Ph.D. DISSERTATION

Efficient and Scalable Hashing Scheme for Persistent Memory

영구 메모리를 위한 효율적이고 확장 가능한 해싱 체계

February 2023

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Dereje Regassa

Ph.D. DISSERTATION

Efficient and Scalable Hashing Scheme for Persistent Memory

영구 메모리를 위한 효율적이고 확장 가능한 해싱 체계

February 2023

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Dereje Regassa

Efficient and Scalable Hashing Scheme for Persistent Memory

영구 메모리를 위한 효율적이고 확장 가능한 해싱 체계

지도교수 염헌영

이 논문을 공학박사 학위논문으로 제출함

2022 년 11 월

서울대학교 대학원

컴퓨터 공학부

에데싸 데레제 레가사

에데싸 데레제 레가사의 공학박사 학위논문을 인준함

2022 년 12 월

위 원 장	엄 현 상	(인)
부위원장	염 헌 영	(인)
위 원	유 승 주	(인)
위 원	이 재 욱	(인)
위 원	손 용 석	(인)

Abstract

The new advances in memory have brought many potential innovations in the data structure. The byte-addressable Persistent Memory(PM) with high capacity and low latency accelerated the shift of most existing hashing-based indexes to exploit these benefits. Hence, many new hashing schemes have been proposed using emulators which are found to be sub-optimal and also not scalable on the real device. Few hash table designs addressed important properties like load factor, scalability, efficient memory utilization, and recovery. One of the challenges in redesigning data structures for an effective hashing scheme in PM is to reduce the overheads of dynamic hashing operations in the hash table. In this paper, we present an Efficient and Scalable hashing scheme called ESH that improves memory efficiency, scalability, and performance on PM. ESH enables us to efficiently use the available spaces in the hash table and delays the full table rehashing to improve performance. This makes ESH achieve maximum load factor with efficient allocated memory space utilization. We evaluate our scheme and compare it with the widely used state-of-the-art dynamic hashing schemes that apply similar hashing techniques on Intel Optane® DC Persistent Memory (DCPMM). The experiment result shows ESH improves data insertion performance by 30% and 4% compared to CCEH and Dash respectively. It also improves the lookup operation by nearly 10% compared to Dash and achieves up to 91% load factor which is higher compared to the other competitors.

Keywords: Persistent memory, Dynamic hashing, Scalable hashing, In-memory systems, Extendible hashing

Student Number: 2018-31651

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The fast growth of data centers is frequently pushing hardware utilization to its limits. This development is forcing innovations to redefine the existing hardware. The recent emergence of persistent memory devices like Intel Optane DC Persistent Memory Module(DCPMM) that delivers the persistence fast speed and high capacity that solves the limits in data access speed and storage capacity [1,2]. They provide large storage capacity and offer a competing performance between DRAM and flash and are resilient to crashes. The access latency close to DRAM, durability, and byte addressability of this new hardware makes it suitable for latency-critical transactions on storage systems.

Moreover, the new features of this device brought changes in the way data can be persisted through direct accesses by load and store instructions. There is good progress on single-level persistent-based applications that directly operate and store data on PM without the involvement of the storage stack in OLTP [3,4]. However, the data consistency and hardware limitations are challenges for prior applications that use indexing techniques as they are originally designed

for DRAM environments. Hence, different persistent indexing techniques are designed and chosen to efficiently store data in persistence memory.

Relatively there are a significant number of researches to improve Tree based indexing structures for persistence memory compared to the attempts for hash-based indexing structures. However, the existing indexes cannot provide the same advantages on PMs as it suffers from asymmetric read/write performance and crash consistency problems. Using the existing indexing structures to run on PM without modification to fit the new architecture would not help to gain the expected performance. Thus, we need to redesign index structures for PM. High-performance and scalable indexing structures are crucial for storage systems to achieve fast queries.

Hashing-based indexing structures are widely used in different applications using key-value stores [5,6]. Several PM indexes such as FAST & FAIR [7], NV-Tree [8], WORT [9] and CCEH [10] are designed to provide low overhead, and cost-efficient indexing operations ensuring the indexes recover correctly in the event of failure or crash. Additionally, numerous proposals for index redesigning [3, 4, 11–13] adopted different indexing structures for persistent memory which mainly uses emulators.

With the arrival of persistence memory, a variety of tree-based techniques are designed to optimize indexing for tree variants and hashing. These results are mostly designed using B^+ tree-based indexes [7, 13] and [10, 14] mainly uses hash-based indexes. There are also significant research results for hash tables to provide faster operations in in-memory systems indexing structures. The efficient lookup time for mapping values with a particular key in hash-based indexes needs to achieve faster access respective of the size of the data. In this hashing scheme, the allocation of sufficient bucket size for the hash function plays a significant role in determining the buffer cache for the hash

table. In applications like a key-value store, the sizes of records can not be predicted as it involves the dynamic insertion or deletion of items. Therefore, a dynamic hashing scheme that involves dynamic resizing is the best choice to adjust the table size to fit records to the scheme. PM's read/write latency during searching a record in a large portion of storage can introduce cache misses [14] and PM accesses. Concurrency controlling during these operations needs careful attention as it introduces additional read/writes for locking that ends up in further bandwidth consumption. As the data insertion increases, the load factor of the hash table becomes high forcing the growth of the hash table. To accommodate that, the hash table will be rehashed and the data will be relocated to the newly created buckets. Rehashing is an expensive operation that involves doubling the buckets as it is an incremental operation and hence the new index at which the values have to be mapped to a new location. Moving the existing records to a new bucket location degrades the throughput halting the access of the indexes during rehashing. On top of the read/write latency of PM, it increases the total query latency.

In this dissertation, we design a scalable hashing scheme that efficiently stores key values on persistent memory. It employs the mechanisms that effectively utilize the hash table by tuning the directory entries and the segment to accommodate more records to available space in the existing buckets by extensively utilizing the segment before initiating the expensive full table rehashing. We introduce an Efficient and Salable Hashing Scheme called ESH that has the following properties. (i) efficiently use the available space in the hash table by moving the overflow records from a bucket to the neighboring buckets in the segment. (ii) reduce or delay the full table rehashing to gain better performance during insertion. (iii) increase the load factor and PM/Memory utilization efficiency without significant performance loss for varying data size and the number

of threads. (iv) present a scalable hashing scheme that reduces unnecessary PM read and write operations that saves the PM bandwidth and scales well in a multi-threaded environment. We evaluate our scheme and compare it with the widely used state-of-the-art dynamic hashing schemes that apply similar hashing techniques on Optane persistent memory with different configurations. The experiment result shows ESH improves the data insertion performance by 30% compared to CCEH and by 4% compared to Dash. It also improves the search operation by nearly 10% compared to Dash. It also achieves up to 91% load factor which is higher compared to the other competitors.

1.1 Motivation

From the recent trends, the rapid development of PM technologies has significantly affected the design of today's storage systems and also shaped the future of storage technologies [15–17]. Designing high-performance and scalable indexing structures that fit this development and achieve fast queries are also becoming crucial for storage systems. However, existing legacy indexes designed for DRAM or disk cannot fully provide their benefit on PM as it suffers from asymmetric read/write performance and crash consistency issues as witnessed in recent research [10, 14, 18]. Hence, it is time to think about designing an efficient and scalable dynamic hashing scheme for persistent memory. Thus, this work is motivated to improve extendible hashing and increase space utilization of this scheme on emerging persistent memories.

1.2 Contribution

The contributions are summarized as follows:

- We present a hashing scheme that improve hashing for emerging persistent memories to efficiently store a key-value pairs on persistent memory. This mainly improves hashing performance and increase space utilization. In our scheme, we delay expensive full table rehashing operations by using the free space in other buckets.

- To do that we introduce an approach that, when the existing bucket is full and requires a rehashing operation, ESH searches the free space in the neighboring buckets in the same segment and allocates the space for the requested record.

- Store a metadata of the bucket to manage the moving records in other buckets so that it can find out the moving records faster. By doing so, it allows to delaying the expensive full table reshaping operation during the insertion operation.

- We implement and evaluate our schemes on a real Intel Optane Persistent memory based system. The evaluation result shows that our proposed scheme can efficiently store key-values pairs efficiently while saving space and also improves the load factor on the real device. We evaluated with various configurations and the results show that ESH can improve the insert and search performance compared with the state-of-the-art hashing schemes.

1.3 Outline

This dissertation is structured as follows:

- **Chapter 2** covers the background about persistent memory, Optane architectures and Instruction supports, Dynamic hashing schemes, effects of

NUMA awareness, inter-thread interference, and the impacts of locking and frequent key resizing on performances.

- **Chapter 3** introduces Adaptive Cache-Conscious extendible hashing, our cache-conscious scheme that increases the efficiency of memory utilization for persistent memory. We first explain the problems of existing persistent memory-based extendible hashing schemes which are cache oriented. Then we describe the details of the design and implementation of our proposed new scheme and evaluate it on the real Optane persistent memory with YCSB benchmark and real-world workload and compare it with available state-of-the-art schemes.

- **Chapter 4** introduces ESH, our Efficient and Scalable hashing scheme that extends the memory-efficient scheme that scales well as the number of threads increases. We start by explaining the effects of sub-optimal space utilization of the existing schemes. Then we propose the details of our scheme. Then we propose the details of our scheme. Finally, we evaluate our scheme on a real Optane Persistent memory against the state-of-the-art schemes.

- **Chapter 5** summarizes and concludes the dissertation. It also points out the directions for future work.

Chapter 2

Background

Many hashing schemes have been proposed for DRAM-based applications. Those schemes will not fit persistent memory as PM has different architectures. Some schemes are designed using emulators which are found to be sub-optimal and others are not scalable on the real device. Few hash table designs addressed the important properties of hashing schemes for PM like load factor, scalability, efficient memory utilization, and recovery. One of the challenges in redesigning data structures for an effective hashing scheme in PM is to reduce the overheads of dynamic hashing operations in the hash table.

2.1 Optane Persistent Memory

Today, Persistent Memory is used in different applications like databases, storage, cloud computing/IoT, and artificial intelligence as it enabled fundamental change in computing architecture. In this regard, Intel's DC Persistent Memory (PM) redefines the traditional memory architecture offering a much higher

capacity that is larger than DRAM at an affordable price. It provides salient features of large capacity, low latency, and real-time crash recovery for storage while directly populated and accessed through the existing memory bus. As it also uses the CPU load and stores instructions thus avoiding the high overheads of conventional interfaces. The increased capacity and enhanced security with hardware-level encryption also solve the greatest business challenges. Persistent memory (PM) also provides both high performance and capacity by utilizing non-volatile memory [19]. Since the volatile memory capacity is exhausted, the non-volatile persistent memory became an alternative to DRAM [20]. As traditional hashing techniques designed for DRAM are inefficient in persistent memory, we designed an efficient hashing scheme that uses the features of persistent memory.

2.2 Optane Architecture and Instructions support

To ensure persistence, the integrated memory controller(iMC) sits within the asynchronous DRAM refresh (ADR) domain, Intel's ADR feature ensures that CPU stores that reach the ADR domain will survive a power failure [21]. The iMC maintains read and write pending queues (RPQs and WPQs) for each of the Optane DIMMs as seen in (Figure 2.1a) thus, once data reaches the WPQs, the ADR ensures that it will survive power loss and the actual access to the storage media happens after the address translation. The Optane controller translates a smaller request into 256 bytes access of the Optane block that causes the write amplification. Applications and file systems modify the Optane DIMMs contents using the store instructions that eventually make applications persistent. Intel ISA provides the *clflush* and *clflushopt* instructions to flush cache lines back to memory, and *clwb* can write back the cache lines. To write

instructions with no cache, applications also use the *ntstore* to write directly to memory. Thus, applications must issue *sfence* to ensure flushing that persisting data.

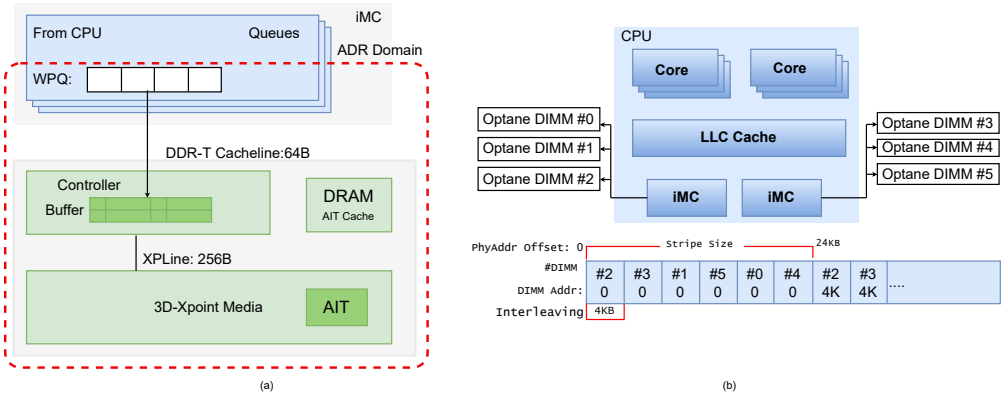


Figure 2.1 Optane Architecture (a) Overview of Optane DIMMs and (b) Optane DIMMs interleaving strategies

Building much faster data structures for persistent memory requires changes in programming that make it complicated and prone to error [22]. However, data should be consistently stored in the order of the store. Typical standard approaches are followed to store data consistently. Therefore, a write-optimized and scalable hash table designed for persistent memory constitutes these features to gain the expected performance.

2.3 Dynamic Hashing

Hashing has been used in data storage in memory for a long. Dynamic hashing is usually used to shorten the string characters. This scheme works by growing, shrinking, and reorganizing the characters to fit the way data is being accessed making it faster and easier for data storage.

Extendible Hashing – a dynamic hashing that does rehash operation incrementally to help applications less affected by hash table growth compared to the standard full-table rehashing. As the in-memory data size increases, rehashing the traditional hash tables incurs higher latency requiring better rehashing techniques for persistent memory. To ease the overhead during rehashing, linear probing, separate chaining, cuckoo hashing, CCEH [10] and Dash [14] are a few of the techniques used. CCEH - the variant of extendible hashing designed to optimize access of hash table buckets to cache-line access that significantly minimizes the number of cache-line accesses. It also reduced the overhead directory operations as it groups a number of buckets into intermediate sizes called segments. This approach helps to reduce the size of the directory to cache-line-sized buckets and also reduces rehashing management during failure recovery.

A dynamic hashing scheme called extendible hashing utilizes a bucket address table called a directory where indexes are used to locate exact matching queries to find a record with a given key. The hashing function is flexible and dynamically changed to effectively manage directories and buckets to hash data. Buckets that store records are pointed to by the global directory which stores the bits that determines the directory entry. Directory entries are shown in binary of the hash value and point to buckets as shown in Figure 2.2 and a bucket stores key-value stores.

In extendible hashing schemes, the maximum number of buckets is denoted as 2^{G} where G is global depth. Look up and update operations following the corresponding directory entry pointer to get to the buckets. Accordingly, a hash table can have at most 2^{G} directory entries to the buckets. Buckets are also traced using the local depth (L) and when a bucket is full and it helps to compare and initiate the bucket splitting or directory doubling.

As shown in Figure 2.2-(a), if a record (i.e., 63) is inserted into the hash

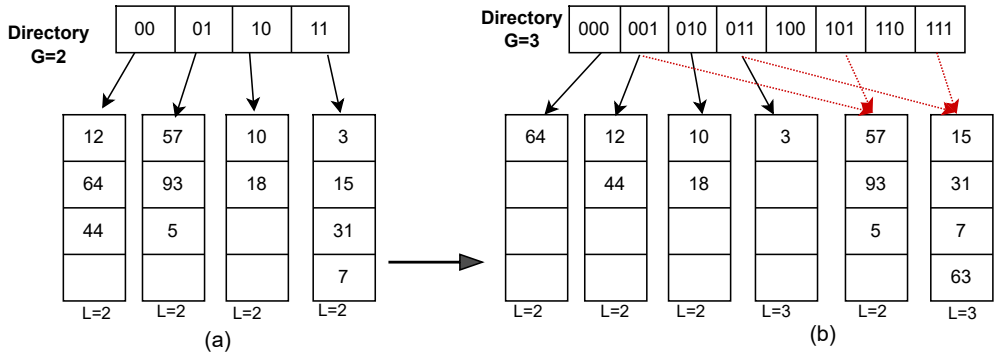


Figure 2.2 Extendible hashing and directory doubling (a) is before directory doubling and (b) shows directory entry and the local depth of each bucket increased after doubling (G: global depth, L: local depth)

table and if there is no space in the last bucket in the hash table, directory doubling is initiated as the local depth is equal to the global depth (L=G). In Figure 2.2-(b), the directory entry is doubled, the local depth of the overflown bucket is increased and the records are rehashed and inserted into the new hash table.

If multiple directory entries point to a bucket and the local depth is less than the global depth, it indicates the overflow. If the bucket overflow happens, bucket splitting will be initiated and buckets will be split into two. This operation depends on whether the local depth of the overflown bucket is equal to the global depth before the split. This process may or may not involve directory doubling.

During splitting, the local depth for the resulting buckets will be incremented, if it is equal to the global depth, overflow happens resulting in the directory doubling. Directory doubling occurs when G bits are not enough to distinguish the search value of the overflow bucket.

Extendible Hashing on PM – There are prior works that used extendible hashing to be adapted to work on PM [10, 14]. Many efforts are done in these works to reduce the PM access as a function of speeding up the hashing scheme. Grouping buckets together in a segment to use a single directory entry to these buckets helped to slow down the directory growth in [10]. Dash [14] attempted to reduce unnecessary read and write operations to conserve bandwidth and alleviate the impact of high end-to-end read latency.

2.4 Effect of NUMA access

Main memories today are composed of DRAM chips packaged in dual inline memory modules (DIMMs), with several DIMMs making up the total main memory of the system. The ever-growing level of parallelism within the multi-core and multi-processor nodes in clusters leads to the generalization of distributed memory banks and busses with nonuniform access costs.

Non-uniform access (NUMA) allocates separate memory banks to each processor by splitting memory and CPUs across different nodes. This avoids performance hits when several processors/cores attempt to address the same memory. processors on the same nodes access their own local memory and remote memories on the other nodes. Accessing the local memory faster than accessing the remote ones contributes to the increase of overall performance.

Most high-performance servers today are NUMA machines that have complex memory hierarchies. Computer Architects adopted NUMA to accommodate many cores in a single computer where these cores are clustered into nodes and each node shares the last level cache(LLC) and memory. Traditional assumptions on memory such as access time, and memory stall can not hold as

these machines have longer memory stalls and inaccurate data structures to design that help to gain good performance [23].

Hence, to get high performance we have to design indexing structures that help to gain the expected benefit from NUMA. These designs should take into consideration that they should be efficient for time complexity, less synchronization overhead, and cache efficient memory access pattern. The latency and bandwidth of NUMA depend on the node accessing the data and the node where the data is stored. NUMA optimizations have been explored extensively and focus on adjusting the thread and data placement across the nodes to minimize latency and maximize bandwidth [24–28]. Hardware and software design that uses the memory system architecture affects the performance. The cores in the multi-core processors share on-chip memory systems resources like memory controllers, last-level caches, or pre-fetcher units. If there is a contention on using these resources, it leads to performance degradation [29, 30].

Operating system scheduler is in a good position to reduce the shared last-level cache contention. Data locality-related problems are also addressed either by profile-based or dynamic memory migration [29]. Poor performance in accessing non-local memory (NUMA effects) can significantly impact non-volatile file system performance. Introducing NUMA-aware interfaces to the non-volatile memory module file systems can relieve this problem [31]. There are many active types of research on increasing NUMA locality. some propose the redesign of data structures with NUMA awareness that help to fully exploit the structure's internal features [32–35].

Efficient allocation of memory in NUMA systems is critical to maximizing the performance of the system. Additionally, choosing efficient NUMA-aware file systems affects overall performance [36]. With the release of Optane DIMMs, many researchers are working on redesigning storage systems to gain full of

their potential [12, 37]. As seen in Figure 2.3, the design of the NUMA-aware approach improves the performance. From the figure, if data is freely stored on either near or far PM memory, it will be affected by the data locality problem. If data is stored in the local PM by all nodes, the performance is improved. As seen in the figure, we can observe the contribution of NUMA even though the gain is not significant. For a 30GB data insertion, we can observe the performance contribution of NUMA up to 29sec. Hence, we consider NUMA aware approach for ESH.

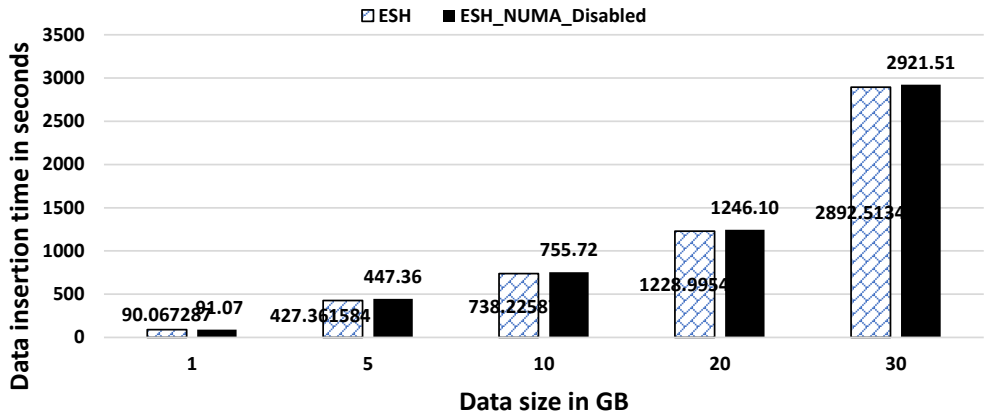


Figure 2.3 The NUMA effect on record insertion performance

2.5 Inter-thread interference

Multi-threaded concurrency is considered one of the key features that many researchers are working on to gain high performance out of multi-core in main-memory database systems. To efficiently gain concurrency at its maximum on modern CPUs, the implementation of latch-free(lock-free) indexing structures that avoid bottlenecks is used. Applications like MemSQL uses lock-free skip-lists [38] whereas others like Microsoft's Hekaton main-memory OLTP engine

use B+Tree lock-free [39] applications.

The design of lock-free algorithms index designs are often complex as they rely on atomic CPU hardware primitives such as compare-and-swap (CAS) to atomically modify the index state. These atomic instructions are single-word instructions whereas extendible hashing schemes usually require multiple-word updates during directory doubling and segment splitting as these are critical operations and multiple steps which are broken into multiple steps that expose the operations to other threads. This problem is even more complicated in multicore systems and may result in race conditions if intermediate states are exposed to other threads.

2.6 Locking

Crash consistency and write optimizations are mainly considered for optimization of the existing hashing-based indexes for PM where there are fewer efforts on the effect of blocking during hashing [10–12]. Locking an entire hash table in which the thread holding the lock will prevent any other threads from accessing the hash table is not a good idea for time-sensitive applications. Therefore, a carefully designed fine-grained locking scheme that protects limited buckets or single buckets is preferable.

Hence, ESH uses optimistic locking that employs locks at the bucket level. This minimizes the number of table blocks that will be blocked from other insert processes. The lookup operations in this scheme are lock-free and multiple processes can access the same record in the bucket.

2.7 Impact of frequent key resizing

A hash table is commonly used to store key values that associate keys with values to implement dictionaries or to test if a key is part of a set of keys to implement set operations. Hash table resizing is inevitably triggered when the initial capacity of the hash table is less than the total size of items to be inserted. Thus, the resizing overhead has an impact on inert throughput. Performing these operations at the lowest cost determines the efficiency of the hash tables.

If we have a small bucket size, the hash table is filled and therefore, the hash function is forced to initiate the rehashing operation that affects the performance. Allocating a relatively larger hash table will be advantageous as it helps to delay the resizing operation. In our scheme, an intermediate resizing during segment split to expand space before a full table rehashing happens. Segment splitting incurs much less PM access compared to the whole table rehashing.

Accordingly, ESH uses 256bytes for a bucket that can store more records once the hash function creates the hash table. Once this bucket is full, the segment rehashing will redistribute the key value to the buckets after the segment splits. This feature and having a larger bucket size helps to reduce the frequency of key resizing contributing to get better performance.

Chapter 3

Adaptive Cache-Conscious Extendable Hashing

3.1 Motivation

A non-volatile persistent memory (NVM) provides low latency close to DRAM, durability, and byte addressability [11, 17, 40]. These new features brought changes to the way data can be persisted via direct accesses using load and store instructions [41]. Thus, to maximize the benefits, data structures need to be re-designed to efficiently store data in persistent memory. Hence, We design and implement a new scheme that increases the memory utilization in hash tables for the cache-conscious extendible scheme.

A faster and easier way of finding the records in a hash table needs efficient indexing structures. Dynamic hashing is preferred over static one as it helps to expand and shrink the size of the table based on the data size. Persistent memory-based dynamic hashing structures that address failure atomic constraints while achieving efficient dynamic hashing are the hot research areas.

Extendible hashing is a dynamic hashing scheme that utilizes a bucket address table called a directory where indexes are used to locate exact matching queries to find a record with a given key. The hashing function is flexible and dynamically changed to effectively manage directories and buckets to hash data. Thus, it is more suitable for time-sensitive applications. Buckets that store records are pointed to by the global directory which stores the bits that determines the directory entry.

A directory can be the form 2^G where G is a global depth of the directory and to locate a key k, the hashing function $h(k)$ uses the last G bits to choose directory entry. Multiple directory entries can point to the same bucket when the bucket fills up. And, it initiates the split which requires more directory entries where every bucket has a local depth L which indicates the length of the commonly used hash key of the bucket.

The difference between local depth and global depth will be used to control bucket overflow indicating. If multiple directory entries point to a bucket and the local depth is less than the global depth, it indicates the overflow. If the bucket overflow happens, bucket splitting will be initiated and buckets will be split into two. This operation depends on whether the local depth of the overflown bucket is equal to the global depth before the split. This process may or may not involve directory doubling. During splitting, the local depth for the resulting buckets will be incremented. And, if it is equal to the global depth, overflow happens resulting in the directory doubling. Directory doubling occurs when G bits are not enough to distinguish the search value of the overflow bucket.

In this section, we present an Adaptive Cache Conscious Extendable Hashing (ACCEH) to increase the utilization of persistent memory. Thus, to maximize the benefits, data structures need to be re-designed to efficiently store data

in persistent memory. Hence, we extend the Cache-Conscious Extendible Hashing (CCEH) scheme that carefully manages the buckets, stores records, and delays directory operations in the hash table to increase memory utilization. In this section, we designed a scheme that achieves a cache-conscious approach to accommodate more records to the existing buckets by fully utilizing the free spaces of the buckets in a segment. Additionally, it can delay the directory doubling operation during record insertion so that the overheads of directory doubling can be reduced.

The implementation and evaluation of ACCEH are conducted on a 32-core machine with Intel Optane DCPMM and compare ACCEH with a state-of-the-art scheme (i.e., CCEH). Hence, this scheme (i.e., ACCEH) improves the performance compared with a state-of-the-art scheme in uniform and skewed data distributions.

3.2 Related Work

To fully utilize the data structures in persistent memory, there are many studies proposed to present new data structures for persistent memory. From many, we can see FAST & FAIR [7], NV-Tree [8], WOART [9], and CCEH [10] which presented efficient data structures to fully exploit the performance of persistent memory. Additionally, other studies [3,4,11–13] also designed indexing schemes for persistent memory. CCEH [10] stands out as a cache-conscious extendible hashing scheme that optimizes a hash table that significantly minimizes the number of cache line accesses. It reduces the overhead of directory operations by grouping several buckets into intermediate sizes called segments.

Different hashing schemes are designed to work for persistent memory to solve some challenges related to consistency issues. A cuckoo-based hashing

scheme called PFHT [42] designed for phase-change memory (PCM) uses a stash that stores any overflow entries for full-table rehashing and gains performance on load factor. Even though this approach improves the insertion performance, it is not a cache-friendly structure and takes a long time during searching for a key in a bucket leading to higher lookup costs. Similarly, path hashing [43] which is a cost-efficient write-friendly hashing scheme using an inverted binary tree reduces the lookup cost. Other hashing schemes that perform a full table rehashing level by level are designed to enable constant scale operations [44]. But this scheme is not efficient in controlling the hash table size which will make the displaced records be more than one cache line during bucket updates.

Guaranteeing failure atomic write or update with a reduced number of cache line access has shown good performance [10]. It is considered to write optimal, as it uses a cache line size bucket even though one cache line cannot hold more than four key-value pairs. In order to control the growth of directory size, it is considered that increasing the bucket size is not a good idea as the size of the bucket cannot be more than the cache line. Additionally, there is a trade-off between the large bucket size and lookup performance as it suffers from access to buckets across multiple cache lines. To balance the directory size and access performance, it uses an intermediate layer called a segment between the directory and buckets.

Segments group the buckets and an 8-byte directory entry is used to point to them for accessing the buckets. This reduces the number of directory entries required for buckets as a single directory entry points to a group of buckets in the segments. Each directory entry points to segments containing a fixed number of buckets. As shown in Figure 4.2, a segment stores a fixed number of buckets, and each directory entry points to segments containing buckets. Hence, the directory entry 00_2 indicates the global directory depth G and the

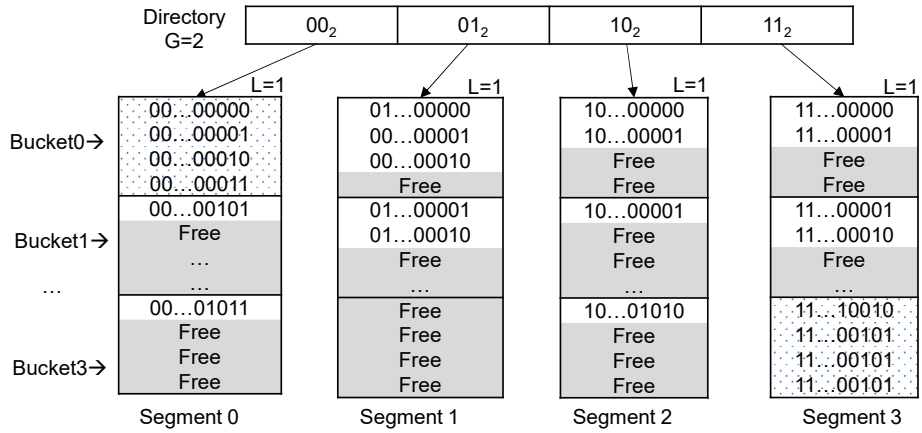


Figure 3.1 Cache conscious extendible hashing

local depth of the segments L in the hashing scheme.

The directory can become significantly smaller as few bits are needed to address segments which makes it cache-conscious and helps to reduce the access to the persistent memory. Under this situation, for better use of persistent memory and increasing memory utilization, this paper complements CCEH. For example, we design a strategy that exhausts the buckets in the segments before the split of segments and/or directory doubling happens.

3.3 Design and Implementation

We present an adaptive cache-conscious extendable hashing (ACCEH) to increase the utilization of persistent memory. To do this, our scheme is designed to store more records in the existing buckets by fully utilizing the free spaces of the buckets in a segment. Accordingly, directory doubling will be initiated when there is no available space in all neighboring buckets in that segment. In summary, exhaustively utilizing the buckets in the segment will contribute to

(1) increasing memory utilization by efficiently filling up the buckets in each segment, and (2) reducing the expensive operation (directory doubling) for the large table rehashing.

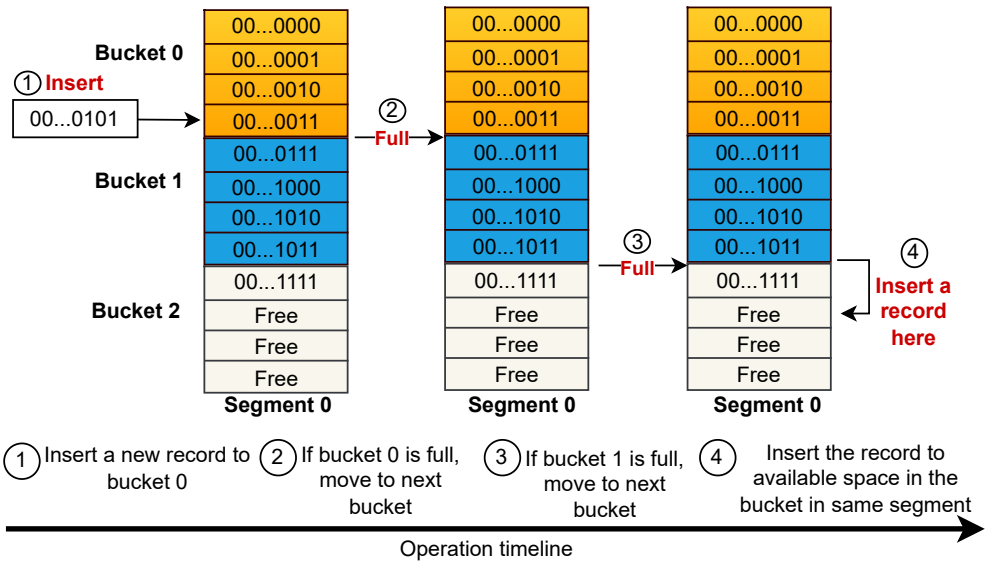


Figure 3.2 Record insertion strategies to a bucket

3.3.1 Design

To get better performance, the segment size and the number of buckets per segment need to be designed carefully. This design option considers the sensitivity of the design options to other operations like insertion, read and search. If the size of the segment is large, it highly hurts the search performance. If we consider the number of buckets in a segment to be 8 or more, we need to have a segment with a larger memory size of up to 2 MB or more. This means, to load one segment, we need to read up to 32 cache lines. This is not cache-friendly and also not aligned with the read block of Optane persistent memory.

ESH uses probing techniques to go through the records in the buckets and

move to the other buckets during overflow. When the number of buckets increases in the segment, the total number of buckets will also increase. This leads to an increase in the number of records in the whole segment. In this situation, a search operation is highly affected as it has to go through all elements to search for the records in a segment. If we consider less number of buckets like 2 in a segment, we store fewer elements in a segment. It will be filled up frequently during insertion operation and cannot help to gain the intended optimal performance. Once it is full, we initiate the full table rehashing more frequently than we are expecting to do that. Hence, it will not be ideal to use the option. In order to strike a balance between the Optane persistent memory read block and cache line reads, it is ideal to set the number of buckets in a segment to four. Most of the other research like Level hashing used four buckets design to get the benefits of small-sized buckets to get better probing. For these reasons, we choose to use the four buckets in one segment design.

Hence, for a hash table that has four buckets stored in a segment and each bucket, has a space that accommodates four records, when a record insertion attempt fails in one bucket, it continues searching for space in other buckets until the last bucket in the segment and then insert a record to one of them. Data is stored in the bucket as arrays of records.

A bucket store records from hashing function and records moved from the neighboring buckets. During the insertion of records, if the bucket or the neighboring buckets has more space, we move the record to these neighboring buckets to delay the doubling operation. This can be used when a given bucket had not had enough space to store a record and there are other neighboring buckets free or have space to accommodate the records.

If there is a bucket with fewer records, we loop in the buckets within the same segment to find the right location to insert the record in neighboring buckets.

This traversal to insert a record cannot go out of the segment as it complicates the lookup. This approach is mainly to effectively utilize the available buckets which are partially filled or not filled at all. For every insertion, it hashes to the right bucket and if it is already full we move to the next buckets with fewer elements and insert using probing.

For the concurrency control, the initial scheme that maintains the lock is used as it shows reasonable performance for extendible hashing and a flag that helps to check if the used bucket is cleaned. When records are inserted into the buckets, if there is no space available for the newly inserted record to any buckets in the same segment, we check for available space to store in neighboring buckets. This can be achieved in a similar way to the state-of-the-art schemes that use the tracker for the global and local depth during splitting and directory doubling.

3.3.2 Implementation

As shown in Figure 3.2 for insertion operation, ① indicates insertion attempt to already full bucket. Here as there is no space in the bucket, as in ② shows a failure of insertion that initiate the bucket split or directory doubling depending on the depth of the segment will proceed. As then the bucket is full, ③ moves to the neighboring bucket in search of available space. If not successful, it advances searching space to the other buckets in the same segment as seen in ④. Once a space is found in one of the buckets, insertion succeeds. The directory doubling happens when the scheme exhausts the space in the segment. The records in the buckets are accessed as arrays and there is a fixed number of elements in each bucket to ease the search operation.

Searching for a value that is moved to another bucket can be accessed by looping through records that belong to the same segment with minimal overhead

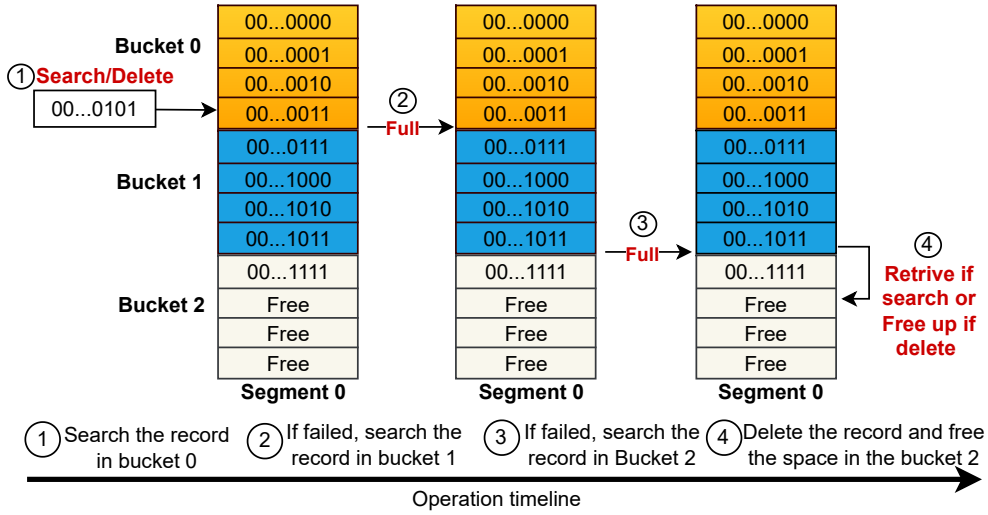


Figure 3.3 Record search/deletion strategies from a bucket

compared to the doubling operations. Figure 3.3 shows this operation procedure where the search always starts from the correct hash value as seen on ①. If searching for the record is not successful in the designated hash location, ② extends the search to the other buckets in the segment and ③ scans all buckets. It retrieves the data if existed or replays it as not existing after this procedure completes scanning the elements in the consecutive buckets. If the operation is to delete the record, based on the operations at ④ it nullifies the record and successfully frees the space in the bucket. The freed space is collected through the garbage collection scheme and later on other records can be assigned based on the hashing operation.

Hashing of records in this approach is relatively similar to hashing of records to a bucket in CCEH to store records. However, when there is a request to insert a record to an already filled bucket while the neighboring buckets have free space, the scheme checks if insertion requires segment splitting. It performs

balanced and exhausted insertion to all buckets in the same segment before rehashing is triggered. If there are neighboring buckets that are either empty or semi-filled, it is less expensive to store records in these buckets than trigger directory doubling. This can be done as records are stored in the buckets and accessed as arrays. Directory doubling and rehashing of records will be initiated when there is no space in all buckets in the same segment.

Deletion of records is done by overwriting during insertion as designed by the initial scheme where invalid values during hashing can be overwritten by replacing them with valid ones. When the record is stored in another bucket as per this scheme, the deletion operation will nullify the record in the bucket and this space is counted as free. Accordingly, if a given record is hashed to bucket *b0* but moved to bucket *b3* because it is already full in this scheme, deletion will look up the record in *b0* and if not found, it continues to *b3* looping through the records in the buckets. Once the record is found in the intermediate buckets or *b3*, it nullifies the record and thus makes the record invalid.

If certain records are deleted from an already filled bucket, the system will move the record from another bucket back to its original hashing bucket to ease the lookup for the next operation if data access is not fast. This operation has less priority as the size of the bucket is the size of the cache line and once all buckets are exhausted, segment split will rehash all records to their proper buckets.

3.3.3 Concurrency

Designing a hashing scheme that scales well on multi-core machines that run on persistent memory is still challenging. ESH aims to tackle this challenge and contribute to a scalable and efficient hashing scheme for persistent memory that guarantees persistence without incurring significant overhead.

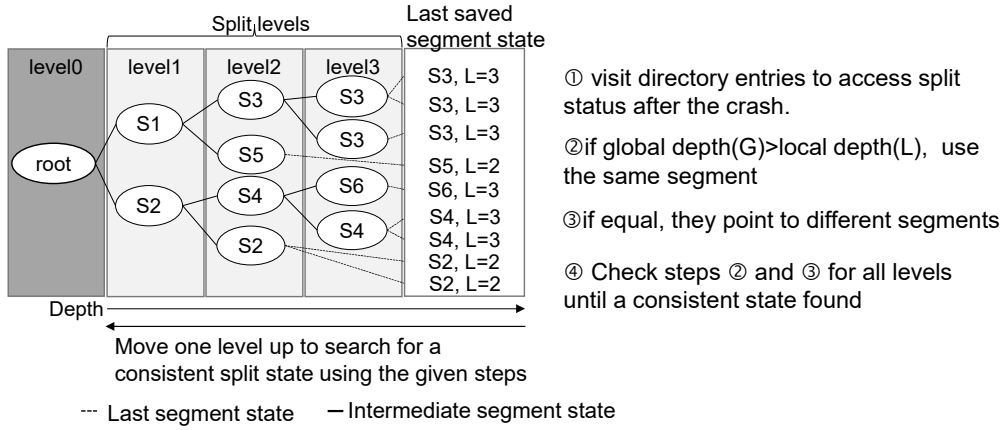


Figure 3.4 Recovery operations: Recovery operation starts from the last saved split state as seen in the steps and moves to the older states until a consistent state is found (S: segment state)

3.3.4 Recovery

Recovery is another fundamental requirement that ensures data consistency (recoverability and correctness) for highly available applications. When a system failure occurs, data in a volatile CPU cache and RAM will be lost but incomplete data modifications may still exist in non-volatile memory like persistent memory that causes inconsistency. Hence, to guarantee data consistency, it is essential to make writes durable in the desired order [9, 13, 45].
The order in which data are stored is valued and then their keys as inspired by the predecessor calling the $mfence$ and $clflush$ instructions to assure durability.

When a system crash happens and recovery is initiated, we traverse the directory entries to check their consistency using the global and local depth indications in a similar approach as the predecessor to access the last consistent state of the hash table. Accordingly, we trace how many times the segment appears contiguously in the directory using the buddy tree. We visit the parent

node after checking the local depth and decrease the local depth by one and again check for consistency at that level. The split history is used as a tracking tool from the current state to the older states and is managed by the directory entries. Insertion and deletion operations that do not incur bucket splits are failure-atomic. To complete recovery, as seen in the figure, directory entries are used for the reconstruction of the table after failure. ① get the last saved directory entry to access the split status to get the consistent state if the failure has happened while segment splits. In ② and ③, we check the values of G and L to know if they point to the same level pointing into the same segment or different segments. These steps as seen in step ④ will be done by moving one level up in the directory tree until a consistent state is found.

3.4 Evaluation

This section evaluates the ACCEH scheme for the delaying of the expensive operation affecting the performance of the dynamic hashing scheme to perform operations at the cache-line level guaranteeing failure atomicity.

3.4.1 Experimental setup

We run the experiments on a server with Intel Xeon® Gold 5218 CPU® 2.30GHz processor with 32 cores (64 hyper threads), 32KB each for data and instruction cache, 1024KB L2 and 22528KB L3 caches, 256GB of Optane DCPMM (2x128GB) DIMMs in Memory Mode and 32GB of DDR4 DRAM. The server is installed with Ubuntu Server 18.04.4 LTS with the kernel 5.4.9-47-generic and PMDK 1.8 for Persistent Memory development and all of the codes are compiled using GCC 9.0.

The initial hash table size is used to be 2048 records for a fair comparison

with the cache line conscious extendible hashing. Records are also randomly inserted into the hash table in the size of 8 bytes as well. Performance evaluation is done on a uniform and skewed distribution data with different record sizes.

Experimental Parameters: as the experiment runs in a multicore environment, threads are also pinned to their physical cores to reduce data issues. Prior works like CCEH [10] and Dash [14] use dynamic hashing on PM. Accordingly, for a fair comparison, the authors used a similar setup with these works to get a fair result as all use extendible hashing schemes. Other prior works which use hashing schemes like level hashing [11] are not used in this scheme as they did not use the PM physical device for their evaluation and also concluded in CCEH [10] as it outperformed them on DRAM implementation.

In order to evaluate the new mechanism, the number of records in a single bucket is set to four and a bucket can store 64 bytes of records that can be accessed as one cache line. As the data size grows, the number of segments will grow up, while each segment contains 4 buckets to minimize the read and write overhead when probing through the records. This makes our scheme a cache-friendly hash table that restricts the size of the bucket to the cache line (64 bytes), and 256KB segment size. In order to effectively utilize the buckets in the segment, we loop in the buckets to get free space and store the new record in relatively less populated buckets with key-value stores.

During the insertion operation, the segment is checked if it can accommodate the newly inserted value. If there is no space, this mechanism will normally perform the bucket splitting until the segment size fills up. When the target bucket is already full and the other buckets are not yet exhausted, we need to move to another neighboring bucket having more space within the same segment to search for space. Hence, we move the new value to these buckets and store

the record within the segment. This helps to effectively utilize all the buckets in the segment before we trigger the directory doubling.

This approach helps to insert records in a balancing bucket and is effective in the utilization of all buckets in a segment before the expensive job of full table rehashing happens. This operation involves the rehashing and redistribution of all records to buckets called directory doubling. During the search for a record, it follows the lookup operation where the search will start from the target bucket and continue to the neighboring buckets in the same segment. Searching for a record will be unsuccessful when the item is not found in the initially hashed bucket and the neighboring buckets which belong to the same segment.

3.4.2 Performance results

During the experiment, as records are also stored in neighboring buckets in the segments, splitting time performance is relatively competitive with the state-of-the-art scheme [10] as seen in Figure 3.5. With varying records size, the average segment split time variation is negligible and hence our approach did not add additional overhead. This shows, storing records into neighboring buckets can positively contribute to the delay of directory doubling. This option can reduce considerable overheads during directory doubling to rehash all buckets to the newly created buckets.

The average directory doubling time of the new scheme is compared with [10] and has shown comparable benefits as per the initial plan. The directory doubling time comparison for the two schemes shows that our scheme is comparably faster in hashing a record to a bucket and finding free space in the neighboring buckets. As data size increases, ACCEH has been shown to store more records before initializing the segment split and directory doubling for dynamic workloads as seen in Figure 3.7. In this way, we can exhaustively store records to the

existing buckets that eventually delay rehashing operation and also efficiently utilize the allocated memory. Figure 3.8 shows, the performance of our scheme for a uniform data distribution. As seen in the figures, compared to CCEH, the time this scheme required to finish the insertion of the same data size is less. This shows ACCEH is more efficient to insert data using a uniform distribution. Additionally, when we test the schemes for skewed data distribution as seen in Figure 3.9, a similar performance gain is observed. From these results, we can conclude that ACCEH exhibited better performance in record insertion for both uniform and skewed distributions over ranges of data size.

3.4.3 Experimental analysis

Insertion operations: As insertion and lookup operations on records are done in buckets in the same segment, our scheme optimally hashes the records to buckets utilizing the space from neighboring semi-filled buckets. Accordingly, the insertion success guarantee is higher on the initial bucket or consecutive buckets having fewer load factors of the available buckets in the segment. As seen in Figure 3.7, our scheme outperformed the existing cache-conscious hashing. It has shown an overall improvement by 17% to accommodate more records compared with CCEH during the insertion of one million records.

As shown in Figure 3.6, our scheme has shown that there is a delay in directory doubling as there are records stored in the neighboring buckets after the initial bucket fills up. In the experiment, the delay becomes visible after segment splitting and directory doubling happened in the first 7 rounds. From the result, we can observe more records are inserted before directory doubling happens from that roundup. It is more visible as the record size increase.

For large data sizes, the overhead of full table rehashing is significantly higher but in this mechanism, we managed to delay and add more records to

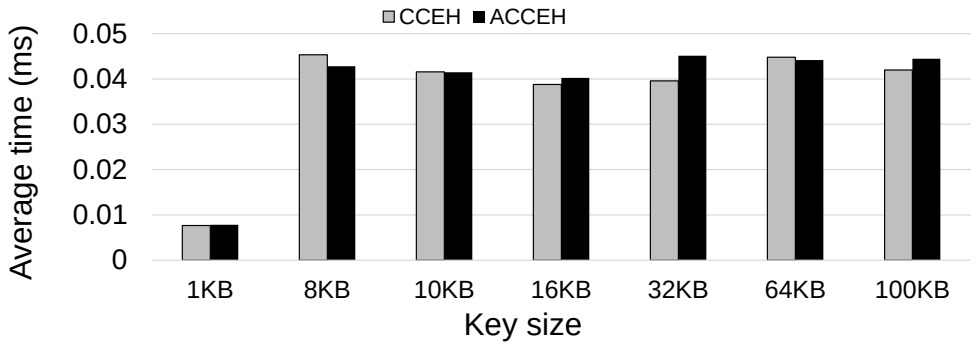


Figure 3.5 Average segment split time

the already created buckets. This is because rehashing is faster as the number of buckets are few and not advantageous to store in other buckets than the designated buckets.

As records are coming from the client, it continues inserting them into the buckets as far as there is free space in the segment. It means our approach started to effectively store at least one record in the neighboring bucket before it calls the segment split and or directory doubling operation. This advantage continues to grow as the data size grows by delaying the spit as shown in Figure 3.6 that also eventually contributes to the delay of directory doubling.

The strategy of postponing directory doubling operation has shown significant improvement in memory space utilization while retaining the original performance in [10]. This approach can even contribute to optimal performance as the data size increases and reduce rehashing as time goes on for large data insertion.

The experimental results show that the average time taken to split the filled segments is smooth and there is no significant overhead introduced in our scheme as data size grows as shown in Figure 3.5. For the uniformly distributed

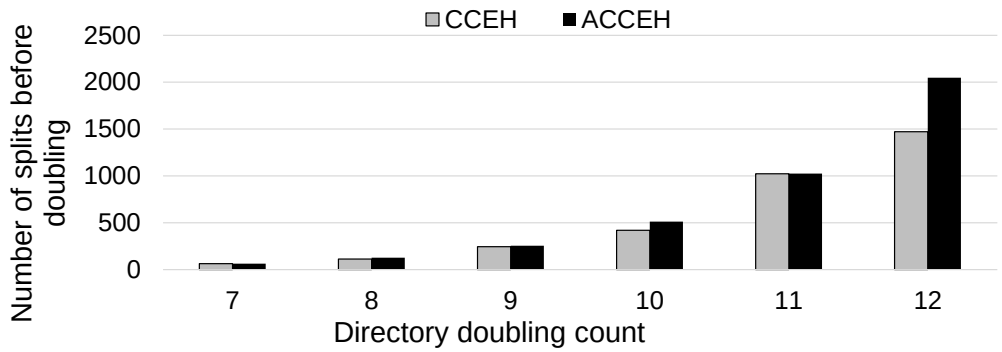


Figure 3.6 Number of splittings happened before directory doubling happens for large insertions

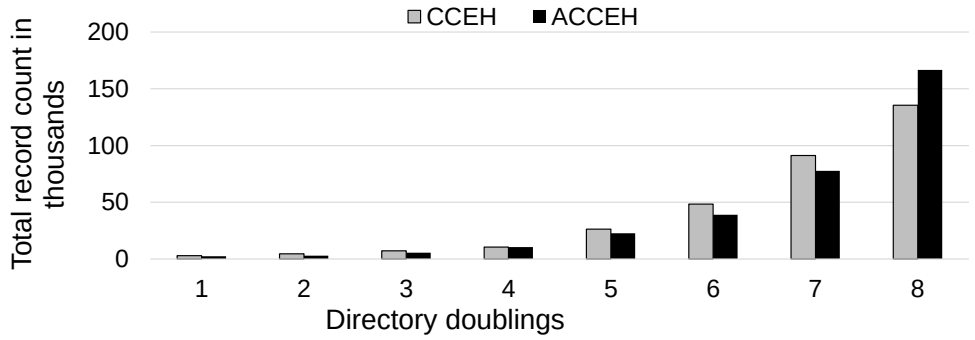


Figure 3.7 Comparison of Key-Value Entries

data, the proposed scheme improves the throughput by 1.9%, 1.7%, 6%, and 8% for 1K, 10K, 100K, and 1000K record sizes respectively gaining overall throughput of 9% during insertion of one million records. Skewed data also shows an average improvement of -10%, 3.5%, 2.3%, and 6.4% for the same data size with uniform distribution and an average of 9% performance improvement.

For small data sizes, our scheme performs less than the state-of-the-art scheme. This happens because the cost of searching an empty space in the hash table is as expensive as a directory-doubling operation for small records. This

is shown in the results and the performance for the 1K data is negative and the state-of-the-art scheme is better. But, as the data size increases, it showed an increased number of records on each round of directory doubling before calling for the next operation as the number of records increase.

During our experiment, we observed an increase in the number of records on each round of directory doubling before calling for the next doubling operations as the number of records increased. Hence, there is an increase in the number of records to be added by 14% after the directory is doubled for 8^{th} round as seen from Figure 3.6. This increases as the data size increases. Hence, as the data size increases, there is a chance of storing more records in once-created buckets which tells us that the insertion of a record in one segment is comparably cheaper than rehashing all records as it is in the predecessor.

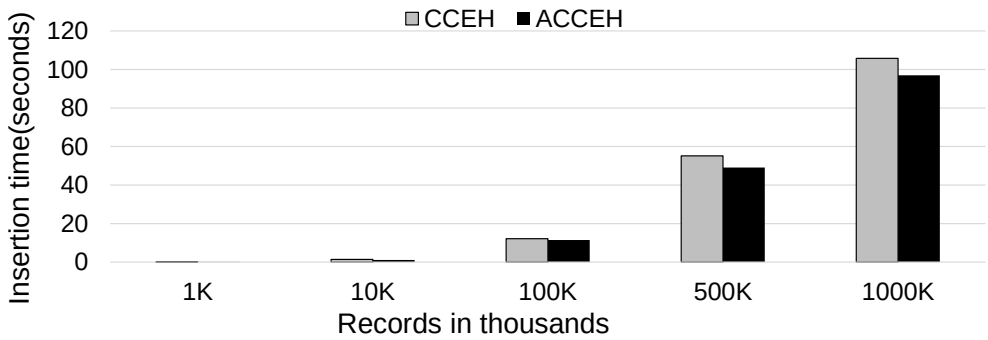


Figure 3.8 Record insertion performance on uniform data distribution

Delete operations: as inspired from [10], deletion of a record from a split segment is achieved by directory entry update and moving the move-out records to their new hash location. The migrated keys are considered invalid by the subsequent transactions like reading and over-written by the insertion operations and operations that do not incur bucket split are failure atomic [10].

Memory gain: when physical memory capacity is exhausted by running

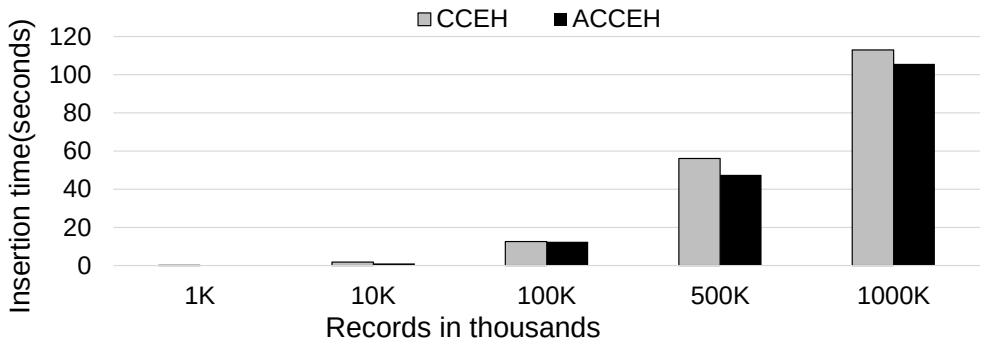


Figure 3.9 Record insertion performance on skewed data distribution

processes, the performance of actual running tasks will become in-responsive. In this regard, our scheme's memory utilization result as seen in Figure 3.10 shows that we can save memory up to 7.53% for 1M records compared to the state-of-the-art scheme. From Figure 3.10 (a), we can see the record insertion progresses that shows the memory consumption of the two schemes as time goes by. We see that ACCEH saves memory during run time and Figure 3.10 (b) shows the total memory consumed to insert 1M records. From this, we can project that for the insertion of larger data size, even more memory can be saved during the execution. This memory can be used by other applications or can be used by the same application to store more records. We observe from this result that ACCEH is memory friendly. Hence, this scheme managed to store records effectively in less memory space compared to CCEH the system can use the saved memory for other operations.

Recovery: failures may happen when a record is not fully written to the hash table. Power failures, system crashes and others may contribute to the failure if it happens while the hash table modification is in process. It is desirable to have a system that recovers from failure in a reasonable time and brings back the system to service. Records are stored as soon as received, and keys follow.

Partially written records will be ignored if the valid key values are not stored in the valid segment. Reconstruction of the hash directory during start-up is normally done as per the extendible hashing schemes but it is required to recover the directory and the local depth by loading the record to memory. In order to test and compare the recovery performance with the predecessor scheme, we run the hashing scheme for insertion. After it started to load records and killed the process as inspired from [14]. After the scheme is restarted, measure the time it takes to accept any request for different data sizes. From the result, we witnessed that our scheme has shown competitive performance. Accordingly, the recovery time required for our scheme for 1 million records is 103ms while it takes 101.7ms for CCEH. Even this result is meaningful for smaller data sizes as the recovery time is almost comparable compared to the other schemes.

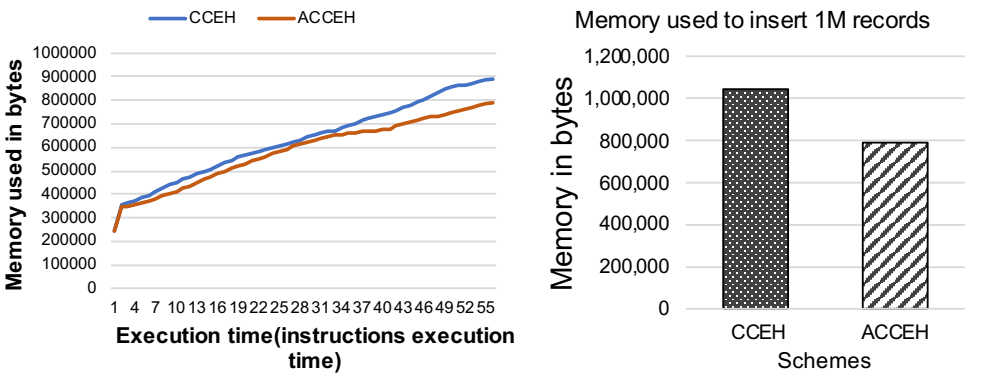


Figure 3.10 Comparison of running time memory consumption

3.5 Summary

ACCEH presented a different dimension that adds the use of available free space in the neighboring buckets to store records. Delaying unnecessary directory doubling operations is achieved by exhaustively utilizing the existing spaces in the neighboring nodes. This can help to effectively utilize the persistent memory space. The performance of the scheme is also better compared with the state-of-the-art scheme for uniform and skewed data distribution. Therefore, our scheme can be an alternative approach that enhances the CCEH by reducing the split management overhead that eventually postpones the expensive operation of directory doubling in the system that uses an extendible hashing scheme on persistent memory.

Chapter 4

Efficient and Salable Hashing Scheme for PMs

4.1 Motivation

Hash-based indexing is widely used in file systems such as ZFS [46], GPFS [47], and GFS/GFS2 [48, 49] as they provide constant lookup time for metadata operation. However, hashing also requires the applications to have predictable data size, fixed entry size, and preallocation of hash buckets. Thus, if applications have unpredictable data sizes, hashing can suffer from hash collisions and underutilization. To resolve this issue, the rehashing operation which creates and moves the entries to a larger hash table is necessary. However, this rehashing operation is expensive as it blocks index access operations and moves many entries. This can negatively impact the overall performance of the file system and lowers the latency.

To address the issue, with the emerging persistent memory (PM), the persistent nature of this device is used to improve the performance of hash-based

indexing. Emerging persistent memory (PM) devices such as Intel Optane DC Persistent Memory Module (DCPMM) provide persistence, high storage capacity, and high performance [1, 2]. As they have access latency close to DRAM, durability, and byte addressability, they are widely adapted to handle latency-critical transactions on storage systems [3, 4]. In particular to hash-based indexing, PM can be used to

However, the characteristics of PM need to be carefully considered to fully exploit the performance of PM. First, while PM has higher performance compared with the existing persistent devices, it still has lower performance compared with DRAMs. Thus, the effect of I/O operation on the performance can be greater in PM than in DRAMs. Second, since PM has a reduced bandwidth compared to DRAM, parallel access to PM by too many processes can cause significant performance degradation. Hashing scheme should involve careful design as using more threads, leads to congestion and a gradual decrease in bandwidth.

Third, as PM is byte addressable, it is also vital to reduce PM access that is synchronized with the device level access behavior as the PM read mostly hits the media due to the hash table's inherent random access patterns. Fourth, is the issue of locking as it highly affects the performance by increasing PM bandwidth. When bucket level locking is used, there is a frequent acquiring and releasing of lock, which requires a design that balances the PM read size with the bucket size.

Relatively, there is much research to improve Tree based indexing structures for persistent memory compared to the attempts for hash-based indexing structures. However, the existing indexes cannot provide the same advantages to PMs as it suffers from asymmetric read/write performance and crash consistency problems. Additionally, the existing schemes expose the PM to excessive accesses that saturate its bandwidth and become a performance bottleneck in-

troducing additional latency. Using the existing indexing structures to run on PM without modification to fit the new architecture would not help to gain the expected performance. Thus, it needs to redesign index structures for PM. High-performance and scalable indexing structures are crucial for storage systems to achieve fast queries.

With the arrival of persistence memory, a variety of indexes based on trees are designed to optimize indexing for tree variants and hashing. There are significant research results for hash tables to provide faster operations in in-memory systems indexing structures. The efficient lookup time for mapping values with a particular key in hash-based indexes needs to achieve faster access irrespective of the size of the data. Hashing-based indexing structures are widely used in different applications using key-value stores [5, 6].

In applications like the key-value store, sizes of records can not be predicted as it involves the dynamic insertion or deletion of items. Therefore, a dynamic hashing scheme that involves dynamic resizing is the best choice to adjust the table size to fit records to the scheme. PM's read/write latency during searching a record in a large portion of storage can introduce cache misses [14] and PM accesses. Concurrency controlling during these operations needs careful attention as it introduces additional read/writes for locking that ends up in further bandwidth consumption. As the data insertion increases, the load factor of the hash table becomes high forcing the growth of the hash table. To accommodate that, the hash table will be rehashed and the data will be relocated to the newly created buckets. Rehashing is an expensive operation that involves doubling the buckets as it is an incremental operation and hence the new index at which the values have to be mapped to a new location. Moving the existing records to a new bucket location degrades the throughput halting the access of the indexes during rehashing. On top of the read/write latency of PM, it increases the total

query latency.

In this paper, we propose an Efficient and Salable Hashing Scheme called ESH to improve hashing for emerging persistent memories. ESH addresses the existing performance challenges that are not addressed by combining efficient utilization of PM read blocks and storing a bucket effectively into it and consuming the allocated PM space and bandwidth. When the existing bucket is full of records and requires an expensive rehashing operation, ESH checks the available space in the neighboring buckets in the same segment and utilizes the free space if it exists. When the existing bucket is full of records and requires an expensive rehashing operation, ESH checks the available space in the neighboring buckets and utilizes the free space if it exists. This allows ESH to delay the expensive full table reshaping operation during the insertion operation, improving the load factor and the overall utilization of the persistent memory. It used a bucket size that fits the one-read block of PM to gain reduced PM access and uses bucket level locking to scale well in a multi-threaded environment. We evaluate our scheme using Intel Optane persistent memory with various configurations. Our evaluation results show that ESH can improve the inertion performance by up to 30%, and search performance by up to 10% compared to the state-of-the-art dynamic hashing schemes. In addition, it can improve the load factor by up to 91% compared with the existing schemes.

4.2 Related Work

The durability, byte-addressable, and access latency close to DRAM benefits of Persistent memory make the promising candidate for building applications that use extensive memory systems. However, the change in memory architectures makes the traditional data indexing inefficient for data consistency [9]. Exist-

ing works have improved tree-based indexing structures for persistent memory with reasonable lookup time and better recovery [3, 7, 9, 13, 45]. As the in-memory data size increases, rehashing the traditional hash tables incurs higher latency requiring better rehashing techniques for persistent memory. To ease the overhead during rehashing, linear probing, separate chaining, cuckoo hashing, CCEH [10] and Dash [14] are a few of the techniques used. CCEH - the variant of extendible hashing designed to optimize access of hash table buckets to cache-line access that significantly minimizes the number of cache-line accesses. It also reduced the overhead directory operations as it groups a number of buckets into intermediate sizes called segments. This approaches helps to reduce the size of the directory to cache-line-sized buckets and also reduces rehashing management during failure recovery [50–52].

Extendible hashing was developed for time-sensitive applications that use a trie for bucket lookup. It uses re-hashing which is an incremental hierarchical operation to fill the hash table. Therefore, it needs to be as effective as possible during the full table rehashing. The dynamic allocation of buckets and their pointers need to be tracked for getting records in the hash table. Different research results contributed to making the rehashing efficient.

As time-sensitive applications are more affected by hash table rehashing than the growth of tables, more researchers have focused on optimizing the rehashing schemes. The growth of table size also has a significant performance factor on record look-ups. As it uses a directory to index buckets that are dynamically added or removed during the run time, it involves splitting new buckets with rearranged values. This will expand the directory to get more storage for pointers to the new buckets. This can be done linearly by organizing buckets using a directory entry pointing to individual buckets. Designing a proper splitting and rehashing strategy for different workloads is a key approach to getting optimal

in-memory systems [53] [54].

A cuckoo-based hashing [42] reduces write to the PCMs with higher memory efficiency by displacing randomly selected records to alternative buckets [42]. Records are inserted into one of the buckets using independent hash functions is designed but the performance is still slower than the linear probing and Cuckoo hashing [45, 55, 56]. Reducing the lookup cost to log scale is achieved using binary tree [43] and is further divided into two level hashes [11].

Most of the research focuses on reducing the cost of full table rehashing and improving the load factor. There are also other proposals that design additional levels [42] that store records that are maintained on memory and then stored in the hashing table. Cacheline level indexing with failure atomic structure that dynamically manages hash expansion [10] on PM guaranteeing constant hash table lookup time gain popularity as it effectively sets the size of a bucket to a cache line that minimizes the number of cache line access reducing the overhead of data access of multiple cache line.

Another approach that uses PM tree structure to avoid unnecessary reads during record probing [14] uses fingerprinting used in PM tree and designed a lightweight one-byte hash that helps to detect if there are keys that save PM read/write bandwidth. It includes a strategy that postpones segment split that helps to improve space utilization and is implemented for PM using the PMDK libraries [57].

As the data set increases, rehashing the entire records is one of the challenges and resource-consuming operations. Extendible hashing, therefore, is designed to be used by applications that are more time-sensitive as the rehashing operations are done incrementally as data size increases or bucket overflows.

During rehashing, in order to place all records to their correct buckets for a given hash key, the directory entry pointers double as the depth increases as

seen in Figure 2.2. The bucket address pointer called directory points to hash buckets [10] and is updated like the above scheme with the size increases.

Building a dynamic and scalable hash table for the new architecture of persistent memory hardware that can run in high load factors and instant recovery is critical [11,14,42,44]. Additionally designing hashing schemes that reduce the overhead of dynamic memory management with better hash table lookup time and other related operations is equally important and this work will contribute to these efforts.

4.3 Design and Implementation

Similar to other approaches [10,14], ESH is designed based on the segmentation of buckets into two layers segments and buckets in it. In this paper, we present an efficient and scalable hashing scheme that effectively stores a key value on persistent memory. It stores more records in the buckets by exhaustively utilizing the free spaces in the buckets in the same segment to increase memory utilization by efficiently filling up the buckets in each segment.

4.3.1 High level Design

The proposed scheme is designed to leverage performance characteristics of Optane persistent memory using hashing techniques that improve performance and scalability addressing problems in hashing index structures. Thus, this work focuses on some design principles as follows:

(a) **Avoid both unnecessary reads and writes** – Write operation using hashing scheme usually requires frequent access to media. This has a big impact on the performance of the scheme. All operations like reading, writing, and other operations will be affected by the cumulative of

these drawbacks. On top of this, for a device with less speed compared to DRAM, frequent read and write operations has a more severe performance overhead. Therefore, ESH reduces these unnecessary PM reads and writes to gain high end-to-end performance.

(b) **Bucket level locking to allow multi-threading** – A good locking strategy generally results in fewer requests to lock and unlock data for sequential access and manipulation, which translates to reduced CPU cost. To provide better concurrency, ESH uses a better lock granularity as the cost of each lock and unlock consumes CPU time. Therefore, a write thread locks only the bucket under operation that reduce the lock contention and the other buckets can be visible for other threads. Other operations are lock-free to permit more concurrency. Readers can access buckets but segment splitting and directory doubling operations are not lock-free to prevent data inconsistency. The active Writer thread is responsible for creating and locking during segment splitting or doubling the directory as access of the segment or directory under modification by one thread will be inconsistent if accessed by other threads. So, ESH is designed with the concept of a lock with as minimal data block as possible. i.e, a bucket.

(c) **Optimistic scaling on multicore machines** – To take the advantage of parallel CPU resources, ESH is designed with optimistic scaling. Previous research mainly focused on reducing cache line flushes and using PM writes to get scalable performance. This actually has scalability issues when they are deployed on the actual PM devices. As PM has limited bandwidth, ESH reduces unnecessary PM reads and lightweight concurrency control to further reduce PM writes to guarantee persistency and

less overhead. Hence, in ESH a bucket is accessed as one block of PM which will reduce the PM access overhead.

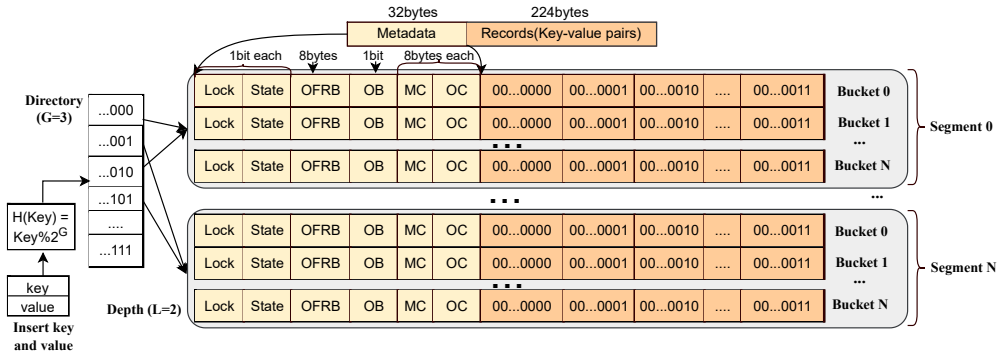


Figure 4.1 Overall architecture of Scalable hashing scheme and bucket structure

4.3.2 Bucket layout

In this section, we will describe the overall architecture of ESH and bucket structure as shown in Figure 4.1. The bucket has two sections - the metadata and the records. The first section of the bucket - the metadata stores information about the bucket to speed up access and keep consistency in storing actual records. contains vital information about the bucket status, lock, and overflow bucket for fast access during operations including those listed below. They account for 32 bytes of the total bucket size for the implementation to run the experiment.

The second section is the record section. It is used to store the actual records (key-value stores). As each directory entry points to a segment that contains a fixed number of normal buckets, the metadata information helps operations like locking, state information and overflow records from neighboring buckets that do have not enough space for the insertion.

For any operations on the buckets, the scheme checks if the bucket is at a consistent state using the *state* flag in the metadata. If any operation has left the buckets at an inconsistent state before it persists as a result of crush or power failure, it avoids any operation during the accident and the data as invalid. To avoid the expensive logging of all operations insertion is successful only after the metadata is persisted otherwise the record is discarded.

Locking (lock) – ensures the correct access to a bucket when there are multiple threads in execution. To support this mutual exclusion as shown in Figure 4.1, ESH provides a lock indicator to enforce a limit to access the bucket to guarantee exclusion. It consists 1bit of metadata to store the locking information for the writer threads. For example, a writer thread must hold the lock bit by changing it to 1 to write the records to the bucket. Then, it returns to 0 for other threads which try to get access to the same bucket again after the lock-holding thread releases the lock. As a result, it maintains consistency and supports bucket-level locking. Writer threads must hold the lock bit changing to 1 while writing to the bucket to maintain consistency and other writer threads should wait until the lock value changes.

State – is a Boolean that checks whether the system was shut down cleanly. If there is a power/system failure or other unexpected events, ESH can be inconsistent. To solve that, once a thread finishes an operation on a bucket correctly, it changes the state to 0 which indicates a correct flag so that the following thread can access the bucket. There are many situations where the system is closed without appropriately closing the hashing scheme because of power failure or other unpredicted cases leaving the system in an inconsistent state. Hence, to maintain better consistency, once a thread finishes operation on a bucket correctly, a 1-bit flag shall trace the state of the bucket so that the following threads can access the bucket. If the state indicates 1, it means that

the bucket was not correctly closed by the previous thread. So, the result of the operation is discarded and the recovery procedure is initiated.

Overflow Bit(OB) – It consists of one bit that indicates the status of the bucket whether it is full or not during insertion. When a thread tries to insert a record, the thread accesses this metadata. If the value of OB is 0, it indicates there is available space in the bucket, Thus, the thread inserts the record in the bucket. Once insertion is done, the membership count increases the value to trace the available space left. Its value will be changed to 1 by the insertion threads when the insertion is succeed. If the value of OB is 1, it indicated the bucket has no more space, so move to the next buckets for insertion. This speeds up checking the status of the bucket for insertion and search operations.

Overflow Records Address Byte(OFRB) – In ESH, records are stored in their hashed buckets in a segment until the bucket is full. Once a bucket is full, ESH does not immediately trigger segment split or directory doubling. Instead, ESH checks whether there is a neighboring bucket that has available space in the same segment. If so, it stores the coming records in the bucket. When the record is moved to the neighboring bucket, the OFRB holds the address of the bucket to which the record is moved. By doing so, when ESH searches the record stored in a neighboring bucket, ESH can search the record easier and faster by referring to OFRB. We note that this OFRB size is variable size and configurable according to the maximum number of the neighboring bucket and architecture. For example, there are three neighboring buckets. If we made three OFRB fields, we can directly point to three neighboring buckets. It enables searching for the moved records faster while the record size is reduced. If we made only one OFRB field, we continue to search for the moving record in the next neighboring buckets until we search for the record. It can induce the search for the record in all the neighboring buckets in the worst case while

the record size can be increased.

Membership Count – it is an 8bytes and counts the total records stored or pushed to each bucket as their hashed bucket is already full. It is incremented by insertion threads when the insertion is succeeded. ESH uses this MC to identify how many records are inserted or moved to each bucket. Thus, ESH knows the available number of records in the bucket for upcoming records and ease of searching.

Overflow Count(OC) – OC indicates the number of moving records from the original bucket to the neighboring buckets because of overflow. As records can be stored in the neighboring buckets, differentiating records that are hashed to their original bucket and those moved from the neighboring buckets may confuse other operations during searching. Thus, ESH increases OC whenever it stores a record in a neighboring bucket due to an overflow of the original bucket. It also decreases OC whenever it deletes a record in a neighboring bucket. During the search operation, to continue the search to the neighboring buckets, the overflow counter(OC) tells that there are more records in the neighboring buckets because of overflow. Then, the search will continue to read the OFBR address to continue the search operation in the other buckets in the same segment.

4.3.3 Operations

Any operation in this scheme has to check if the intended bucket is in a consistent state. A running process that intends to write or update the bucket reads the metadata and gets the lock and checks the overflow bit to know if it is full. As shown in Algorithm 4.1, when a thread inserts a record to a bucket, it performs a hash operation via a key and gets a hash value (i.e., `hashval`) as seen in line 2, Algorithm 4.1. Then, it gets the original bucket i.e., `original_bucket` to

which the hash value is going to be stored and the target buckets in case there is overflow (`target_bucket`) as indicated in lines 4-5, Algorithm 4.1. Then, first, the threads have to check the state information of the bucket. Hence, the thread gets the state and lock variable from the target bucket as seen in lines 7-8, Algorithm 4.1.

From this information, if the state of the target bucket is not consistent, the insert operation returns an error and performs a recovery operation to get a consistent bucket. If there is no problem, the thread continues to perform the insertion operation to the original bucket as seen in line 9, Algorithm 4.1. To check if that bucket is full, the thread checks if `overflow_bit` of the target bucket is set or not as seen in line 10, Algorithm 4.1. If it is set, then the thread updates the target bucket's neighboring bucket address (i.e., `neigh_bucket_add`) that traces the neighboring bucket by the next bucket of the target bucket as seen in line 11, Algorithm 4.1. This means that there is no free space to insert in the target bucket. To move to the next bucket to search for a free space, the thread updates the target bucket by the next bucket of the target bucket since the thread retries the insertion operation using the next bucket as seen in lines 12-13, Algorithm 4.1.

If the bucket has free space, the thread locks the target bucket and performs insertion. This is known by the threads if `overflow_bit` of the target bucket is not set in line 14, Algorithm 4.1, this means that the target bucket has free space. Thus, the thread first holds a lock and it inserts the key, value, and hash value into the target bucket as seen in line 16, Algorithm 4.1. Then, it updates `overflow_count` of the original bucket and `member_count` of the target bucket by 1 as seen in lines 17-18 and if `member_count` is full by this update, `overflow_bit` of the target bucket is set as seen in lines 19-20, Algorithm 4.1. After this insert operation, the thread releases the lock and returns the result

(success) of this insert operation.

Generally, if there is a bucket with fewer records, we loop in the buckets of the same segment to find the right location to insert the record in neighboring buckets. This traversal to insert a record can not go out of the segment as it complicates the lookup. This approach is mainly to effectively utilize the available buckets which are partially filled or not filled at all. For every insertion, it hashes to the right bucket, and if it is already full move it to other buckets with fewer elements and insert using probing. It inserts the records into the available space in the segment which holds four buckets as indicated in the architecture in Figure 4.1.

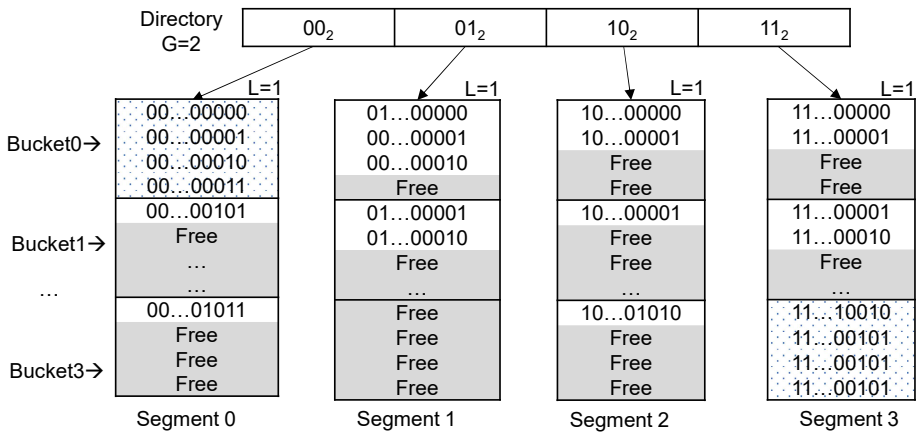


Figure 4.2 Segmented Extendible hashing with cache conscious feature

In ESH, segment stores a fixed number of buckets of metadata of 32 bytes and 224 bytes of key-value pairs and each directory entry points to these segments containing buckets and data are stored in the bucket as arrays of records as seen in Figure 4.2. A bucket stores records from hashing function and records moved from the neighboring buckets. During the insertion of records, if the bucket has more space and the neighboring bucket has, we move it to these

Algorithm 4.1: Simplified insert algorithm in EHS

```
1: def EHS_insert(key,value):
2: hashval = hash(key);
3: /* check state and get the lock from the targeted bucket */
4: original_bucket = get_segment.bucket(hashval)
5: target_bucket = original_bucket
6: retry:
7: state = target_bucket.state
8: lock = target_bucket.lock
9: if state == 0 /* check if the bucket is consistent */ then
10:     if target_bucket.overflow_bit == 1 then
11:         target_bucket.neigh_bucket_addr = target_bucket.next
12:         target_bucket = target_bucket.next
13:         goto retry:
14:     else
15:         lock target_bucket /* if the bucket is not full */
16:         target_bucket.insert(key, value, hashval)
17:         update original_bucket.overflow_count
18:         update target_bucket.member_count
19:         if target_bucket.member_count == full then
20:             target_bucket.overflow_bit = 1
21:         end if
22:         unlock target_bucket
23:     end if
24: end if
25: return insert_result
```

neighboring buckets to delay doubling. This can be used when a given bucket has not had enough space for insert and there are other neighboring buckets free or have space to accommodate the records.

For the concurrency control, the initial CCEH scheme that maintains the lock is used as it has witnessed showing reasonable performance for extendible hashing and a flag that helps to check if the used bucket is cleaned. When records are inserted into the buckets, if there is no space available for the newly inserted record to any buckets in the same segment, we check for available space for insertion in neighboring buckets. This can be achieved in a similar way the CCEH uses a tracker for the global and local depth during splitting and directory doubling.

If there is a bucket with fewer records, we loop in the buckets of the same segment to find the right location to insert the record in neighboring buckets. This traversal to insert a record can not go out of the segment as it complicates the lookup. This approach is mainly to effectively utilize the available buckets which are partially filled or not filled at all. For every insertion, it hashes to the right bucket and if it is already full we track the other buckets with fewer elements and insert using probing.

The records in the buckets are accessed as arrays. As there is a fixed number of elements in each bucket, it is easy for the search operation to go through the buckets as seen in Algorithm 4.2. Searching for a value is done starting by hashing the key and looking it up in the buckets as other schemes did. For the normal search operation, items are accessed by hashing from the first hashing location. But, when searching for a value that is moved to another bucket, the search item can be accessed by looping through records that belong to the same segment. Search also performs a hash operation via a key and gets a hash value like the insert operation as seen in line 2, Algorithm 4.2. Then, the search

thread gets the target bucket and reads the `overflow_bit` flag and state from the target bucket as seen in lines 4 and 6-7, Algorithm 4.2. And then, like the insert operation, the search operation can be performed only if the state of the target bucket is consistent as seen in line 8, Algorithm 4.2. If the state is consistent, the thread checks whether the `overflow_bit` is set or not refer to line 9, Algorithm 4.2. If `overflow_bit` is not set, it means that the target bucket is not full now. Thus, the thread can find the value of the key via the search operation in the target bucket as seen in line 10, Algorithm 4.2.

If the `overflow_bit` is set as seen in line 11, Algorithm 4.2, the thread tries to find the key in the target bucket as seen in lines 12-13, Algorithm 4.2. Even if the target bucket is full now, the requested key could be already inserted in the target bucket. If the target bucket has no key, the thread updates the target bucket by the neighboring bucket and retries to search for the key in the changed target bucket as seen in lines 15-16, Algorithm 4.2. Finally, the thread returns the value of the key as in line 19, Algorithm 4.2.

4.3.4 Metadata and hash table operation

Figure 4.3 depicts the more detailed metadata and hash table operations (i.e., two insert and one delete operations) to show how the metadata is updated or used with our hash table operations. As shown in the figure, when a thread tries to insert a record (R1) whose original bucket is bucket 0, it checks the state of bucket 0 if it is consistent or not (we assumed that all the buckets are consistent) ①. If the state is consistent, the thread checks the overflow bit (OB) if bucket 0 is already full i.e., have the overflow or not.

In this case, the original bucket is full. Thus, the thread gets the neighboring bucket, bucket 1 in this case in the same segment. It updates the OFBR of bucket 0 by bucket 1's address and searches for the free space in the bucket

Algorithm 4.2: Simplified search algorithm in EHS

```
1: def EHS_search(key):
2: hashval = hash(key);
3: /* get target_bucket from the target segment */
4: target_bucket = get_segment.bucket(hash)
5: retry:
6: overflow_bit = target_bucket.overflow_bit
7: state = target_bucket.state
8: if state == 0 /* check if the bucket is consistent */ then
9:     if target_bucket.overflow_bit == 0 then
10:        value = target_bucket.search(key)
11:    else
12:        if if key is in the target bucket then
13:            value = target_bucket.search(key)
14:        else
15:            target_bucket = target_bucket.neigh_bucket_addr
16:            goto retry
17:        end if
18:    end if
19:    return value
20: end if
```

②. In this case, there is free space in bucket 1 so that the thread can insert the record in bucket 1. Thus, the thread increases bucket 0's overflow count (OC) by 1 and bucket 1's member count (MC) by 1. In this case, the member count of bucket 1 is full which denotes bucket 1 is full so that the overflow bit is updated by 1.

Consider another insertion operation that continues to add a record to the same bucket. Then, the thread tries to insert another record (R2) whose original bucket is bucket 0 as seen in ③. In this case, the original bucket and bucket 1 are already full ④. Thus, the thread searches for free space in bucket 2 by getting the neighboring bucket (bucket 2). Additionally, it adds bucket 2's address to OFBR of bucket 0 to sustain the later tracing ⑤. Since bucket 2 has free space, the thread can now insert R2 into bucket 2. Then, the thread increases the overflow count of bucket 0 by 1 and the member count of bucket 2 by 1.

After these two insertion operations, another thread tries to delete R1 which is stored in bucket 1 ⑥. First, the thread searches for the record in its original bucket (bucket 0). In this case, there is no R1 in the original bucket. Thus, the thread tries to search for the record in the next neighboring bucket (bucket 1). Since bucket 1 has R1, the thread deletes R1, decreases the member count of bucket 1, and updates the overflow bit of bucket 1 by 0. Finally, the thread removes bucket 1 as a neighboring bucket.

4.3.5 Implementation

From the aforementioned properties of persistent memory and the hashing techniques, we implemented our mechanism to minimize the performance impact and optimize memory utilization by reducing frequent full table rehashing that contributes to tail latency.

Hashing of records in this approach is relatively similar to hashing of records

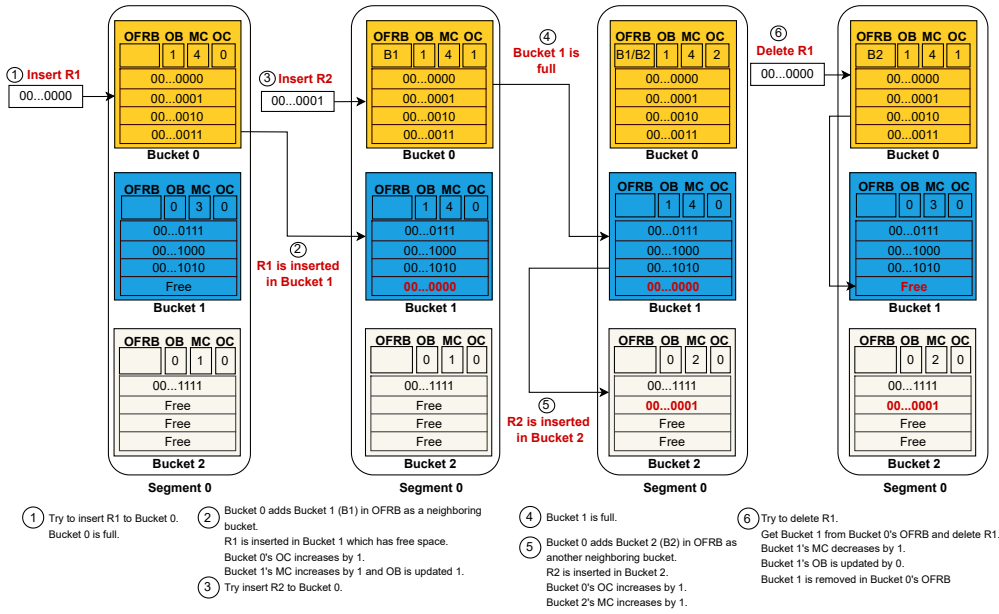


Figure 4.3 Metadata and hash table operations in ESH (R1: record 1, R2: record 2, OFRB: neighboring bucket address, OB: overflow bit, MC: member count, OC: overflow count)

to a bucket in CCEH to store records. When there is a request to insert a record to an already filled bucket while the neighboring buckets have free space, the scheme checks if insertion requires segment splitting. It performs balanced and exhausted insertion to all buckets in the same segment before rehashing is triggered. If there are neighboring buckets that are either empty or semi-filled, it is less expensive to store records in these buckets than trigger directory doubling. This can be done as records are stored in the buckets and accessed as arrays. Directory doubling and rehashing of records will be initiated when there is no space in all buckets in the same segment. and This insertion can be pointed to by the pointer that helps other operations and is also a signal that shows some records are shifted to other buckets in the segment.

Deletion of records uses the insertion overwrite as designed by the initial scheme where invalid values during hashing can be overwritten by replacing them with valid ones. When the record is stored in another bucket as per this scheme, the deletion operation will nullify the record in the bucket and this space is counted as free. Accordingly, if a given record is hashed to bucket *b0* but moved to bucket *b3* because it is already full using our scheme, deletion will lookup the record in *b0* and if not found it continues to *b3* looping through the records in the buckets. Once the record is found in the intermediate buckets or *b3*, it nullifies the record and thus makes the record invalid. If certain records are deleted from an already filled bucket the system will move the record from another bucket back to its original hashing bucket to ease the lookup for the next operation if data access is not fast. This operation has less priority as the size of the bucket is the size of the cache line and once all buckets are exhausted, segment split will rehash all records to their proper buckets.

4.3.6 Concurrency

In a multi-threaded environment, performing multiple queries is challenging as multiple threads are going to access a hash table. There is a possibility of collision as they may run off the same object in the table. Especially, expensive operations like full table rehashing require exclusive access to the entire hash table that could block other subsequent operations resulting in to increase in response time.

This becomes critical when the size of the hashing table increases. Therefore, we evaluated the latency of concurrent operations like insertion and search operations. As we run a large number of insertions in a multi-threaded environment, the insertion throughput of all schemes under comparison increased and both Dash and ESH showed better overall performance. From the comparison

result, ESH has shown better performance as the lock contention is reduced to a single bucket level, and also all operations are aided by the metadata information.

4.3.7 Recovery

Recovery: failures may happen when a record is not fully written to the hash table. Power failures, system crashes and others may contribute to the failure if it happens while the hash table modification is in progress. For time-sensitive applications, it is desirable to have a system that recovers from failure in a reasonable time and is ready for services. Records are stored as soon as received, and keys follow. Partially written records will be ignored if the valid key values are not stored in the valid segment. Reconstruction of the hash directory during start-up is normally done as per the extendible hashing schemes but it is required to recover the directory and the local depth by loading the record to memory. In order to test and compare the recovery performance with the predecessor scheme, we run the hashing scheme for insertion. After it started to load records and killed the process as inspired from [14]. After the scheme is restarted, measure the time it takes to accept any request for different data sizes. From the result, we witnessed that our scheme has shown competitive performance. Accordingly, the recovery time required for our scheme for 1 million records is 103ms while it takes 101.7ms for CCEH. Even this result is meaningful for smaller data sizes as the recovery time is almost comparable compared to the other schemes.

4.4 Evaluation

In this section, we set up the experiment for our scheme and compare it with state-of-the-art schemes designed on the same dynamic hashing on persistent memory. In this evaluation, we chose to compare our scheme with CCEH [10] and Dash [14] as these use extendible hashing schemes where CCEH mainly focused on cache consciousness and Dash aimed to reduce read and write on a 256byes of bucket size.
Experiment result shows that our scheme:

- Efficiently utilizes segment level space as there is no space left empty in the segment before a segment split operation is triggered or directory doubling happens.
- In a multi-core environment, the performance of our scheme scales up well compared to the state-of-the-art hashing that uses similar hashing schemes.
- Competitively achieves high load factor without compromising performance and recovery with minimal cost.

4.4.1 Experimental setup

We run the experiments on a server with Intel Xeon® Gold 5218 CPU® 2.30GHz processor with 16 cores (32 hyper threads), 32KB each for data and instruction cache, 1024KB L2 and 22528KB L3 caches, 256GB of Optane DCPMM (2x128GB) DIMMs in Memory Mode and 32GB of DDR4 DRAM. The server is installed with Ubuntu Server 18.04.4 LTS with the kernel 5.4.9-47-generic and PMDK 1.8 for Persistent Memory development and all of the codes are compiled using GCC 9.0.

Experiment parameters – as the experiment is run on a multicore environment, threads are also pinned to their physical cores to reduce data issues. Prior works like CCEH [10] and Dash [14] use dynamic hashing on PM. Accordingly, for a fair comparison, we used a similar setup with these works to get a fair result as all use extendible hashing schemes. As other prior works which use hashing schemes like level hashing [11] are not used in this scheme as they did not use the PM physical device for their evaluation and also concluded in CCEH [10] as it outperformed them on DRAM implementation. Accordingly, setting parameters and initial data size to set up an experiment. CCEH uses a 16KB segment size and the bucket size is limited to 64 bytes and four probing lengths whereas Dash uses 256 bytes(four cache lines) bucket size and also 16KB segment size. Hence, we planned to compare ESH with these hashing schemes, we use 256 bytes for bucket size and a 16KB segment size with 4 probing distances. In the prior work in CCEH, the increase in performance is recorded as the segment size increases. This is because, the larger the segment size is, the more the scheme stores data in memory and performs probing less frequently. In ESH, a bucket stores 28 key-value pairs of 8 bytes each totaling 224 bytes and 32 bytes of metadata, and a single segment stores 4 buckets. We choose this to efficiently use the advantage of Optane persistent memory buffer that issues a 256bytes write block [58]. Even though the previous scheme in CCEH [10] used a 64byte size to utilize one cache line for a bucket, that is chocked by the PMEM write block. Extending the size of a bucket to a 256-byte block of PMEM will be a balancing point where we use the two blocks efficiently. As the data size increases, the size of the hashing table increases resulting in to increase in directory entries and the number of segments following the extendible hashing technique.

Benchmarks – to test how the hashing scheme runs under a particular

load to find the performance and tune collecting diagnostic information needed to eliminate bottlenecks. Testing hashing operations using benchmarking tools that represent realistic workloads are used and results are compared with the other schemes. Insertion, search and update operations of the scheme are evaluated and get a competitive performance.

To run the experiment, the initial data load of 10 million records is preloaded using the GCC file that defines Hash_bytes, a primitive used for defining hash functions based on the public domain from MurmurHashUnaligned [59] to get fast and high-quality hashes and get a fair comparison result with Dash as this was used as hashing index. To stress-test our hashing scheme, we use both uniform and skewed /Zipfian distributions with varying sizes. The experiment result shows our scheme achieved better performance benefiting from cross-bucket record storage that reduced the frequency of directory doubling – the expensive operation in hashing operations and fewer contentions as the metadata helps to minimize unnecessary operations. For the experiment, we used a fixed size key-value of 8 bytes of integers, and the variable length key experiment is used from the pre-generated variable keys by Dash [14] by the benchmark.

4.4.2 Comparative Performance for varying data sizes

To understand the basic performance gain of the design, we measured the insertion performance of ESH and compared the performance with CCEH and Dash on varying data sizes for a single thread. This helps to show the performance improvement of the schemes regardless of the data size. Later we extend this evaluation to varying numbers of threads to show the total improvement. Hence, as reading uses the underlying performance bound, we compared the performance of data insertion operations as the data size increased. Figure 4.4

shows the insert performance when the data size is 1, 5, 10, 20, and 30 GB. As shown in the figure, ESH outperformed CCEH hashing schemes by 31.83% and Dash hashing scheme by 3.94% when inserting 30GB of data.

This performance gain comes because ESH efficiently utilizes the empty spaces in the same segment, delaying the expensive rehashing operation. Thus, while other schemes perform the rehashing operations, EHS can continue to perform the insert operation, leading to improve insert performance. Another contribution to this gain comes from the NUMA efficient design used by ESH. As remote PM access always suffers from low performance, most or all tasks' memory/PM in this design are assigned to their local NUMA node. This optimizes memory/PM-processor locality by binding to their local nodes.

The difference in performance between CCEH and EHS is greater than that between Dash and EHS. It is because CCEH uses 64 bytes for a bucket size that does not match the read block of the persistent memory while EHS and Dash use 256 bytes for a bucket size that exactly matches one read block of the persistent memory. Thus, this small bucket size of CCEH forces more frequent rehashing and flush operations compared with the other two schemes.

As the size of key-value stores increases, both Dash and ESH achieved relatively similar performance but better than CCEH by up to 3X as both used metadata and fingerprints respectively. Although CCEH has shown better performance on small data, it loosed its performance as the data size grows. This is because it uses cache line-sized data and flushes it faster than the other hashing scheme to complete hashing in a shorter time.

4.4.3 Performance on a varying number of threads

Insert: Figure 4.6 shows the throughput of CCEH, Dash, and EHS during the insert operation while a varying number of threads. As shown in the figure, EHS

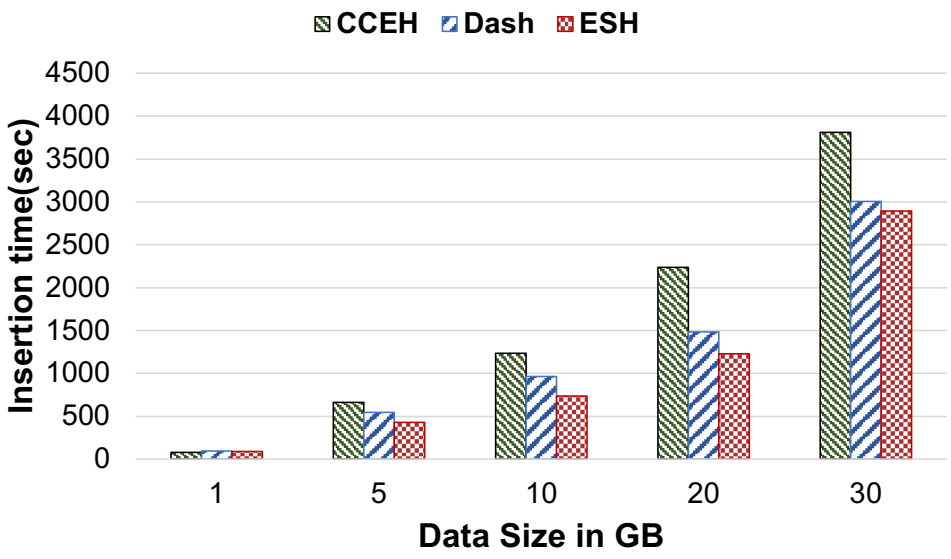
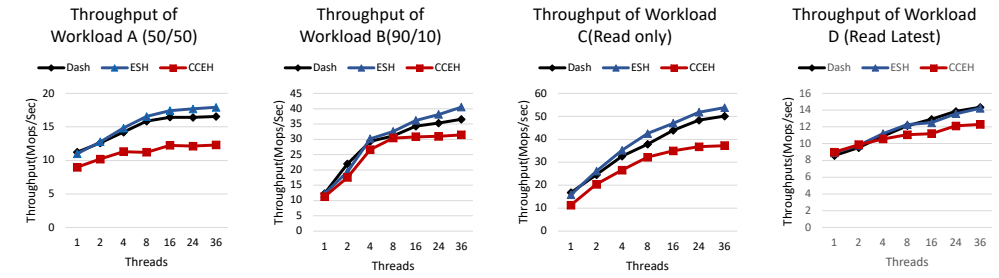


Figure 4.4 Single thread data insertion performance comparison under a fixed key length

outperforms CCEH and Dash by up to 53% and 14%, respectively, when the number of threads is 36. More specially, when the number of threads is small, the performance of EHS and Dash is similar. However, when the number of threads is more than 8, the performance of EHS increases more. It demonstrates that utilizing available spaces in the buckets in a segment can improve performance and scalability.

In the other schemes, once a given bucket is full, there is no mechanism to check the neighboring buckets for available space even if all of them are empty. This leads to triggering full-table rehashing assuming that the hash table is full while it is actually semi-empty. In ESH, we effectively used the hash table as we move records to the neighboring bucket in the same segment thus it delays the rehashing operation.

Additionally, to evaluate the performance of the scheme, testing using the YCSB benchmark is performed. The evaluation is done using 30GB of data generated by the schemes under a similar evaluation setup for a fair comparison. As we see from Figure 4.5, the schemes are evaluated by four YCSB workloads and the evaluation result shows ESH has shown better performance as the number of threads varies. Figure 4.5(a) is an update-heavy operation, (b) shows performance results for reading-heavy workloads, (c) is a result of mainly read operation, and the last shows evaluation result of the latest read operations. For all operations, ESH shows comparably better than other schemes even though the performance improvement is not too high.



(a) Update heavy workload (50%read, 50% update)

(b) Read heavy workload (90%read, 10% update)

(c) Read-only workload (100% read)

(d) Read the latest workload (95%read, 5% insert)

Figure 4.5 YCSB Evaluation for different workloads

As we increased the size of a bucket, more records can be inserted into the hash table. In this design, we used the advantage of Persistent memory write block to store one bucket that helps to insert more records into buckets. This also helps to delay the full table rehashing and reduce unnecessary directory doubling which is expensive and affects the total performance of the hashing scheme.

For insertion to be successful, the hashing function checks to access the segments using the directory entry that points to the segments. As a segment

consists of 4 buckets in this design, we can access them using the entry pointer to the segment. Insertion operation, therefore, checks the segment if the buckets in it can accommodate the newly inserted value. If there is no space in the bucket in the segment, then it will perform the bucket splitting and insert the new value to a designated location.

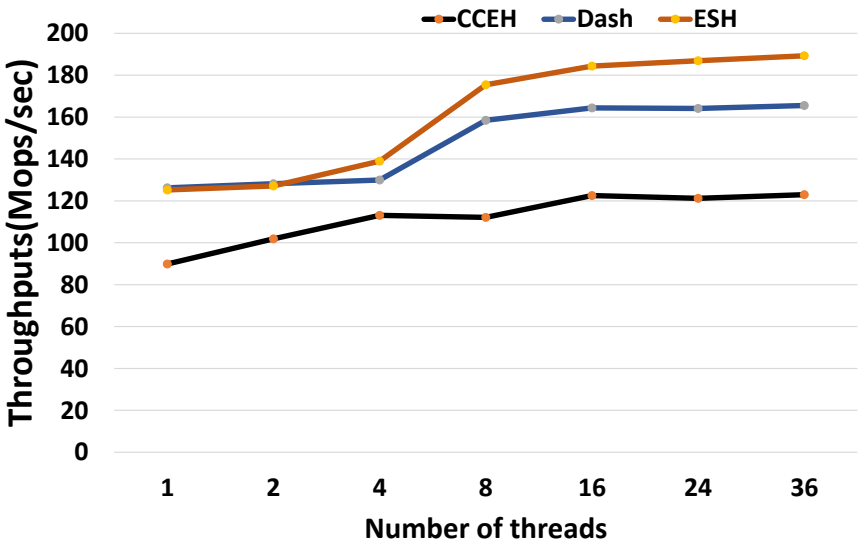


Figure 4.6 Multiple thread performance comparison

This operation will continue until all the buckets in the segment fill up. When the target bucket is already full and the other buckets are not yet exhausted, we check the metadata of each neighboring bucket that could have space to accommodate the record within the same segment. Hence, we move the new value to these buckets, modify the original bucket's OB and the OFRB pointer for later access and store the record within the segment. This helps to effectively utilize all the spaces in the buckets and fill up the segment before we trigger the directory doubling.

Our scheme, therefore, insert records into all bucket in the same segment

and effectively utilize the hash table before it initiates the expensive job of full table rehashing. To do that, we look into the header information of each bucket to get free space and store the new record in the buckets with few key-value stores in them.

This operation involves the rehashing and redistribution of all records to buckets called directory doubling. During the search operation, it starts from the target bucket. If the search key is not matching with the header information, there is no need of looping through the elements in the bucket, it moves to the next neighboring bucket. Moving to the next bucket will be initiated only if the OB flag shows that the bucket has overflowed elements that are moved to the neighboring bucket. Otherwise, the search will end looping in the segment and return the result as no element. But, if there is an overflow indicator OC for a given bucket, the search operation will move to the neighboring bucket and check the header or metadata of each if their search key matches with any elements in the bucket that are identified as moved from another bucket. These elements are identified by the $Membership\ count$ in the metadata. Searching for a record will be unsuccessful when the item is not found in the initially hashed bucket and the neighboring buckets which belong to the same segment. **Search:** Figure 4.7(a) shows the throughput of CCEH, Dash, and EHS during the search operation while varying the number of threads. As shown in the figure, EHS improves the search by 8.3% and 2x compared with Dash and CCEH, respectively, in the case of 36 threads. This improvement comes from the early metadata information about each record in the buckets. Additionally, it is because utilizing free space in all the buckets in a segment helps to reduce the number of buckets to be searched. Also, EHS first checks the overflow bit per bucket to perform the search operations.

It helps to search the moving record even if the record is stored in the

neighboring buckets. Thus, it demonstrates that delaying rehashing in EHS can also improve the search operation by reducing the number of buckets. Once the search starts from one bucket, it checks the header of the bucket if the value exists. If the value is not in the bucket and if there is no overflow $OFRB$ indicator, it will reply from the same bucket, otherwise, it will continue searching from the location at the pointer address of $OFRB$ in the same segment. The experiment result shows our mechanism uses these headers at the bucket level paid off.

Delete: Figure 4.7(b) shows the throughput of CCEH, Dash, and EHS during the delete operation on a varying number of threads. As shown in the figure, EHS outperforms CCEH and Dash by 10% and 3.2%, respectively. This result is similar to that in the search operation since the delete operation accompanies the search operation. Thus, it demonstrates that the search operation in EHS can accelerate the delete operation.

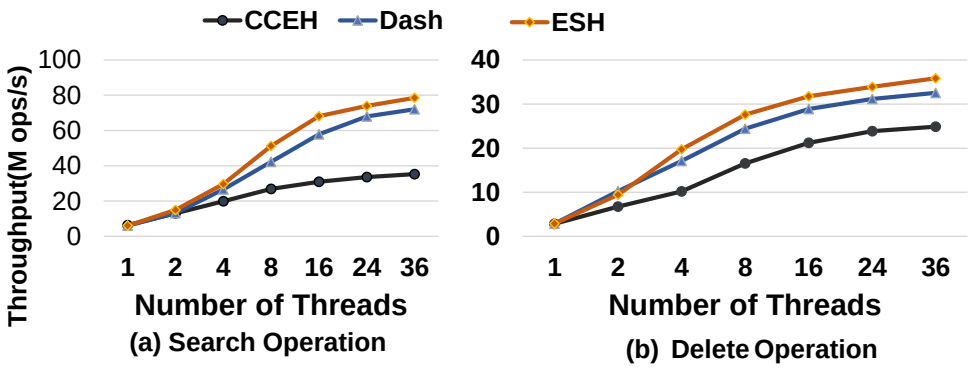


Figure 4.7 Throughput Comparison for varying number of threads

4.4.4 Benefits of Metadata

The performance gain and scalability of ESH come from the effective utilization of metadata or header at the entry of the bucket that greatly helped to reduce data access. This is shown by comparison with other state-of-the-art schemes that use similar hashing schemes on a range of data sizes by varying numbers of threads. The improved throughput in section 4.4.5 for insertion, search, and delete operations with varying threads shows the use of metadata helped to stop searching operations as information about the bucket is accessed from the metadata. This reduces the PM access and unnecessary time spent searching records in the bucket. Insertion operation checks the OB and $OFRB$ metadata elements to the information about the status of the bucket and deletes operations also check the OC before going into the records in the bucket which reduces the response time.

4.4.5 Concurrency

In a multi-threaded environment, performing multiple queries is challenging as multiple threads are going to access a hash table. Especially, expensive operations like full table rehashing require exclusive access to the entire hash table that could block other subsequent operations resulting in to increase in response time. This becomes critical when the size of the hashing table increases. Therefore, we evaluated the latency of concurrent operations like insertion and search operations. As we run a large number of insertions in a multi-threaded environment, the insertion throughput of all schemes under comparison increased and both Dash and ESh showed better overall performance. ESH has shown better performance as the lock contention is reduced to a single bucket level whereas Dash locks a segment for operations.

4.4.6 Scalability

From Figure 4.6, we see the scalability of the insertion operation of hashing schemes under a varying number of threads. When the size of hashing key increases, ESH shows better scaling compared to the other schemes. Hence, ESH is more scalable than CCEH by 34% and Dash by 12% for insertion operations. The scalability of Dash and ESH was not significant until the threads are more than four. But it shows better scalability when the threads increase to almost the same as the number of cores in the system and there is no big change after that. For search operations, Dash and ESH show near-linear performance but CCEH falls behind because the locking and cache line level hashing is affected by the size of the hash table that requires many PM writes. All three schemes use similar extendible hashing but the difference in lock strips [60] makes locking fit into the CPU cache. The effect of the building blocks of PM on log writing and block flushing [61] also affects the write bandwidth. ESH showed higher insertion, search, and delete performance and scaled with varying threads that come as the result of the modifications that delayed the full table rehashing by exhaustively utilizing the allocated hash table. Additionally, optimistic locking at the bucket level helps give access to other threads to access other buckets which contributed to reducing the response time for each thread.

4.4.7 Load Factor

Limiting the number of bucket in a single segment to four helps to minimize the number of total buckets in a single segment. This helps to achieve a better load factor as the ratio of the number of elements stored in each bucket to the total number of positions available in the segment will be minimal. The hash table with the best memory efficiency is therefore the one with the highest load factor.

As operations like insertion and search are done at the segment level when one of the buckets is full, having an increased number of buckets in a segment is not advantageous. Having a limited number of buckets in a segment help to get a better linear probing result. To exhaustively store records in the hash table, ESH uses a balanced insertion of records by moving to the neighboring buckets that improve the load factor. In contrast, when the segment size or bucket size increases, it reduces the directory size with the expense of a reduced load factor.

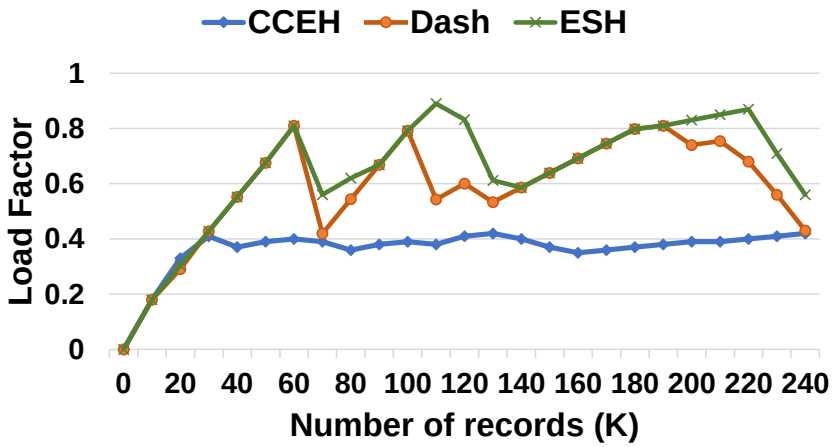


Figure 4.8 Load factor with respect to the number of items inserted into the hash table

Figure 4.8 shows the load factor changes for a varying number of insertion and measure the load factor as different numbers of records are added to the hash tables. CCEH shows a relatively stable result as it performs a split of a segment after four cache line probing. This is also witnessed [14], long probing lengths increase load factor at the cost of performance, yet short probing lengths lead to premature splits. The load factor of Dash and ESH is higher as the change in design paid off. We can see a better load factor on ESH compared to Dash because we delayed the segment split and directory doubling operation

by storing records to all spaces in the neighboring buckets before triggering the split or directory doubling operation. On the Experiment result, we see there is "valley" like shapes that happens because that is where the segment split or directory doubling resulting in rehashing happens. As a result, we observed ESH achieved up to 91% load factor which is higher compared to the state-of-the-art schemes.

4.4.8 Recovery

Recovery is considered one of the features of a system that help to get the data to its normal state when failure happens. A system may fail because of many reasons. A persistent hash table also crashes due to power failure or other unknown causes that result in service unavailability or slowdown. We used a similar testing method that Dash deployed for a fair comparison. This is done by loading some records for some time and then killing the process that runs the loading operation then, measure the time required for the scheme to accept another incoming request. The result in Table 4.1 shows that both ESH and

Table 4.1 Recovery time in (ms) comparison with respect to data size.

Hashing Schemes	Record size in GB				
	1	5	10	20	30
CCEH	50	121	256	503	1082
Dash	62	63	65	65	65
ESH	62	65	66	66	67

Dash took less than a second time even for large data sizes where CCEH took more time compared to the other competitors and also scales up as the data size increases. This result is achieved because Dash used a clean marker and

ESH used a ”state” marker that checks whether the system was shut down clearly in a prior operation. Accordingly, a recovery operation always involves reading this flag and recovering to the consistent state stored on PM before. We observed that ESH has shown, it needs more time to recover compared to Dash which is negligible even for large records. This happened because ESH works per bucket while the other considers a segment as one recovery block. The other consistency issues during multi-threaded operation are maintained by the ”lock” signal included in the metadata.

4.5 Summary

In this work, we presented a scheme that uses available free space in the neighboring buckets to store records. Delaying unnecessary directory doubling operations is achieved by exhaustively utilizing the existing spaces in the neighboring nodes. This can help to effectively utilize the persistent memory space. The performance of the scheme is also better compared with the state-of-the-art scheme for uniform and skewed data distribution. Therefore, our scheme can be an alternative approach that improves the extendible hashing by reducing the split management overhead that eventually postpones the expensive operation of directory doubling in the system that uses an extendible hashing scheme on persistent memory.

Chapter 5

Conclusion

Efficient utilization of memory is critical in all applications. In specially, on memory-intensive applications, effective utilization of memory is a matter of no choice. As the memory development trend is changing, the storage systems need to also change to fit the new architectures. In this dissertation, we presented a scheme that effectively uses available free slots in the buckets to store records. This mechanism helped to reduce PM access and maximized bandwidth consumption. The scalability test also witnessed that this scheme scales well in multi-threaded environments.

In chapter 3, we proposed and designed a scheme that delays unnecessary directory doubling operations by exhaustively utilizing the existing spaces from neighboring nodes. By doing so, we can effectively utilize the persistent memory space. To get better performances, we proposed a minimal locking approach where a bucket is locked while writing a request allowing other processes to access/read the buckets in the segments. This allows multiple access to achieve more scalability in a multi-threaded environment. The performance

of the scheme is also better compared with the other scheme for uniform and skewed data distribution.

in Chapter 4, we extended our scheme to again fit the characteristics of Optane Persistent memory for its read block size. Applying the improved performances in chapter 3, we redesigned the bucket layout to store more records in a single bucket and minimize unnecessary PM access. We extended the bucket size to 256B to fit the PM read block. This is advantageous in minimizing the number of reads and effectively using the block of PM. To effectively manage the records in the bucket, we designed the metadata that stores information about the buckets. This metadata information is used to access records faster as the size increases.

The performance of this scheme is also better compared with the state-of-the-art scheme for uniform and skewed data distribution. The evaluation results show that our scheme can be used as one of the schemes that enhanced the extendible hashing schemes for in-memory applications.

In future work, we will evaluate the scheme for a range of applications and different workloads for real scenarios to enrich the mechanism. Based on the results, we also have a plan to add more tests and peer evaluations that help to get a general framework that can handle hashing for a range of applications.

Bibliography

[1] Intel, "Intel® optane™ dc persistent memory," 2021.

[2] I. Corporation, "Intel® optane™ dc persistent memory," 2022.

[3] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "Bztree: A high-performance latch-free range index for non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[4] S. Chen, P. B. Gibbons, S. Nath, *et al.*, "Rethinking database algorithms for phase change memory.," in *Cidr*, vol. 11, pp. 9–12, 2011.

[5] Redis, "Redis is an open source (bsd licensed), in-memory data structure store," 2021.

[6] Memcached, "Memcached, free open source, high-performance, distributed memory object caching system," 2021.

[7] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, pp. 187–200, 2018.

[8] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pp. 167–181, 2015.

[9] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "{WORT}: Write optimal radix tree for persistent memory storage systems," in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pp. 257–270, 2017.

[10] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pp. 31–44, 2019.

[11] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 461–476, 2018.

[12] 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*, pp. 462–477, 2019.

[13] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[14] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *arXiv preprint arXiv:2003.07302*, 2020.

[15] F. Xia, D. Jiang, J. Xiong, and N. Sun, "{HiKV}: A hybrid index {Key-Value} store for {DRAM-NVM} memory systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 349–362, 2017.

[16] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 91–104, 2011.

[17] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, pp. 1–15, 2014.

[18] 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*, pp. 1049–1063, 2022.

[19] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "Chameleondb: a key-value store for optane persistent memory," in *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 194–209, 2021.

[20] Intel, "Intel and micron produce breakthrough memory technology," 2021.

[21] I. Corporation, "Enabling persistent memory programming," 2022.

[22] J. Ren, Q. Hu, S. Khan, and T. Moscibroda, "Programming for non-volatile main memory is hard," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pp. 1–8, 2017.

[23] N. Shavit, "Data structures in the multicore age," *Communications of the ACM*, vol. 54, no. 3, pp. 76–84, 2011.

[24] I. Sánchez Barrera, D. Black-Schaffer, M. Casas, M. Moretó, A. Stupnikova, and M. Popov, "Modeling and optimizing numa effects and prefetching with machine learning," in *Proceedings of the 34th ACM International Conference on Supercomputing*, pp. 1–13, 2020.

[25] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: a holistic approach to memory placement on numa systems," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 381–394, 2013.

[26] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß, "kmaf: Automatic kernel-level management of thread and data affinity," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 277–288, 2014.

[27] M. Diener, E. H. Cruz, L. L. Pilla, F. Dupros, and P. O. Navaux, "Characterizing communication and page usage of parallel applications for thread and data mapping," *Performance Evaluation*, vol. 88, pp. 18–36, 2015.

[28] P. Memarzia, S. Ray, and V. C. Bhavsar, "Toward efficient in-memory data analytics on numa systems," *arXiv preprint arXiv:1908.01860*, 2019.

[29] Z. Majo and T. R. Gross, "Memory management in numa multicore systems: trapped between cache contention and interconnect overhead," in *Proceedings of the international symposium on Memory management*, pp. 11–20, 2011.

[30] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 28, no. 4, pp. 1–45, 2010.

[31] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 427–439, 2019.

[32] S. Thomas, R. Hayne, J. Pulaj, and H. Mendes, "Using skip graphs for increased numa locality," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 157–166, IEEE, 2020.

[33] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek, "Cphash: A cache-partitioned hash table," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 319–320, 2012.

[34] I. Calciu, J. Gottschlich, and M. Herlihy, "Using elimination and delegation to implement a scalable numa-friendly stack," in *5th {USENIX} Workshop on Hot Topics in Parallelism (HotPar 13)*, 2013.

[35] H. Daly, A. Hassan, M. F. Spear, and R. Palmieri, "Numask: high performance scalable skip list for numa," in *32nd International Symposium on Distributed Computing (DISC 2018)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[36] M. Dong, Q. Yu, X. Zhou, Y. Hong, H. Chen, and B. Zang, "Rethinking benchmarking for nvm-based file systems," in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 1–7, 2016.

[37] S. Ma, K. Chen, S. Chen, M. Liu, J. Zhu, H. Kang, and Y. Wu, "{ROART}: Range-query optimized persistent {ART}," in *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, pp. 1–16, 2021.

[38] A. Prout, "The story behind memsql's skiplist indexes," *Published on: Jan*, vol. 20, p. 7, 2014.

[39] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pp. 1243–1254, 2013.

[40] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 135–148, 2017.

[41] J. Xu and S. Swanson, "{NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories," pp. 323–338, 2016.

[42] 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.

[43] P. Zuo and Y. Hua, "A write-friendly and cache-optimized hashing scheme for non-volatile memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 985–998, 2017.

[44] P. Zuo, Y. Hua, and J. Wu, "Level hashing: A high-performance and flexible-resizing persistent hashing index structure," *ACM Transactions on Storage (TOS)*, vol. 15, no. 2, pp. 1–30, 2019.

[45] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory," pp. 371–386, 2016.

[46] S. ORACLE, "Architectural overview of the oracle zfs storage appliance," 2018.

[47] S. Patil and G. Gibson, "Scale and concurrency of {GIGA+}: File system directories with millions of files," in *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[48] S. R. Soltis, T. M. Ruwart, and M. T. O'keefe, "The global file system," in *NASA Conference Publication*, pp. 319–342, 1996.

[49] S. Whitehouse, "The gfs2 filesystem," in *Proceedings of the Linux Symposium*, pp. 253–259, Citeseer, 2007.

[50] S. Imamura, M. Sato, and E. Yoshida, "Evaluating a trade-off between dram and persistent memory for persistent-data placement on hybrid main memory,"

[51] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing—a fast access method for dynamic files," *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 3, pp. 315–344, 1979.

[52] Z. Liu, I. Calciu, M. Herlihy, and O. Mutlu, "Concurrent data structures for near-memory computing," pp. 235–245, 2017.

[53] Intel, " *Intel Threading Building Blocks Developer Reference*, `https://software.intel.com/en-us/tbb-reference-manual/`," 2021.

[54] W.-H. Kim, J. Seo, J. Kim, and B. Nam, "clfb-tree: Cacheline friendly persistent b-tree for nvram," *ACM Transactions on Storage (TOS)*, vol. 14, no. 1, pp. 1–17, 2018.

[55] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[56] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.

[57] Intel, " *Intel® Persistent Memory Development Kit*, `https://pmem.io/pmdk/libpmem/`," 2021.

[58] 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)*, pp. 169–182, 2020.

[59] A. Appleby, "Murmurhash - murmurhashunaligned," 2022.

[60] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear, *The art of multiprocessor programming*. Newnes, 2020.

[61] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, "Building blocks for persistent memory," *The VLDB Journal*, vol. 29, no. 6, pp. 1223–1241, 2020.

요약

메모리의 발전은 자료 구조에서의 잠재적인 혁신을 가져왔다. 대용량이고 낮은 지연시간을 보이며 바이트 단위 접근이 가능한 영구 메모리(PM)는 그 이점을 활용하여 대부분의 기존 해시 기반 인덱스들의 변화를 가속시켰다. 이로 인해 많은 새로운 해시 기법들이 에뮬레이터를 통해 제시되어왔지만 최적의 설계를 가지지 못 했고 실제 장치에서의 확장성(Scalable)을 갖추지 못 하였다. 일부 해시 테이블 설계만이 load factor나 확장성(Scalability), 메모리 효율성, 복구 등의 중요한 속성을 다루었다. PM에서 효과적인 해시 기법을 다시 설계하는 것에 어려운 점 중 하나는 해시 테이블에서의 동적 해싱 연산 비용을 줄이는 것이다. 본 논문에서는 PM에서 메모리 효율성, 확장성(Scalable), 성능을 개선하는 효율적이고 확장성 가능한(Scalable) 해시 기법을 제시하며 그것을 ESH라고 부른다. ESH는 해시 테이블의 공간을 효율적으로 사용할 수 있게 해주며 성능을 향상시키기 위해 전체 테이블의 재해시(rehashing)를 늦춘다. 이는 ESH가 할당된 메모리 공간을 효율적으로 사용하게 하여 최대 load factor를 낼 수 있게 한다. 우리는 Intel Optane DC Persistent Memory(DCPMM)을 사용하여 우리의 기법을 평가하고 최신 동적 해싱 기법들과 비교한다. 실험 결과는 ESH가 삽입 연산에 대해 CCEH보다 30%, Dash보다 4% 성능을 개선했음을 보인다. 또한 검색 연산에 대해 Dash에 비해 약 10% 성능을 개선하였고 다른 경쟁 기법들에 비해 최대 91%의 load factor를 달성하였다.

주요어: 영구 메모리, 동적 해싱, Scalable 해싱, In-memory 시스템, Extendible 해싱

학번: 2018-31651