [12]. Additionally, we ensured that no duplicate keys existed in any persistent hash table.

Benchmark Framework: We utilized the PiBench [9] framework to evaluate various PM-based hash indexes, similar to HashEvaluation [12]. PiBench offers a wide range of metrics, such as throughput, latency, and low-level hardware perspectives utilizing interfaces provided by the Processor Counter Monitor library [14] to induce general operations like

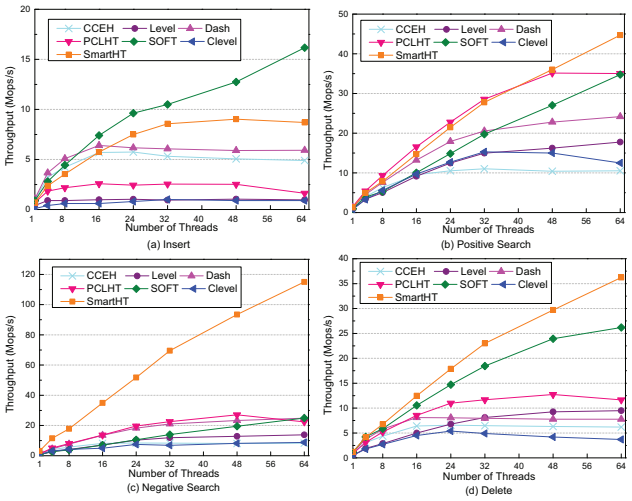

Fig. 5. Multi-threaded scalability for uniform workloads.

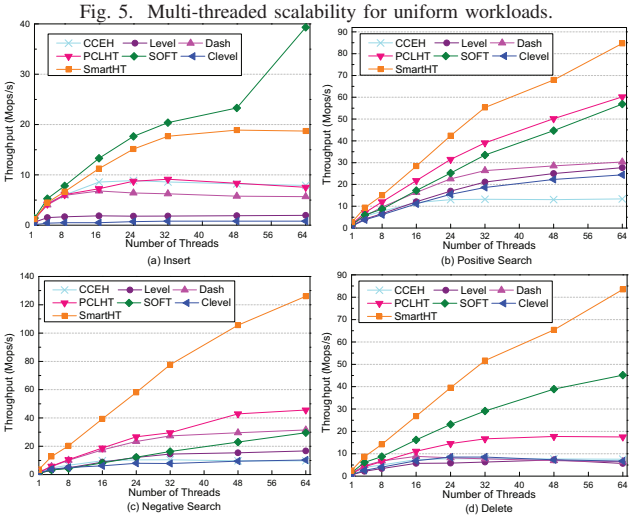

Fig. 6. Multi-threaded scalability for skewed workloads.

insert, search, and delete. We extended PiBench to evaluate other metrics, such as load factor and recovery time. PiBench allows for three random distributions, uniform, zipfian, and self-similar, during the workload generation phase.

Workloads: For insertion, we directly inserted 200 million key-value pairs into different persistent hash indexes. For positive/negative search and deletion, we first initialized 200 million key-value pairs of various hash indexes in the loading phase and then executed corresponding operations on these 200 million key-value pairs in the running phase to evaluate performance. We primarily used uniform and skewed distributions in our experiments, similar to previous studies [8], [9], [12]. The skewed distribution employed the self-similar distribution with a factor of 0.2 by default, which implies that 20% of the keys accounted for 80% of the accesses. SmartHT, CCEH, Level hashing, and PCLHT did not support variable-length keys and values. Therefore, we utilized 8-byte fixed-size keys and values for all PM-based persistent hash indexes.

B. Multi-threaded Performance

In our multi-threaded performance evaluations, we tested the scalability of various PM-based persistent hash indexes

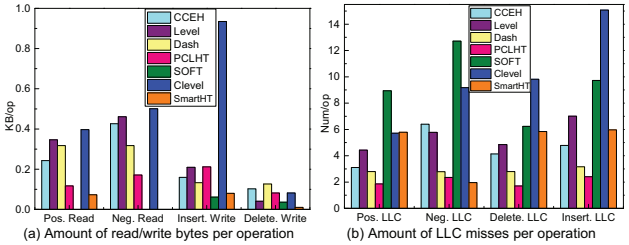

Fig. 7. Low level performance with processor count monitor describes the amount of (a) read/write bytes per operation and (b) LLC misses for various operations under uniform.

under different thread numbers ranging from 4 to 64 for insert, positive query, negative query, and delete operations under uniform and skewed distributions. Figure 5 and Figure 6 present the results of these evaluations.

Search. Regarding scalability in positive search operations, SmartHT outperforms other persistent hash indexes and achieves high throughput rates of 44.7 Mops/s and 84.7 Mops/s under 64 threads in uniform and skewed distributions, respectively, as demonstrated in Figure 5(b) and Figure 6(b). SmartHT's performance is 1.28x-4.25x and 1.41x-6.34x faster than other persistent hash indexes due to its shortened average chained length of the head-bucket to decrease the search path length.

SmartHT outperforms other indexes in negative query scalability, as Figure 5(c) and Figure 6(c) demonstrate. Under 64 threads, SmartHT achieves negative query throughput rates of 115.04 Mops/s and 125.98 Mops/s in uniform and skewed distributions, respectively. Compared to CCEH, Level, Dash, PCLHT, SOFT, and Clevel, SmartHT's negative query performance outperforms them by 4.61x-13.86x and 2.76x-12.99x, respectively. SmartHT's superiority in negative query performance is mainly attributed to the Bloom filter of VFL in DRAM. This filter enables the prediction of non-membership of key-value entries without costly access to the chained hashing of PDL in PM. Figure 7(a) shows that SmartHT can retrieve 0 amount of data from PM for each negative query, resulting in the lowest number of LLC misses as illustrated in Figure 7(b). Other persistent hash indexes, in contrast, experience lower efficiency when performing negative queries as they need to traverse long and complete query paths.

Insert. We analyzed the scalability of multi-threaded insertion and presented results in Figure 5(a) and Figure 6(a). SOFT displayed the highest insertion scalability among all tested persistent hash indexes due to its efficient lock-free insertion mechanism that minimizes lock-contention costs and reduces the need for memory fence (MFENCE) and cacheline flush (CLFLUSH) operations. This results in a significant reduction in the amount of data that needs to be written. SmartHT also showcased impressive insertion scalability by reducing the number of flushes with merge small writes and utilizing head insertion to shorten the insertion path. On the other hand, the less favorable insertion scalability of Level and Clevel can be attributed to their excessive use of lock-based resizing operations, leading to frequent data movement and additional PM writes, which hinders their scalability.

Delete. Figures 5(d) and 6(d) present the multi-threaded scalability results of various persistent hash indexes under uniform and skewed distributions, respectively. SmartHT achieved the best deletion scalability with 36.24 Mops/s and 83.5 Mops/s 64-thread deletion throughput under uniform and skewed distributions, respectively. Compared to CCEH, Level, Dash, PCLHT, SOFT, and Clevel, SmartHT outperformed them by 1.38x-9.79x and 1.85x-14.75x, respectively, because SmartHT's superior deletion performance results from efficient low-overhead deletion operations that utilize lazy deletion techniques. The lazy deletion mechanism of SmartHT enables the least amount of data being written during a deletion operation in all persistent hash indexes, as illustrated in Figure 7(a). SOFT significantly shortens the query path to improve query efficiency upon node deletion, leading to its second-best deletion scalability. PCLHT exhibits moderate deletion scalability due to its good positive query efficiency. However, other persistent hash indexes demonstrate limited deletion scalability because of their inefficient query or delete performance.

C. Tail Latency

In the context of large-scale storage systems, effectively managing and reducing tail latency is essential for delivering optimal user experiences. Figure 8 presents the results of tail latency performance for the tested persistent hash indexes. The results in Figure 8(c) demonstrate that SmartHT exhibits the lowest tail latency in negative queries, with p99.9 and p99.99 tail latencies of 1.09us and 2.52us, respectively. These are 4.88x-28.34x and 3.81x-20.33x lower than state-of-the-art persistent hash indexes. SmartHT's impressive performance is achieved by quickly identifying non-existent keys through the Bloom filter of the VFL located in DRAM, thus avoiding expensive PM accesses. SmartHT also shows relatively low tail delays for insertion, achieving a p99.9 insertion tail latency performance that surpasses other persistent hash indexes by 1.56x-7.30x. This performance is attributed to SmartHT's efficient merge writes and head insertion. Figure 8(b) also indicates that SmartHT exhibits relatively low tail latency performance in positive queries, showing a p99.9 and p99.99 tail latency that is 2.11x-5.04x better than other persistent hash indexes. Finally, SmartHT's deletion tail delay is also low, as shown in Figure 8(d), with p99.9 and p99.99 deletion tail latencies that are 1.46x-5.78x and 1.45x-32.84x lower than other persistent hash indexes, respectively. SmartHT achieves this excellent deletion performance due to its low deletion-consistency overhead.

D. Load Factor

The load factor of a hash table is determined by the ratio of the number of key-value pairs to the total number of allocated slots in the table. The load factor represents the space utilization rate, and a higher value indicates better space utilization. Figure 9 illustrates the maximum load factor for each persistent hash index. Remarkably, SmartHT exhibits an exceptionally high load factor of up to 94.1%. In contrast, SOFT demonstrates a constant load factor of one due to its demand-based slot allocation mechanism. Compared to other

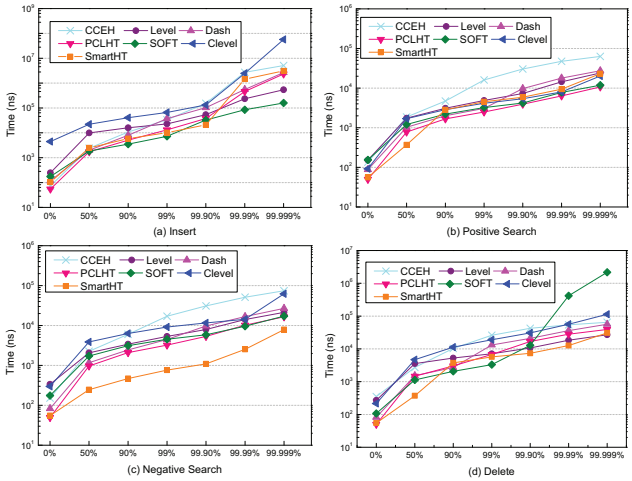

Fig. 8. Tail latency at different percentiles under the uniform distribution with 32 threads.

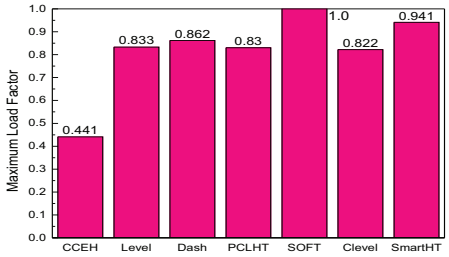

Fig. 9. Load factor for various persistent hash indexes.

persistent hash indexes, CCEH has the least space utilization and its maximum load factor of only 44.1%. The low load factor of CCEH results from its inherent limitation on probing up to 16 slots upon hash conflicts, coupled with suboptimal capacity expansion via segment splitting, leading to many unused key-value pair spaces. However, Dash's balanced insert, displacement, and stashing techniques allow for an improved load factor of up to 86.2%. Level, Clevel, and PCLHT demonstrate much better load factors than CCEH. The reason is that Level and Clevel rely on two hash functions, enabling them to detect and manage more slots. In contrast, PCLHT achieves a higher load factor by resolving hash collisions using chained hashing.

TABLE I
RECOVERY TIME (SECONDS) FOR SMARTHT WITH DIFFERENT DATA SIZE.

Hash Table	50 Million	100 Million	150 Million	200 Million
SmartHT	23.3	48.7	75.3	102.3

E. Failure Recovery

To evaluate SmartHT's recovery performance after a system failure, we inserted datasets ranging from 50 million to 200 million entries into the index and terminated the process. We then reconstructed the DRAM-based Bloom filter of VFL by performing a single-threaded read of the PM-based chained hashing of PDL and recorded the reconstruction time. Table I presents the recovery time results of SmartHT using different dataset sizes. The experimental results show that SmartHT's

recovery time exhibits a linear growth pattern as the dataset size increases. The maximum recovery overhead is 102.3 seconds when reconstructing 200 million key-value pairs. However, SmartHT's recovery time overhead is acceptable due to its capability of lazy recovery.

V. Related Work

PM-based Hash Indexes. PFHT [15] restricts to at most one cuckoo displacement, resulting in fewer PM writes than the original cuckoo hashing. Path hashing [16] employs an inverted binary tree structure and shortens the path to reduce lookup time. Group hashing [17] and HDNH [18] ensure data consistency on PM. Group hashing [17] uses an 8-byte failure-atomic write to guarantee data consistency without duplicate copies for logging or copy-on-write. Similar to CCEH [4] and Dash [8], REH [19], HASDH [20], PMEH [21], and OP-HMEH [22] are variants of extendible hashing on Intel Optane DC Persistent Memory (DCPMM).

Key-value Stores for Hybrid DRAM-PM Memory. SPHT [23], Viper [24], and Halo [25], similar to SOFT [6], store all their indexes in DRAM for high performance but at the cost of more DRAM space overhead or long recovery time cost. Flatstore [26] utilizes a DCPMM-based persistent log structure to build DRAM-assistant write-optimized key-value stores via batched small writes of the log structure. HiKV [27] is a hybrid DRAM-PM index, a B+Tree index stored in DRAM for fast search, and a hash index in PM for persistence.

VI. Conclusion

In this paper, we propose SmartHT, a PM-based hash table accelerated by a small DRAM-based Bloom filter, which can improve negative query performance and reduce the number of accesses to PM. SmartHT leverages efficient merge writes with head insertion, lazy deletion, and a shortened average chained length of head-bucket to improve the multi-threaded insertion/deletion/positive-search scalability, respectively. Furthermore, SmartHT uses a merged-flush mechanism based on an 8-byte failure-atomic write method to achieve low data consistency overhead. Experimental results on Intel Optane DCPMM demonstrate that SmartHT significantly outperforms existing persistent hash indexes.

References

[1] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARH Computer Architecture News*, vol. 37, no. 3, pp. 24–33, 2009.

[2] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Berkeley, CA, USA: USENIX, 2020, pp. 169–182.

[3] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *OSDI*. Berkeley, CA, USA: USENIX, 2018, pp. 461–476.

[4] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *FAST*. Berkeley, CA, USA: USENIX, 2019, pp. 31–44.

[5] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: Converting concurrent dram indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2019, pp. 462–477.

[6] Y. Zuriel, M. Friedman, G. Sheffi, N. Cohen, and E. Petrank, "Efficient lock-free durable sets," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–26, 2019.

[7] Z. Chen, Y. Hua, B. Ding, and P. Zuo, "Lock-free concurrent level hashing for persistent memory," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. Berkeley, CA, USA: USENIX, 2020, pp. 799–812.

[8] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, 2020.

[9] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 574–587, 2019.

[10] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H. Du, "Bloomflash: Bloom filter on flash-based storage," in *2011 31st International Conference on Distributed Computing Systems*. Piscataway, NJ, USA: IEEE, 2011, pp. 635–644.

[11] L. Luo, D. Guo, R. T. Ma, O. Rottenstreich, and X. Luo, "Optimizing bloom filter: Challenges, solutions, and comparisons," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 2, pp. 1912–1949, 2018.

[12] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen, "Persistent memory hash indexes: an experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 785–798, 2021.

[13] Intel, "Persistent memory development kit," https://pmem.io/pmdk/, 2022.

[14] I. C. et al., "Processor counter monitor," https://github.com/intel/pcm, 2022.

[15] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting hash table design for phase change memory," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, pp. 18–26, 2016.

[16] P. Zuo and Y. Hua, "A write-friendly hashing scheme for non-volatile memory systems," in *MSST*. Piscataway, NJ, USA: IEEE, 2017, pp. 1–10.

[17] X. Zhang, D. Feng, Y. Hua, J. Chen, and M. Fu, "A write-efficient and consistent hashing scheme for non-volatile memory," in *ICPP*. New York, NY, USA: ACM, 2018, p. 87.

[18] J. Zhu, K. Huang, X. Zou, C. Huang, N. Xu, and L. Fang, "Hdnh: a read-efficient and write-optimized hashing scheme for hybrid dram-nvm memory," in *50th International Conference on Parallel Processing*. Piscataway, NJ, USA: IEEE, 2021, pp. 1–10.

[19] Z. Li, Z. Tan, and J. Chen, "Reh: redesigning extendible hashing for commercial non-volatile memory," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Piscataway, NJ, USA: IEEE, 2022, pp. 742–747.

[20] ——, "Hasdh: A hotspot-aware and scalable dynamic hashing for hybrid dram-nvm memory," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 2021, pp. 154–161.

[21] J. Hu, J. Chen, Y. Zhu, Q. Yang, Z. Peng, and Y. Yu, "Parallel multi-split extendible hashing for persistent memory," in *50th International Conference on Parallel Processing*. New York, NY, USA: ACM, 2021, pp. 1–10.

[22] X. Zou, F. Wang, D. Feng, J. Zhu, R. Xiao, and N. Su, "A write-optimal and concurrent persistent dynamic hashing with radix tree assistance," *Journal of Systems Architecture*, vol. 125, p. 102462, 2022.

[23] X. Zou, F. Wang, D. Feng, F. Yang, M. Lei, and C. Liu, "Spht: A scalable and high-performance hashing scheme for persistent memory," *Software: Practice and Experience*, no. 7, pp. 1679–1697, 2022.

[24] L. Benson, H. Makait, and T. Rabl, "Viper: an efficient hybrid pmem-dram key-value store," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1544–1556, 2021.

[25] D. Hu, Z. Chen, W. Che, J. Sun, and H. Chen, "Halo: A hybrid pmem-dram persistent hash index with fast recovery," in *Proceedings of the 2022 International Conference on Management of Data*. New York, NY, USA: ACM, 2022, pp. 1049–1063.

[26] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2020, pp. 1077–1091.

[27] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *ATC*. Berkeley, CA, USA: USENIX, 2017, pp. 349–362.