R^2Hash: A Read-Optimized and Resize-Friendly Hashing Index for Persistent Memory

Jinlei Hu, Bo Chen, Miaosong Zhang, Jing Hu, Jianxi Chen*, Dan Feng*
Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System of MoE,
School of Computer Science and Technology, Huazhong University of Science and Technology
{hujinlei,brianchan,zhangms,hujingreal,chenjx,dfeng}@hust.edu.cn,

***Abstract*—Persistent memory (PM) offers a compelling combination of durability and near-DRAM performance, but it also introduces new challenges for hashing indexes. Existing persistent hashing designs prioritize resizing efficiency at the expense of increased query latency, losing the key advantage of hash tables. This paper introduces R^2Hash, a persistent hash index redesigned from the persistent cache-line hash table, to balance high read performance with efficient resizing. R^2Hash is guided by a migration rule, enabling it to meet both goals through two main contributions: (i) a cooperative and low-overhead resizing strategy based on split-order hashing, and (ii) a shift-aware search combined with a two-layer bucket layout that enables lock-free reads with only one PM access on average. Furthermore, R^2Hash provides log-free consistency and a non-blocking recovery mechanism. Experimental results demonstrate that R^2Hash achieves up to 8.1× higher search throughput and 7.5× higher insert throughput compared to other persistent hash indexes across a range of workloads.**

I. Introduction

Persistent Memory (PM) combines the capacity of traditional storage with DRAM-like performance, offering byte-level access and data persistence. It can be implemented using various technologies, including PCM, STT-RAM, 3D XPoint, and emerging CXL-based storage devices [1]. PM has already shown its potential in enhancing the performance of hashing-based key-value caches such as Memcached [2] and Redis [3].

Unlike tree indexes, which suffer significant query overhead due to their pointer chasing, hash indexes eliminate hierarchical structures and offer consistent point query performance. This feature is crucial for indexing in-memory key-value cache systems. And the hash index employed in these storage systems demands both low-latency read operations and high-throughput write operations, as emphasized in a report by Twitter [4]. Resizing is the primary performance bottleneck for persistent hash indexes. When the number of key-value pairs approaches the table's capacity, resizing is triggered, requiring expensive data migration to a larger table, significantly degrading write performance. Some hashing indexes lock the entire table [5]–[7], causing read operations to be blocked during resizing.

This work is funded by the National Key Research and Development Program (No.2024YFB4505104), National Natural Science Foundation of China (NSFC No.U22A2027). Corresponding authors: Jianxi Chen, Dan Feng.

Several research efforts [5], [7]–[9] have aimed to redesign persistent hash indexes based on PM features, optimizing the resizing process. However, these approaches compromise the primary benefit of hash indexes: low query latency. PCLHT is a PM adaptation of the cache-line hash table [6], originally designed for volatile DRAM to minimize query overhead using simple bucket chaining. On average, each head bucket links to only one additional overflow bucket, providing shorter access for read operations than other hashing indexes. To support lock-free reads during resizing, PCLHT uses a copy-on-write (COW) approach at the table level by maintaining a full copy of the hash table. However, this mechanism introduces substantial overhead during resizing. In this work, we focus on optimizing the PCLHT to reduce resizing costs while preserving its fast query performance.

We found that the *migration rule* can enable PCLHT to achieve the best of two worlds: high read performance and an efficient resizing mechanism. When the resizing operation doubles the table, the migration rule refers to each key-value pair in the new table is either placed at the same offset as in the original table or shifted by the original table's size. This predictable pattern allows us to split original linked buckets proactively: one part remains in place, while the other is directly moved to its new offset in the expanded table. While this approach reduces migration overheads of resizing, it introduces a key challenge for ensuring consistent lock-free reads. Since the original table is reused and migration is not atomic, inconsistent reads can occur if a lookup happens during a partial migration.

Guided by the migration rule, we propose a read-optimized and resize-friendly hashing index called R^2Hash, to balance high read performance with efficient resizing. Specifically, we made the following contributions:

- **Efficient Resizing.** R^2Hash utilizes two mechanisms to improve the efficiency of the resizing mechanism on PM: *the split order mechanism* and *cooperative resizing*. R^2Hash accelerates the migration operation by inserting new key-value pairs based on the split order. During rehashing, only consecutive key-value pairs with valid split orders are transferred to the newly created hash table, requiring only one persistent operation. Through cooperative resizing, R^2Hash improves resizing operation concurrency to obtain higher PM bandwidth

utilization without extra cost.

• **Optimized Read.** The read operation in R^{2}Hash is highly optimized, requiring only one PM access on average, even under frequent hash collisions. This is achieved through the combination of two methods. Firstly, the *shift-aware read* approach ensures that the read operation is lock-free and guarantees a maximum retry count of three. Secondly, R^{2}Hash uses the *two-layer buckets* technique to avoid unnecessary accesses to PM and reduce read latency.

• **Comprehensive Evaluation.** Compared with state-of-the-art persistent hashing indexes, R^{2}Hash achieves up to 8.1x higher search throughput and 7.5x higher insert throughput under YCSB and PiBench [10] workloads. Additionally, the resizing time is reduced by up to 86.2%.

II. Background

A. Persistent memory

Persistent memory (PM) aims to bridge the gap between fast, byte-addressable DRAM and high-capacity, block-based flash storage by offering near-DRAM speed with non-volatile characteristics. PM is especially beneficial for workloads with random and small-sized accesses, such as those found in core storage structures like hash indexes. Current large-capacity PM products include Intel *Optane* (3D XPoint) and emerging CXL-based solutions such as memory-semantic SSDs [11]. Despite architectural differences, these products face common challenges: limited bandwidth, lower random access performance, and greater read-write asymmetry compared to DRAM, which constrains the performance of persistent hash indexing. For instance, Intel Optane's random read bandwidth is roughly one-third that of DRAM, while its random write bandwidth is only about one-tenth. It also suffers from significantly higher read and write latencies and experiences write amplification when access sizes are smaller than its 256-byte minimum write granularity [9]. Similarly, upcoming CXL-SSDs are expected to have SSD-level page sizes as their minimum write unit [11].

B. Persistent Hashing

Hashing indexes are widely used in high-performance in-memory caches due to their core advantages of low query latency [2], [3]. However, their performance remains limited on PM, which introduces significant overhead compared on DRAM. This is particularly evident during hash table resizing, which incurs costly additional query latency. To reduce the frequency of resizing, existing designs often adopt complex hashing collision resolution strategies, which come at the cost of read performance. Moreover, queries may be blocked during resizing, leading to high access latency.

As illustrated in Figure 1, most existing persistent hash indexes focus on optimizing the resizing process, but this often compromises the primary benefit of hash indexes (low query latency): *1)* Level-based hashing uses a hierarchical structure for resizing [5], [8], [12]. During lookups, it sequentially checks top-level buckets mapped by two hash functions, followed by corresponding lower-level buckets. Because of

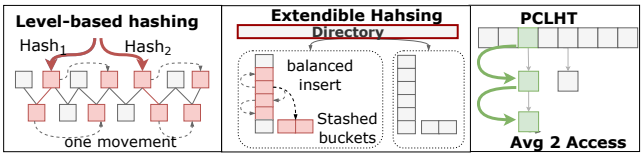


Fig. 1: PCLHT shows lower query access overhead.

the one movement methods, Level-based hashing also need to check additional buckets. During inserting a new item, if the two top-level buckets are full, Level-based hashing check whether it is possible to move any key-value item from one of its two top-level buckets to its alternative toplevel location. The same one movement process is excuted for the bottom-level buckets. This results in long query paths and increased lookup latency. *2)* Extendible hashing require an additional directory lookup [7], [9], [13]. Their insertion strategy spreads key-value pairs across up to four neighboring buckets and includes two stash buckets per segment to handle overflows, further extending the query path on PM. *3)* The Persistent Cache-Line Hash Table (PCLHT) [6] minimizes query overhead by resolving collisions through simple bucket chaining. On average, each head bucket links to only one additional linked bucket, providing faster lookups than other approaches. PCLHT also supports lock-free reads by maintaining a copy of the entire table, which enables concurrency but incurs high overhead during resizing. In this work, we focus on optimizing the PCLHT design to reduce resizing costs while preserving its low query overhead.

C. Problems and Motivation

Problems: PCLHT provides excellent read performance with inefficient resizing operation. As shown in Figure 2-a, PCLHT achieves the highest read throughput among existing persistent hashing indexes. However, its write performance is slightly lower than that of level-based hashing indexes (e.g., LEVEL [14] and CLEVEL [15]), and significantly lower than extendible hashing indexes (e.g., DASH [9] and CCEH [14]). Our analysis attributes this limitation to PCLHT's costly resizing process (Figure 2-b). When resizing is not triggered (PCLHT-write), its write throughput approaches that of DASH. However, once triggering resizing (PCLHT), its write performance drops sharply. This is because resizing requires copying the entire hash table and sequentially migrating all buckets to a new table that is twice the original size. Although this design supports lock-free reads, the associated overhead of migration significantly impact write efficiency. As shown in Figures 2-c, while the total data moved during resizing is comparable to other hash table designs, PCLHT incurs significantly higher overhead in memory allocation and reclamation. This leads to a slower overall resize process. Further analysis (Figures 2-d) reveals two main sources of overhead during resizing. First, in the Create phase, a new, double-sized hash table must be allocated, often blocking insertion threads. Second, during the Rehash phase, all key-value pairs must be migrated to the new

Fig. 2: Resizing overheads analysis by evaluating PCLHT with other persistent hashing indexes.

table, resulting in frequent random writes and additional effort to reclaim obsolete hash-table space.

Motivation: reducing redundant migration during resizing via the migration rule. In PCLHT, resizing the hash table usually requires doubling its capacity, which involves migrating key-value pairs. We define the migration rule of PCLHT that during the resizing process, the offset of each key-value pair in the new table is either the same as in the original table or shifted by the size of the original table. Leveraging this predictable migration pattern, we can enable the reuse of the original hash table, and proactively split each original bucket into two: one remains in place, and the other is directly moved to its new offset in the new table. This approach effectively halves the number of key-value migrations. However, this optimization introduces a key challenge: **maintaining consistency for lock-free reads.** PCLHT traditionally copies the entire hash table during resizing to ensure that ongoing reads can safely complete before deallocating the original table. Reusing the original table disrupts this guarantee, as key-value pairs may reside in either the old or new table, and migration is not atomic. This can lead to inconsistent reads if a partially migrated entry is accessed. To address this, we introduce a slot-level copy-on-write (COW) mechanism to preserve the consistency of lock-free read operations during migration.

III. The Design of R^{2}Hash

A. Overview

Based on PCLHT, R^{2}Hash aims to reduce resizing costs while preserving its low query overhead. R^{2}Hash adopts the linked overflow buckets similar to PCLHT, reducing the average access length in the read operation. Inspired by the migration rule, R^{2}Hash preserves the previous hash table using a directory structure. Unlike classic extendible hashing, R^{2}Hash uses a fixed directory that consists of 64 segments, which eliminates the overheads associated with doubling the directory. Each segment holds an 8-byte pointer to the first address of a head bucket array in the hash table, as shown in Figure 3. The size of each new head bucket array is the same as the all previous head buckets. Assuming the size of the first head bucket array is $base$, the fixed directory with

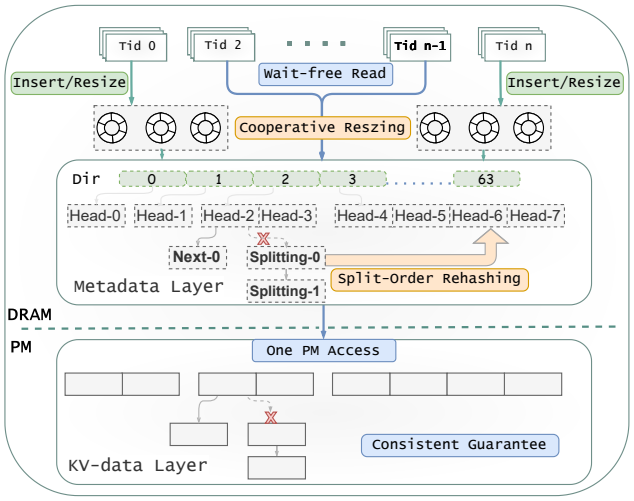


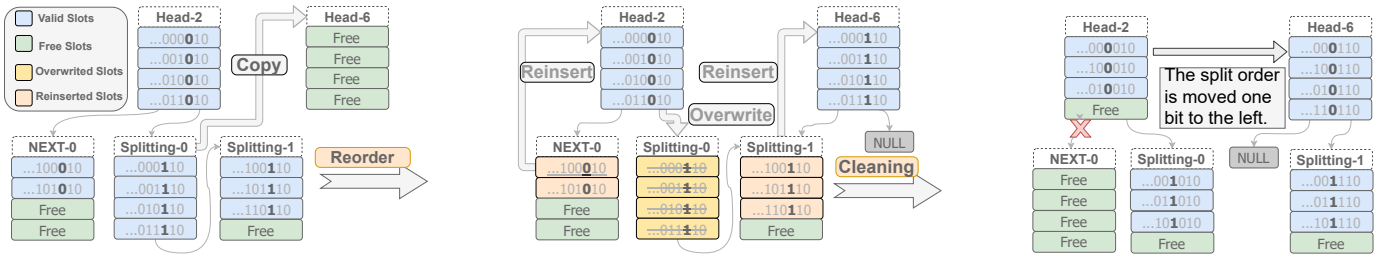
Fig. 3: The R^{2}Hash architecture overview.

64 slots can store up to $2^{64} * base$ buckets. The total size of the specified directory is 512 bytes, and it is placed in DRAM, which has minimal impact on reading performance. As shown in Figure 3, the structure of R^{2}Hash consists of two layers: metadata layer and kv-data layer. This design effectively avoids unsatisfactory PM bandwidth under small-size random accesses.

B. Efficient Resizing Based on Split Order

The fundamental principle of the split-order mechanism is to continuously position the key-value pairs requiring migration to the splitting overflow bucket. By adopting this method, we can reduce nearly half of the migrated key-value pairs.

The Split-Order Mechanism. We define split order based on the migration rule first. The key-value pair with *false* split order, which will stay in the original bucket during the next resizing. The *true* split order means that $hash(key)\ mod\ 2N \neq hash(key)\ mod\ N$ is satisfied, will migrate to the new position. When a head bucket in R^{2}Hash becomes full, a bucket split operation is triggered, generating a new overflow (next or splitting) bucket. In contrast to PCLHT, the overflow bucket in R^{2}Hash is divided into two linked lists. One consists



(a) Only copy Splitting-0 to the Head-6. (b) Sorted on the next/new split order. (c) Sorted on the new split order.

Fig. 4: The Split-order rehashing Operation of R^{2}Hash.

of a head bucket and its subsequent buckets, called the next list, while the other consists of a splitting bucket and its succeeding buckets, called the splitting list. The split order of key-value pairs in the splitting list is true, and only one migration operation is required for them during the next resizing. The split order of the next list is all false, meaning they remain in place during resizing. PCLHT expands by allocating a new hash table that is twice the size of the original and then migrates each bucket before reclaiming the old table. In contrast, R^{2}Hash uses a split-order mechanism that requires allocating a new table of the same size, retaining nearly half of the existing buckets. As an example, Head-2 and Next-0 in Figure 4a form the next list because their slots have the same false split order. The false split order, third bit from the right, indicates that they will remain at their original offset when doubling the hash table from 4 to 8. The Splitting-0 and its potential linked bucket will form the splitting list. The current size of the hash table is 4, and all their third bits in the splitting list are one. These slots will be transferred to the new position: Head-6.

Efficient Resizing Process. To determine whether to trigger a hash table resizing, we establish *a resizing threshold* which is defined as the number of overflow buckets divided by the number of head buckets. By default, we set this threshold to one, resulting in the majority of head buckets containing only a linked splitting bucket. In most cases, the migration process only needs to copy the first splitting bucket to the new hash table instead of all the splitting buckets if exists. This is because the first splitting bucket is the head node of the subsequent original splitting bucket. And if any next buckets exist, there needs to be a reorder phase to regenerate the split order. In the reorder phase, all linked buckets will be reinserted into the corresponding head bucket and sorted according to the split order at the next resizing. In our tests, the probability of reorder triggering is no more than 3% with the default resizing threshold. Typically, a reorder operation adds only one additional read and write to PM, which introduces negligible overhead. If the head bucket does not split up, all the slots to be migrated will be discontinuously located in the head bucket. In this rare case, traditional migration methods are required. As depicted in Figure 4a, the splitting-0 bucket needs to be copied to the new Head-6 bucket because all the slots in the splitting list have the true split order. Unlike PCLHT's rehashing operations that copy the discontiguous slots in entire overflow bucket lists, split-order rehashing directly copies one bucket, which significantly helps to reduce migration overhead. After the bucket is copied in Figure 4b, R^{2}Hash will re-insert the valid slots of the next buckets into the head bucket in rare cases. The goal is to obtain a new split order according to the new hash table length. When the head bucket begins to split, an allocation for the new bucket is unnecessary. Instead, R^{2}Hash directly reuses the migrated splitting bucket. The newly migrated split bucket is temporarily cached on the free-bucket list, ensuring that the read operation does not access released memory and avoiding the overhead of repeated memory allocation and deallocation.

Cooperative Resizing. R^{2}Hash accelearte the migration phase by increasing the concurrency of the resizing operation without the need for additional threads. It is achieved by reusing subsequent insert/update threads. In R^{2}Hash, *the resizing degree*, denoted as T, represents the maximum number of threads capable of performing rehashing operations concurrently. The hash table is divided into T blocks, each containing a continuous head bucket. Each insert/update thread fetches a block, enabling it to migrate data within that specific block. If all blocks have been dispatched, the new coming thread has to wait for other rehashing threads to complete their tasks. Once all blocks have been migrated, the thread responsible for the last block atomically releases the new hash table length, indicating the completion of resizing. Furthermore, the rehashing bucket ID of every thread is persisted to ensure recovery from a crash during resizing. T is set to 24 based on the test described in Sec.IV-D.

C. Read-Optimized Through One PM Access

By combining the wait-free read mechanism with a two-layer bucket, R^{2}Hash ensures that the optimized read operations require only one PM access on average even during resizing. Additionally, R^{2}Hash takes a different approach compared to existing solutions, which store fingerprints in PM [9]. Instead, R^{2}Hash stores fingerprints in DRAM to achieve higher acceleration effects.

Two-layer Bucket Architecture. As illustrated in Figure 3, the hashtable structure of R^{2}Hash consists of two layers: a key-value data layer in PM and a metadata layer in DRAM. Each bucket of R^{2}Hash contains a 256-byte key-value data

bucket in PM and a 32-byte metadata bucket in DRAM. The query request first hits the metadata layer and then jumps to the corresponding key-value data layer. The two layers of the head buckets share the same offset in their respective bucket arrays, allowing for direct jumps from the metadata to the key-value data. Whereas the splitting *ptr* of the overflow bucket in the meta layer is multiplexed to point to the key-value data bucket. The bucket size can be adjusted, and in our experiments, we set the data layer size to 256 bytes, taking into account the minimum write granularity of *Optane*. In the event of a recovery, the metadata layer can be reconstructed from the persistent key-value data layer. Both buckets have a next and a splitting pointer. The key-value data bucket stores the actual key-value pairs, while the metadata bucket contains 1-byte locks and 15-byte fingerprints. R^{2}Hash leverages the most significant byte of the key's hash value to predict the possible existence of a key-value pair in the data layer. The remaining data bucket is divided into multiple 16-byte slots, with each slot storing a key-value pair. The first 8 bytes of each slot store the key, and the remaining 8 bytes store the value. If the need arises to support variable-length key-value pairs, the value can be replaced with a variable-length pair storage address [15]–[17].

Wait-free Read Mechanism. In the following, we discuss how R^{2}Hash resolves the read-insert conflicts.

1). Wait-free read without retry. R^{2}Hash uses 8-byte key-value pairs, which enables atomic operations under the x86 architecture in the case where the writer does not trigger resizing. As shown in Figure 5, R^{2}Hash uses the fingerprint as the key for the global visibility flag to maintain the global visibility of the inserted key-value pairs. When inserting a new key-value pair, the value is written first, followed by the key, and finally, the fingerprint is updated. Until the fingerprint is updated, other read operations will see the visibility of the key-value pair. If the key has not been persisted, the insert operation is considered incomplete. The read operation will reconfirm the key when obtaining the value to avoid inconsistencies.

2). Wait-free read within one retry during splitting. When an insert thread triggers a head bucket split and another read thread enters, the original key in the head bucket is deleted after migration. If the read thread does not detect the split, it will return false indicating an unsuccessful search. Unlike optimistic locking with keeping retry, R^{2}Hash is designed to know where the target slot is stored and in which direction the slot moves through the migration rule. Therefore, finding the target slot in R^{2}Hash typically requires at most one retry, as shown in Figure 6. If the target key-value pair with true split order is not found in the head bucket, and a splitting bucket exists, a re-search will be performed in the splitting bucket. Moreover, since most negative searches are filtered by fingerprints, only a very small fraction of positive searches require additional query overhead. When splitting linked buckets, such as the next bucket and the splitting bucket, read concurrency consistency remains unaffected. As they do not involve key-value pair migration and require only a simple compare-and-swap(CAS) synchronization instruction.

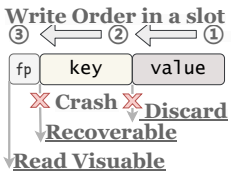


Fig. 5: The consistent guarantee in a slot.

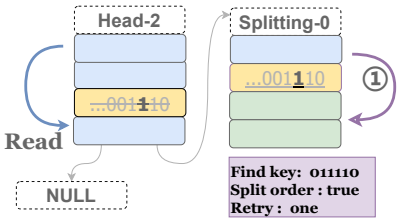


Fig. 6: Wait-free read within one retry during splitting.

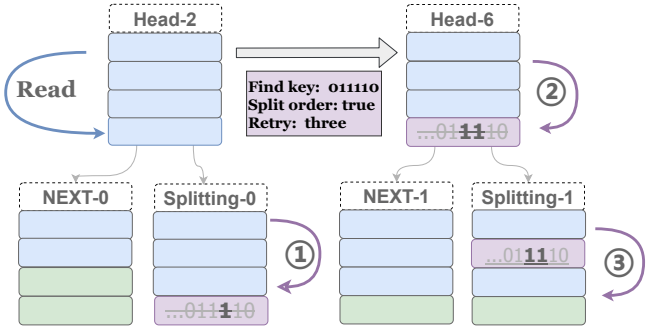


Fig. 7: Wait-free read within triple retries during resizing.

3). Wait-free read within triple retries during resizing. A read operation may not find the target key-value pair during resizing. Since the read operation does not hold any locks, the target key-value pair may be at the original offset before the expansion or have migrated to the new bucket. As illustrated in Figure 7, R^{2}Hash uses shift-aware search based on the migration rule to ensure that the target pair can be found within three retries during resizing. Notably, the shift-aware search does not require additional metadata or structural elements, because key-value pairs have a fixed migration order according to their split order. (i) If the key's split order is false, which means that the key-value pair will remain in place after rehashing, it only needs to search the head bucket and its next bucket for the first time. If the key is not found during the first search, it can be concluded that either it does not exist or may have been migrated to the splitting bucket during the reordering phase. To further investigate the key's existence, a second search will then be performed on the splitting list in the original offset. If both searches are unsuccessful, it is clear that the key-value pair does not exist. (ii) If the key's split order is true, R^{2}Hash initially searches for it in the splitting list. If the target key is not found, it may not exist or have been migrated to the new head bucket in the newly created hash table. It then continues to jump to the new head bucket to search. Furthermore, the new head bucket will be searched the same way as in (i), requiring at most two additional searches. To ensure wait-free read operations, bucket deallocation (i.e., recycling empty buckets) should only occur after all current readers have finished using the bucket. To achieve this, we utilize a free bucket list that caches empty buckets without causing excessive overhead.

Consistency Guarantee The consistent guarantee in a slot is shown as Figure 5. For the head bucket split process, R^{2}Hash uses a fine-grained delete-after-copy mechanism. When the head bucket splits, the slots with true split order will be copied to the splitting bucket first, and then deleted in the head bucket. If no available slots in the linked bucket, a new bucket is created and linked without any migration of slots. Intel PMDK module guarantees the creation of buckets for crash consistency, while CAS instructions guarantee the modification of the next pointer for concurrency. Furthermore, R^{2}Hash employs a locking mechanism based on the head bucket during write operations. This means that only one write operation can modify a head bucket and its corresponding linked buckets at a time. During the process of shrinking, we employ the classic merge mechanism like PCLHT.

Lazy-Rebuilding. The recovery process of R^{2}Hash is divided into four steps: ❶ allocate DRAM space for the metadata layer; ❷ remove any duplicate data in the bucket; ❸ rebuild the fingerprint and pointer of the metadata, and ❹ continue with potential ongoing resizing operation. During the rebuilding process, buckets that have been deduplicated and whose metadata has been rebuilt will continue to provide read services. Similar to cooperative resizing, the recovery operation also reuses subsequent operations to execute the following three steps. In this way, recovery time is mainly dependent on DRAM space creation, significantly reducing cold start time.

IV. Evaluation

In this section, we compare R^{2}Hash with state-of-the-art persistent hashing. We mainly confirm the following contributions through comprehensive evaluations: (1) excellent read performance, (2) efficient resizing and recovery mechanism, and (3) high-performance resource utilization.

A. Experimental Setup

Hardware Environment. Our experiments are performed on a Ubuntu 20.04 server (kernel version 5.15), equipped with two Intel Xeon Gold 6330 CPUs, both having 56 logical cores. The system has 256 GB of 2666MHz DDR4 DRAM (four 32 GB DIMMS per socket) and 1TB of *Optane DCPMM* (four 128GB Barlow Pass DIMMs per socket) configured in the App Direct mode. In our evaluation, threads are pinned to NUMA node 0 and only allowed to access the local DRAM and PM.

Implementation. We compare R^{2}Hash against five state-of-art persistent hashing indexes, including LEVEL, CLEVEL, CCEH, PCLHT, and DASH. For a fair comparison, we set all the hashing indexes to use the default parameters as in their original papers. All the code is compiled using GCC 9.5 with all optimization enabled. We use *clwb+sfence* instructions in eADR mode to persist a store and 128-bit SIMD instructions to accelerate the fingerprint comparison.

Workloads. Our evaluation uses workloads generated with PiBench and YCSB, as listed in Table I. PiBench generates a skewed load, while YCSB is evenly distributed. The columns and rows of the table can be combined to indicate specific workloads, such as PiBench-a and YCSB-a for the two workloads in *column a*. Unless otherwise stated, we use the following method for stress testing. We initialize hash tables with a capacity that can accommodate 13 million pairs. To measure insert-only performance, we directly insert 200 million key-value pairs into an empty hash table. Thus it will trigger the resize operation in the insert-only workload. For other workloads, we first load 200 million key-value pairs into the hashing and then execute 200 million operations to perform the evaluation. By default, the key and value are both 8 bytes.

TABLE I: The Workloads Configuration. The prefix characters W, R, P, N, D, and U before the numbers stand for the operations *insert, read, positive search, negative search, delete and update*, respectively.

Benchmarks	Workloads				
	a	b	c	d	e
PiBench-Skew	W100	P100	N100	D100	U100
YCSB-Uniform	W100	W50 R50	W5 R95	R100	\

Implementation assumption. Although we employ *Optane* for evaluation, the assumptions to PM in R^{2}Hash are mostly based on its generic and important properties, such as byte addressability, non-volatility, access latency between DRAM and SSD, and high capacity at low cost. These assumptions are independent of the specific characteristics of *Optane*'s internal media. Therefore, we believe that R^{2}Hash will be effective for future forms of PM, such as CXL storage devices [11].

B. Overall Performance

In this section, we present R^{2}Hash's overall performance using the workloads listed in Table I. However, we have observed that the variable-length value test exhibits a similar trend to that of the head slots. Due to space limitations, we have omitted the detailed results of the variable-length tests. For hashing indexes that require full table resizing, it will show the highest insert performance but the lowest search performance before the resizing, and vice versa. To illustrate the disparity between these two states, we evaluate the performance before the next resizing in PiBench workloads and after resizing in YCSB workloads.

Exp#1 Overall Performance. Figure 10 shows that R^{2}Hash outperforms DASH by up to 1.32× and the other four indexes by up to 1.57∼5.53× in YCSB-a workload using 56 threads, mainly due to its lower overheads for memory traversal and creation. In YCSB-b and YCSB-c workloads, R^{2}Hash outperforms the other five indexes by 1.26∼5.78× using 56 threads. We verified the mechanism and performance under mixed workloads by ensuring that PCLHT has resizing operations under YCSB-b workload but not YCSB-c workload. So its write performance will not be influenced by the blocking resizing. As the proportion of pure read operations increases, the read gap between PCLHT and R^{2}Hash narrows. In YCSB-c and YCSBD-d, PCLHT achieves the best performance except for R^{2}Hash, mainly because of its wait-free read. However, R^{2}Hash still has better read performance than PCLHT. R^{2}Hash demonstrates much better performance in PiBench, especially

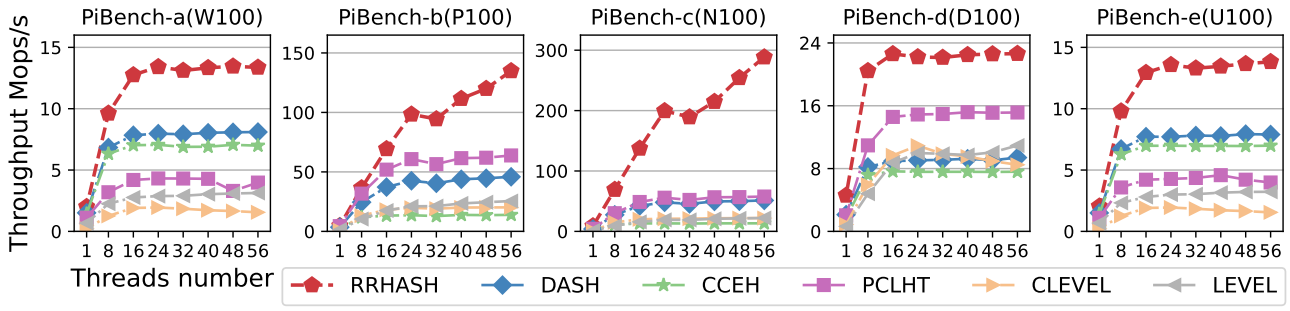


Fig. 8: The PiBench Benchmarks with Persistent Hashing

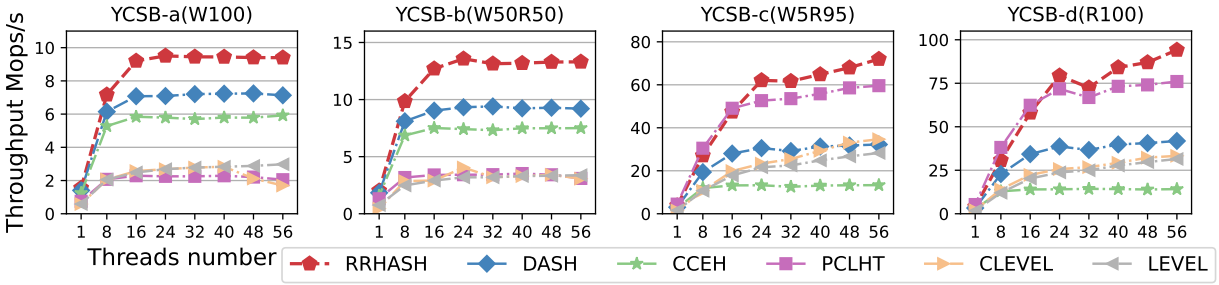


Fig. 9: The YCSB Benchmarks with Persistent Hashing

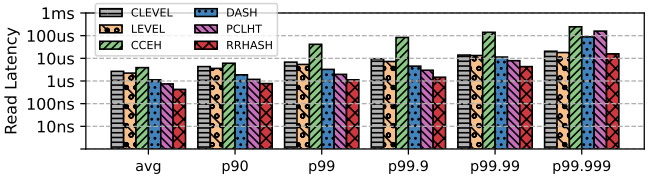


Fig. 10: The read latency under YCSB-b with 56 threads.

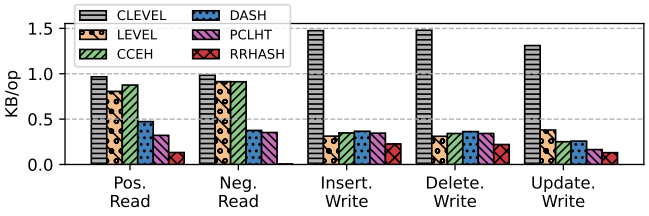


Fig. 11: The amount of read/write bytes per operation on the PM device under different PiBench workloads.

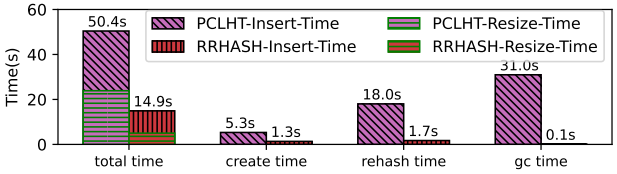


Fig. 12: The insert and rehashing time for R^{2}Hash and PCLHT under PiBench-a workload.

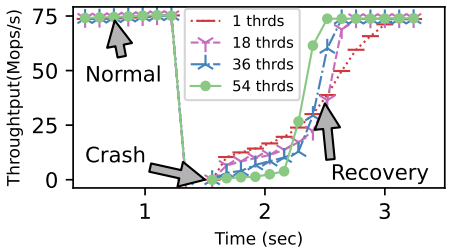


Fig. 13: Non-blocking recovery under different cooperative threads.

in the skewed workloads. The pure positive and negative lookups are significantly faster, because of the efficiency of the two-layer bucket and the wait-free read mechanism.

C. Effects of All Methods

Exp#2 Read latency. The read latency of each hashing index is shown in Figure 10 under a mixed workload YCSB-b with 56 threads. In average latency, R^{2}Hash can outperform all other hashing indexes by 42.4% $\sim$ 89.1%. Even in the other $9th$ percentile, R^{2}Hash still has an order of magnitude lower read latency because of its fewer PM accesses and wait-free within three retries.

Exp#3 Hardware metrics. We use the Intel Performance Counter Monitor to collect the amount of read/write bytes per operation on the PM device under all 5 PiBench workloads. As shown in Figure 11, we can find that R^{2}Hash has the lowest read and write bytes per operation. The average number of bytes read by a positive read is only 133 bytes, which is

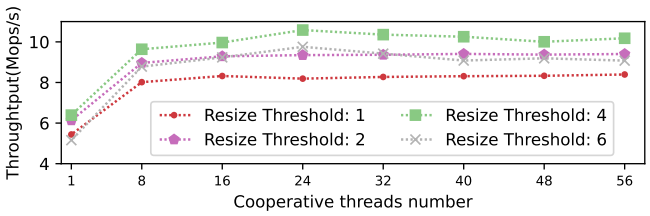


Fig. 14: The write performance of different resizing degrees and threshold.

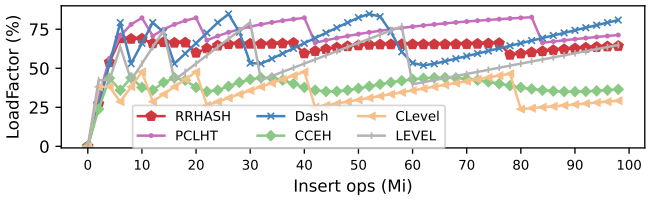


Fig. 15: The load factor of all persistent hashing.

only 27.9/41.1% of DASH/PCLHT. This shows that the read mechanism of R^{2}Hash significantly reduces the average PM accesses. The negative read basically accesses the DRAM, and the number of accesses to the PM is negligible. Although R^{2}Hash needs to resize the entire hash table, its amount of write bytes per operation in insert-only workload is only 61.7% of DASH, which are 231.3/375.0 bytes per write, respectively. It shows that R^{2}Hash significantly reduces the redundant overhead during resizing by using the split-order mechanism.

Exp#4 Resizing Time. We implemented the resizing mechanism of R^{2}Hash based on PCLHT. Both mechanisms consist of two phases for a resize operation: creating a new hash table and rehashing the items from the original hash table to the new hash table. It is noticeable that the GC phase in PCLHT is not in the critical path, so it will not block other threads. As shown in Figure 12, R^{2}Hash only takes 14.9s for total insertion time, while PCLHT needs 50.4 s to finish insertion. This is because R^{2}Hash has reduced by almost 90% of rehashing time. The creation time drops from 5.3s to 1.3s, benefiting from the reusing of the buckets of R^{2}Hash. The split-order and cooperative rehashing mechanism significantly reduces the rehash time from 18.0s to 1.7s. Furthermore, by reusing old hash tables, R^{2}Hash almost does not need much GC operation.

Exp#5 Non-blocking Recovery. All metadata will be restored after the restart. R^{2}Hash has introduced a no-blocking rebuilding mechanism to hide the recovery overhead. As shown in Figure 13, the tests use 200M keys and 1/18/36/54 cooperative threads, respectively. The simulation injects a crash at 1.2 sec and then restarts the R^{2}Hash. After restarting, R^{2}Hash can process requests immediately. The recovery of metadata can be delayed until the corresponding buckets are accessed. Evaluations have shown that the more threads there are, the faster the recovery process finishes.

TABLE II: The Memory Usages. Write traffic/speed to PM (MB, MB/s). Space consumption for DRAM and PM (MB). Util is defined as the valid key-value pair's size divided by the entire PM usage.

	R^{2}Hash	PCLHT	DASH	CCEH	CLEVEL	LEVEL
DRAM Usage	**451**	54	58	58	53	48
Write Speed	**1,931**	883	1,851	1,497	1,478	616
Write Traffic	**28,739**	43,934	46,591	44,180	187,628	39,754
PM Usage	**3,478**	3,376	4,262	4,492	6,656	6,912
Util	**57%**	58%	46%	44%	29%	28%

D. Sensitivity Parameter Test

Exp#6 DRAM Usage. As shown in Table II, we can conclude that R^{2}Hash has the highest write speed, 1.89GB/s to PM, which is slightly better than that of DASH. The write traffic is defined as the total write bytes to the PM during the workload. Notably, the R^{2}Hash has lower write traffic than all other hashings, reducing it by 61.7/65.1/15.3/72.3% respectively. R^{2}Hash uses 451.2MB of DRAM to accelerate find operations. Moreover, it utilizes PM more efficiently, resulting in an average reduction of more than 1000MB of PM space consumption compared to other persistent hashing indexes (except for PCLHT). Although R^{2}Hash has slightly higher PM usage than PCLHT, its bucket pre-splitting mechanism provides significant advantages in terms of write traffic and PM consumption.

Exp#7 Resizing Thresholds and Degrees. As shown in Figure 14, we can find that when the number of resizing degrees, denoted as the cooperative threads increases from 1 to 8, the writing performance increases sharply, then slowly increases until 24 threads. This is due to the low random write concurrency of PM. The resize threshold shows a similar pattern, increasing from the number of 1 to 4 and then decreasing at the number of 6. It is because larger thresholds can delay resizing but also reduce read performance.

Exp#8 Load Factor. Load factor is a key metric for hash tables, and it is defined as the percentage of valid key-value pairs [18]. We cannot blindly try to improve performance at the cost of maintaining a large amount of unutilized memory. Wildly fluctuating load factors can also lead to jittery read and write performance. Because usually, a higher load factor means lower read performance and better write performance and vice versa. R^{2}Hash has a nearly constant load factor (59%∼66%) regardless of whether it finished resizing in Figure 15. This is because R^{2}Hash has a split-order mechanism.

E. Real-world Application Benchmarks

Exp#9 Persistent KV stores using a hashing index. Some persistent KV stores directly place the persistent hashing index structure in DRAM, such as Halo [17] using a variant of PCLHT and Viper [16] with CCEH. They cache data in DRAM to exceed 256 bytes and then refresh them into PM. Our R^{2}Hash is orthogonal to these research works, which can be directly applied to the index structure in Halo and Viper. As illustrated in Table III, R^{2}Hash exhibited the lowest DRAM usage while delivering near-optimal read performance among all the tested persistent hash indexes.

TABLE III: Comparison with other persistent key-value stores using a hash table. R^{2}Hash-H means that R^{2}Hash is used as Halo's volatile index structure, with its write buffer and log mechanisms.

Benchmarks (MB/s)	YCSB Workloads				DRAM (MB)
	a	b	c	d	
RRHash	9.7	18.9	95.2	96.7	**450**
RRHash-H	**37.2**	**69.1**	**97.2**	**98.8**	6700
Halo	30.4	62.2	93.5	93.7	7400
Viper	24.3	54.9	65	43.8	8100

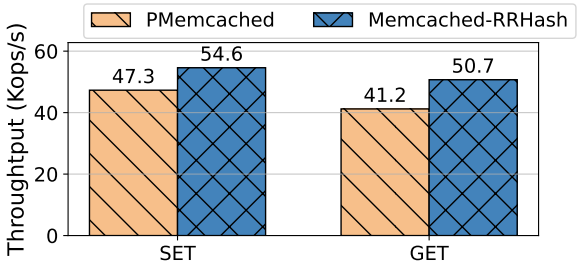


Fig. 16: The Persistent Memcached.

Exp#10 Persistent Memcached. We adopt the real-world caching system Memcached to PM, called the PMemcached. The raw hash index in Memcached is similar to PCLHT. We also implement the Memcached_RRHASH, to show the efficient index performance of R^{2}Hash in Memcached. In the popular test tools *memtier_benchmark*, applying R^{2}Hash in Memcached results in a 1.15x/1.23 improvement in set/get operations, compared to the raw hash table in memcached as shown in Figure 16.

V. Related works

Persistent tree indexes can support fast-range queries but have limited point query performance. ROART [19] proposes leaf compaction to improve range query performance and uses entry compaction to minimize persistence overhead. It hides long recovery time through fast memory management. PACTree [20] is a high-performance persistent index designed by the Packed Asynchronous Concurrency. It uses the radix tree as the internal nodes, and B-Tree nodes as the leaf nodes. The heterogeneous nodes allow it to prevent blocking concurrent accesses by updating internal nodes asynchronously.

VI. Conclusion

We propose a read-optimized and resize-friendly hashing index called R^{2}Hash, enabling a crash-consistent guarantee and fast recovery. R^{2}Hash ensures that the wait-free read operation within three retries only needs one PM access on average, and minimizes resizing overhead with the split order mechanism based on the migration rule. R^{2}Hash also provides a non-blocking recovery mechanism by lazy rebuilding. Evaluations show that R^{2}Hash achieves up to 8.1x/7.5x higher throughput than other persistent hashing indexes. R^{2}Hash also reduced the resizing time by up to 86.2%.

References

[1] "Compute Express Link™: The Breakthrough CPU-to-Device Interconnect." [Online]. Available: https://www.computeexpresslink.org

[2] V. J. Marathe, "Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory," ser. HotStorage'17, USA, Jul. 2017.

[3] "pmem/pmem-redis: A version of Redis that uses persistent memory." [Online]. Available: https://github.com/pmem/pmem-redis

[4] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," in OSDI'20, 2020, pp. 191–208.

[5] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in OSDI'20, ser. OSDI'18. USA: USENIX Association, Oct. 2018, pp. 461–476.

[6] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: converting concurrent DRAM indexes to persistent-memory indexes," in SOSO'19, ser. SOSP '19, New York, NY, USA, Oct. 2019, pp. 462–477.

[7] M. Nam and et al., "Write-optimized dynamic hashing for persistent memory," in FAST'19, ser. FAST'19, USA, Feb. 2019, pp. 31–44.

[8] Z. Chen, Y. Hua, L. Ding, B. Ding, P. Zuo, and X. Liu, "Lock-Free High-Performance Hashing for Persistent Memory via PM-Aware Holistic Optimization," ACM TACO., Aug. 2022.

[9] B. Lu and et al, "Dash: scalable hashing on persistent memory," Proc. VLDB Endow., vol. 13, no. 10, pp. 1147–1161, Apr. 2020.

[10] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen, "Persistent memory hash indexes: an experimental evaluation," Proc. VLDB Endow., vol. 14, no. 5, pp. 785–798, Jan. 2021.

[11] "[Webinar] Memory-Semantic SSD™," Dec. 2024. [Online]. Available: https://semiconductor.samsung.com/us/news-events/tech-blog/webinar-memory-semantic-ssd

[12] C. Wang, J. Hu, T.-Y. Yang, Y. Liang, and M.-C. Yang, "{SEPH}: Scalable, Efficient, and Predictable Hashing on Persistent Memory," 2023, pp. 479–495, gSCC: 0000003. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/wang-chao

[13] Z. Liu and S. Chen, "Pea Hash: A Performant Extendible Adaptive Hashing Index," Proceedings of the ACM on Management of Data, vol. 1, no. 1, pp. 108:1–108:25, 2023, gSCC: 0000005 1 citations (Crossref) [2024-03-18]. [Online]. Available: https://dl.acm.org/doi/10.1145/3588962

[14] P. Zuo and Y. Hua, "A Write-Friendly and Cache-Optimized Hashing Scheme for Non-Volatile Memory Systems," TPDS, vol. 29, no. 5, pp. 985–998, May 2018, conference Name: IEEE Transactions on Parallel and Distributed Systems.

[15] Z. Chen, Y. Huang, B. Ding, and P. Zuo, "Lock-free Concurrent Level Hashing for Persistent Memory," 2020, pp. 799–812. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/chen

[16] L. Benson, H. Makait, and T. Rabl, "Viper: an efficient hybrid PMem-DRAM key-value store," Proc. VLDB Endow., vol. 14, no. 9, pp. 1544–1556, May 2021.

[17] D. Hu, Z. Chen, W. Che, J. Sun, and H. Chen, "Halo: A Hybrid PMem-DRAM Persistent Hash Index with Fast Recovery," in SIGMOD'22, ser. SIGMOD '22, New York, NY, USA, Jun. 2022, pp. 1049–1063.

[18] K. Huang and T. Wang, Indexing on non-volatile memory: techniques, lessons learned and outlook, ser. SpringerBriefs in Computer Science. Cham: Springer Nature Switzerland, 2024. [Online]. Available: https://link.springer.com/10.1007/978-3-031-47627-3

[19] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu, "{ROART}: Range-query Optimized Persistent {ART}," 2021, pp. 1–16, gSCC: 0000074. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/ma

[20] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, "PACTree: A High Performance Persistent Range Index Using PAC Guidelines," in SOSP'21, ser. SOSP '21, New York, NY, USA, 2021, pp. 424–439, gSCC: 0000074.