# EA-PHT-HPR: Designing Scalable Data Structures for Persistent Memory

by

Diego Cepeda

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2020

## Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## Abstract

Volatile memory has dominated the realm of main memory on servers and computers for a long time. In 2019, Intel released to the public the Optane data center persistent memory modules (DCPMM). These devices offer the capacity and persistence of block devices while providing the byte addressability and low latency of DRAM devices. The introduction of this technology now allows programmers to develop data structures that can remain in main memory across crashes and power failures. Implementing recoverable code is not an easy task, and adds a new degree of complexity to how we develop and prove the correctness of code.

This thesis explores the different approaches that have been taken for the development of persistent data structures, specifically for hash tables. The work presents an iterative process for the development of a persistent hash table. The proposed designs are based on a previously implemented DRAM design. We intend for the design of the hash table to remain similar to its original DRAM design while achieving high performance and scalability in persistent memory.

Through each step of the iterative process, the proposed design's weak points are identified, and the implementations are compared to current state-of-the-art persistent hash tables. The final proposed design consists of a hybrid hash table implementation that achieves up to 47% higher performance in write-heavy workloads, and up to 19% higher performance in read-only workloads in comparison to the dynamic and scalable hashing (DASH) implementation, which currently is one of the fastest hash tables for persistent memory. As well, to reduce the latency of a full table resize operation, the proposed design incorporates a new full table resize mechanism that takes advantage of parallelization.

**Acknowledgements**

I would like to thank my supervisor, Professor Wojciech Golab, for providing me with valuable observations and recommendations in the development step of this research. I sincerely appreciate the opportunity given to work with him throughout my Master's program. I would also like to thank Professor Trevor Brown and Professor Seyed Majid Zahedi, for their valuable comments and reviews that helped me to further refine this work. I would like to express my gratitude to my family and my fiancee for their unconditional support throughout my Master's degree program. This accomplishment would not have been possible without any of them.

## Dedication

This thesis is dedicated to the one I love.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

During the last decade, there has been intense research towards the development of non-volatile main memories [4, 32, 52]. Foreseeing the introduction of persistent memory devices, there also have been several programming models and software techniques that consider the existence of these devices [11, 14, 17]. The public release of the Intel Optane Data Center Persistent Memory Module (DCPMM) in 2019, came to introduce disruptive changes in the memory hierarchy. DCPMM devices, which we will refer to in this thesis as "persistent memory", combine the capacity and persistence of block storage devices while providing the byte addressability and a slightly higher latency to that of DRAM.

Systems that incorporate DCPMM modules still need a hybrid architecture containing DRAM. When DCPMM devices are configured in memory mode, they behave like volatile main memory, and DRAM is used as a cache. When DCPMM devices are configured in app direct mode, they are shown to the system as independent persistent memory resources, and DRAM is still needed to boot the OS in this case. Still, these persistent memory modules are an attractive option, since, for large capacity modules, the price per GB compared to DRAM devices is much lower. Applications that rely on memory capacity over memory bandwidth and latency will significantly benefit from DCPMM devices. Some examples of these applications may include cloud and Infrastructure-as-a-Service (IaaS) applications, by allowing more virtual machines and cloud containers per server. Another application that can significantly benefit from these new devices is a database. The large capacities and persistence of persistent memory modules will enable databases to maintain their data in main memory rather than in slow secondary storage.

## 1.1 Motivation

Persistent memory up to this date can be considered the "sweet spot" of modern storage. Now in-memory data structures can provide features such as recovery after a crash or power failure. However, at the same time, this adds a new layer of complexity when designing recoverable applications. While persistent memory can maintain data across power cycles, the cache, which is used at runtime by the processor, remains volatile.

Previously, DRAM-based in-memory applications did not care if the data they were storing reached DRAM or remained in the cache since, in the event of a power failure, all data would be lost. In the case of recoverable in-memory applications, developers need to ensure that the data they store reaches the persistent memory device at some specific point during runtime. This is to ensure that the data is safely stored in persistent memory. When we talk about concurrent recoverable code implementations, another layer of complexity is added to the designs. Developers still need to ensure that data reaches the persistent memory device, but now the order of these stores is more relevant to ensure the correctness of these implementations. There has been intense interest [2, 8, 39, 45, 54, 60, 69] in developing techniques that allow developers to use persistent memory devices to create high performing code and take advantage of the persistence of data these devices offer. While prior work on emulated devices is surely relevant towards the design of efficient code for persistent memory, previously made assumptions regarding the latency of persistent memory devices and their access granularity size can now be reevaluated. As well, the impact of these assumptions on the performance can now be measured.

Hash tables are fundamental data structures that provide expected constant-time lookups and insertions on individual items. Many data-intensive systems such as key-value stores [5, 7, 35, 49, 53], and databases [16, 18, 42, 43] use hash tables as a building block of their designs. Prior research on the design of persistent hash tables includes both work based on emulated persistent memory [39, 45, 60, 69], and based on DCPMM devices [2, 8, 54].

A full table resize operation is used to dynamically grow the hash table implementation to store more key-value pairs. Often, a full table resize function is implemented by using a global lock to restrict changes to the state of the hash table and the copying of the key-value pairs to a new larger table. Most of the hash table designs for persistent memory tend to evade the full table resize of the hash table since these operations incur a large number of stores to persistent memory. Often, these new designs are complex compared to DRAM implementations since they need to ensure their data structure is recoverable and, at the same time, provide new mechanisms that avoid the full table resize operation.

In this thesis, we explore the different techniques that had been proposed by these designs alongside their benefits and drawbacks. This research intends to design a simple implementation that resembles a DRAM hash table while achieving high scalability and performance. In contrast to prior works such as [2, 54], which convert DRAM hash table implementations, we only abstract the base of the design of a DRAM implementation, while taking into consideration specific properties of DCPMM devices in our design. In contrast to prior persistent memory implementations such as [39, 45, 8, 69], that look to reduce the number of stores to persistent memory in their operations, we focus on reducing the cost of allocation. While prior work avoids the full table resize operation, we intend to reduce the latency of this operation by parallelization.

## 1.2 Contributions

In this thesis, the iterative process for the development of a persistent hash table is presented, and we make the following contributions:

1. We explore and evaluate the advantages and drawbacks of different design decisions when designing data structures for persistent memory.

2. We present the design of three persistent hash table implementations. The provided implementations surpass the open-source state-of-the-art implementations. The implemented hash tables are tested using a linearizability checker tool and provide linearizable executions in the scenarios tested.

3. When comparing our write-optimized implementation to current state-of-the-art implementations, the hash table is up to 47% faster in a write-heavy workload, and 19% faster in a read-only workload.

4. When comparing our read-optimized implementation to current state-of-the-art implementations, the hash table is up to 22% faster in a write-heavy workload, and 33% faster in a read-only workload.

5. We design a new full table resize mechanism for a persistent hash table. This new mechanism allows multiple helpers to aid in the initialization of the volatile portion of the hash table, and the rehashing of the key-value stores in persistent memory. At the same time, this mechanism allows the volatile and persistent regions to be initialized/rehashed in parallel.

## 1.3 Thesis organization

The thesis is organized as follows: Chapter 2 goes through the overview of persistent memory devices, followed by proposed correctness properties in this line of research. An overview of current techniques used towards the design of persistent memory data structures is described. Finally, the hash table implementation on which the new work is based is explained. Chapter 3 presents the first iteration of a persistent memory hash table. The allocation mechanism, the different operations, and the recovery mechanism are explained in depth. The implementation is compared to other persistent hash tables, and the weak points of the design are identified. Chapter 4 presents the second iteration of the persistent memory hash table. The changes in the design are shown. The hash table is then compared with its predecessor and other persistent hash tables, showing improvements in performance. Chapter 5 presents the last iteration of the persistent memory hash table, which presents a novel resize mechanism for the hash table. The resize mechanism is explained in depth, and the hash table is then compared with its predecessors and the other persistent hash tables. The results show that the final design can outperform all previously shown hash tables in all scenarios considered. Chapter 6 presents the results of linearizability testing for the hash table implementations presented in the previous chapters. Finally, Chapter 7 concludes the thesis and discusses future work.

# Chapter 2

# Background and Related Work

Main memory is used to store programs or data that are currently in use by the processor. These devices are a crucial piece of hardware in modern computer systems. Main memory, often called Random Access Memory (RAM), offers byte addressability in contrast to secondary storage memory, which takes a block-based approach to access data. While initially main memory was non-volatile, throughout the last decade, main memory devices have predominately remained volatile. There have been different approaches [4, 19, 32, 52] to develop modern non-volatile memory, which try to find a balance between cost, density, reliability, and endurance. In 2019, Intel released to the public the Optane Data Center Persistent Memory (DCPMM) modules, attracting serious attention to the integration and use of non-volatile main memory in modern computer systems.

## 2.1 Overview of DCPMM devices

DCPMM devices share properties with both DRAM, and secondary storage. Similarly to DRAM, DCPMM devices sit on the memory bus, are byte-addressable, and are faster than secondary storage. In contrast to DRAM, these devices have higher latency, lower bandwidth, and asymmetric load and store performance [28]. Another difference between DRAM and DCPMM modules is their access granularity, which is 256 bytes, meaning that 64 byte loads/stores will translate into larger 256 byte accesses [28]. It is important to note that loads and stores that are smaller than this granularity waste bandwidth as they have the same latency as a 256 byte access [28]. Like secondary storage devices, DCPMM can maintain data across power cycles and offers modules with large capacities. DCPMM modules are offered in 128G, 256G, and 512G sizes, notably larger than DRAM

devices. The high density of these devices makes them an attractive option for systems that rely on storing large amounts of data in main memory during run-time. Not all current processors support DCPMM devices since they communicate via a special protocol with the processor's integrated memory controller (iMC). This protocol allows for variable-latency memory transactions. In this thesis, an Intel Xeon Gold 6230 CPU is used, which is included in the list of CPUs that support DCPMM devices.

DCPMM modules can work in two modes: memory mode and app direct mode. Memory mode is used to expand main memory capacity, by using DRAM as a direct-mapped cache for persistent memory. It is important to note that this mode does not offer persistence. As previously stated, this configuration will not be relevant to the focus of this thesis, and further clarification of this mode will not be discussed. A more detailed explanation of this mode can be found in [28]. App direct mode, on the other hand, exposes the DCPMM device as a distinct persistent memory device. This mode allows persistent updates to bypass the kernel and file system, enabling developers to create applications that explicitly control writes into DCPMM modules.

## 2.2 Consistency and persistency

The addition of persistent memory devices to the memory hierarchy does not mean that data is instantly persisted when writing to these devices. The current memory access model involves the cache, an intermediary in data transfer between the persistent memory devices and the Central Processing Unit (CPU). The Intel memory access model is shown in Figure 2.1 below, specifying when data is safe in case of a power failure. The cache memory is volatile and will probably remain this way in the near future since the cache's size is too large to atomically flush during a power failure. Intel processors support a feature called asynchronous DRAM refresh (ADR). In the event of a power failure or a shutdown, the ADR triggers an interrupt in the processor's memory controller, which flushes the data buffers in the iMC [58]. This is crucial to deal with program crashes and power failures since it is ensured that data that reaches the ADR domain will be persisted. Flushing instructions are used to ensure the data reaches the ADR domain.

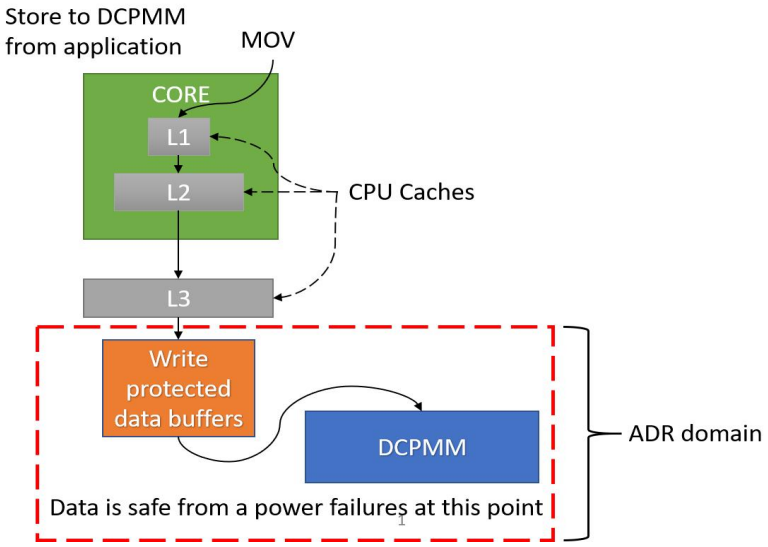


Figure 2.1: Intel memory access model.

It is known that processors are free to reorder memory operations, which can be problematic when a specific ordering of stores or loads is required. To address this, fence instructions can be used. These instructions impose ordering constraints on how operations are performed. The following described fence and flushing instructions are part of the Intel x86 instruction set [9].

- SFENCE: guarantees that all previous stores before the fence are globally visible before any store after the fence [9].
- LFENCE: guarantees that all previous loads before the fence are performed prior to any load after the fence [9].
- MFENCE: guarantees that all previous stores and loads before the fence are globally visible, meaning that the value to be loaded into its destination register is determined, before any store or load after the fence [9].
- Cache line flush (CLFLUSH): flushes a single cache line to memory and invalidates the cache line. This instruction is serialized, which means that any modifications to flags, registers, and memory for previous instructions are completed before the next instruction is fetched and executed [3].

- Cache line flush optimized (CLFLUSHOPT): newly introduced instruction for DCPMM support, same as CLFLUSH but without the serialization.[3].
- Cache line write back (CLWB): newly introduced instruction for DCPMM support, flushes the cache line to memory, but the cache line can remain valid in the cache [3].

An atomic store guarantees that in the event of a failure, the store will either take effect entirely or not at all. The previously shown flush instructions do not guarantee that stores will be atomic in case of failures, which can lead to corrupted data. To guarantee an atomic store, the data being written must be an 8-byte aligned store [56]. It is important to note that atomic stores alone are not sufficient to ensure the correctness of a data structure. As mentioned earlier, instructions can be reordered by the processor. Using a combination fence instructions and flushes is a viable way to ensure the proper ordering of stores.

## 2.3 Correctness properties for recoverable systems

Introduced by Herlihy and Wing, linearizability [40] has been widely used as a correctness condition for concurrent data structures. The linearizability model requires that a process finishes its operation before invoking the next operation and that an operation takes effect instantly at some point between its invocation and response step, called the *linearization point*. This correctness property provides an "illusion" that operations on shared objects appear to happen in a sequential manner. Under the linearizability model, pending operations may take effect at any time in the future. This can be troublesome for implementations where recovery from a crash is possible with the use of persistent memory.

Strict Linearizability [38], states that in the event of a crash or power failure, pending operations can only take effect before the crash, meaning that either the operation took place or it did not. For operations that were able to take effect but did not manage to produce a response event, the *linearization point* of the operation happens just before the crash. If the operation did not manage to take effect before the crash, this means it will never take effect. Persistent atomicity [48], proposes that the operation either linearizes or aborts before any later operation by the pending process on any object. Recoverable linearizability [46] proposes that the operation either linearizes or aborts before any later operation by the pending process on that same object. It is important to note that given the previous condition in this model, processes may have multiple pending operations at a time following a failure. Durable linearizability [27] considers all processes to fail together as a "system crash", assuming we are referring to an individual system in which a crash

or power failure is known to terminate all processes associated with the program. Thus it can be safely assumed that later accesses will be performed by different processes. It was found that under this assumption, persistent atomicity and recoverable linearizability are identical. Durable linearizability specifies that operations become persistent before they return. In case of a crash, all previously complete operations remain complete and visible. Pending operations may or may not have taken effect. Pending operations may have taken effect and crashed before issuing a response, or a later process can finish their execution.

## 2.4 Persistent memory allocation

To handle persistent memory, currently, there are two options to choose from, using multi-purpose persistent memory allocators such as the libpmemobj allocator [23], Ralloc [65], NV-Heaps [26], and Makalu [31], or creating custom allocation techniques. Multi-purpose persistent memory allocators can safely allocate any requested size in memory and have recovery mechanisms to handle crashes and prevent memory leaks. While this is a convenient feature for programmers, these techniques often require the persistence of extra metadata to allocate data securely. In contrast, custom allocation techniques leave the proper handling of persistent memory and recovery mechanisms entirely to the developer, and while this can be complicated, it allows creating tailormade and fast allocation mechanisms.

## 2.5 Data structures for persistent memory

Before the release of persistent memory to the public, in-memory data structures that needed to maintain data in the event of a crash or power failure had to store the data in secondary storage devices such as an HDD or SSD. These operations have a considerable overhead given the difference in performance between main memory and secondary storage. Often, checkpointing techniques [33, 41, 67] are used to reduce the cost of persisting data. The arrival of persistent memory modules has changed the spectrum of what in-memory data structures can do. Now logless fast recovery is an option where the data structure can execute a recovery function to remove any inconsistencies left in the memory after a crash, in contrast to log-based checkpointing techniques which need to redo or reload pending operations.

Since persistent memory modules offer large capacities and persistence, there has been growing interest in building data structures that offer efficient and reliable recovery, while maintaining scalability and performance. Previous work on developing data structures for

persistent memory has explored different data structures such as trees [25, 51, 50, 12, 66, 55, 20, 57, 30], queues [36] and hash tables [68, 8, 69, 39, 13, 15, 44, 45]. This thesis will focus on hash tables, specifically dynamic hash tables, which are a fundamental building block of many data-intensive systems such as key-value stores [5, 7, 35, 49, 53] and databases [16, 18, 42, 43].

Research towards the development of hash tables for persistent memory can be grouped into three different approaches. Some work focuses on converting existing DRAM data structures, including hash tables, to work with persistent memory. These approaches either develop software mechanisms to convert volatile data structures into persistent and recoverable structures [60, 2, 13], or specify a principled approach that defines a set of conditions that need to be met by the DRAM implementation in order to be converted into a persistent memory recoverable data structure [54]. The other branch of research focuses on redesigning and creating hash tables for persistent memory from "scratch." This approach implies the redesigning of the whole data structure based on the specific hardware characteristics of persistent memory modules. Inside this branch of research, we have seen two "sub-branches" on the design of these structures.

There is research on hybrid implementations [15, 68, 69], which keep some part of the data structure in volatile DRAM, commonly the indexing part of the data structure, and maintain the necessary data, commonly the stored values and metadata for recovery, on persisting memory. The other sub-branch keeps all the data structure in persistent memory [8, 39, 44, 45], giving special attention to the number of writes, reads and flushes needed to ensure recoverability and performance.

Some of the key differences among these implementations are the recovery mechanisms. For hybrid approaches, it is a fact that everything that is on DRAM will be lost in case of a power failure. Some implementations, such as Flat Store [68], implement checkpointing mechanisms to copy the volatile part of the data structure to the persistent domain. In this case, during normal restart, these data structures can reload the volatile segment from persistent memory, and in case of an unexpected crash or power failure, they perform a recovery function to restore the volatile segment of the data structure. While these approaches sacrifice some of the recovery time, they aim to improve failure-free performance by only saving necessary data to persistent memory. In contrast, designs that keep all the data structure on persistent memory employ different recovery approaches. Some will scan the data structure after a crash, in search of inconsistencies that need to be fixed prior to starting execution. For example, CCEH [39] follows this approach. Others implement specialized recovery techniques that allow the data structure to recover during execution. This is done by detecting inconsistencies on the fly and fixing them prior to continuing the requested operation, which is the case in DASH [8]. In Table 2.1, the recovery performance

for different implementations is shown; hybrid designs, in contrast with those that keep all their data in persistent memory, are orders of magnitude slower. When fast recovery is necessary, hybrid implementations are not ideal.

| Implementation | Type | M key-value pairs/s |
|---|---|---|
| DASH | All in PMEM | Constant 57ms |
| CCEH | All in PMEM | 2169 |
| Flat Store | Hybrid | 25 |
| HiKV | Hybrid | 0.57 |

Table 2.1: Recovery time performance comparison.

### 2.5.1 Conversion approaches

RECIPE [54] is a principled approach for converting concurrent DRAM indexes into crash-consistent indexes for persistent memory. This approach defines a set of conditions, each for a different type of data structure, that must be met in order to perform a successful conversion. The conditions are as follows:

- **Condition#1:** reads must be non-blocking, writes may be blocking or not, and the index performs write operations visible to other threads using a single hardware-atomic store.
- **Condition#2:** reads and writes must be non-blocking, write operations are performed in an ordered sequence of hardware-atomic stores, reads can tolerate inconsistent states, and writes can fix inconsistent states via a helping mechanism.
- **Condition#3:** reads must be non-blocking, writes must be blocking, writes are performed in an ordered sequence of hardware-atomic stores. Both reads and writes can tolerate inconsistencies.

For all three conditions, the conversion involves adding a cache line flush and a memory fence after each store. Only for condition three is additional code on the write function needed to detect and fix permanent inconsistencies.

The work by David et al. [60] proposes a set of techniques crafted for lock-free data structures, with the purpose of removing the need of logging for recovery entirely. These techniques include:

- **Link and persist:** allows the atomic change and persistence of a link in a data structure.
- **Link cache:** allows persisting batches of modified links.

The link and persist technique uses pointer marking to signal the current link might not be persisted. This ensures the current operation is the one persisting the link, or other operations will be able help persist the link. This technique is similar to the "dirty" bit technique used in [64]. The link cache technique uses a volatile hash table to store the links that have not been durably persisted to memory. When the durable write of one of the links is needed for correctness, the entire batch is then persisted.

In contrast with the previous two approaches, Pronto [2] is a library designed to abstract the complexity of adding persistence to data structures. This library allows programmers to convert existing volatile implementations to persistent ones with ease. To do this, Pronto uses asynchronous semantic logging (ASL), where all operations invoked in an object, along with their arguments, are recorded. Logging happens in parallel with the execution of the operation. A disadvantage of this approach is the need for background threads that perform the logging, which decreases the number of available cores in the system. While the previous two approaches are designed for specific types of data structures, Pronto offers a generic approach that applies to many data structures.

### 2.5.2 Hybrid data structure approaches

The approach taken by FlatStore [68] consists of decoupling the index from the actual key-value store. They do this by saving the key-values in a persistent log structure while maintaining a volatile index in DRAM for fast indexing. In this approach, operations that modify the data structure, such as insert and delete, must write the metadata and key-value item to persistent memory, then proceed to attach it to the end of the log and update the volatile index. For get operations, the volatile index is used to find the exact log entry and then read this log entry. This implementation, in comparison to approaches that store the whole hash table in persistent memory, aims to avoid the cost of splitting/resizing and enable log entries to be efficiently searched at run time without the need to traverse the log.

Bucket Hash [69] takes a similar approach by also decoupling the index from the actual key-value stores. In this approach, the key-value stores are saved in an ordered doubly-linked list structure. To locate entries, they propose a new indexing structure called multi-level lookup tables. This structure is comprised of several lookup tables that can point to

another lookup table or a bucket. Given the properties of this design, rehash operations occur only on buckets and do not need to perform costly full table rehashes. It is also important to note that given that the key-value pairs are stored in an ordered manner across buckets, the overhead of range queries is reduced compared with traditional hashing schemes.

On the other hand, HiKV [15], combines two index structures, maintaining a partitioned hash index in persistent memory and a global B+-Tree index in volatile memory. For operations that modify the data structure, such as insert and delete, the operations are performed on the persistent memory hash index, and the B+-Tree index is updated asynchronously. This reduces the latency of these operations. This approach allows porting the properties of B+-Trees and improves the performance of scan operations, while retaining the efficiency of single key operations, such as put or get.

### 2.5.3 Persistent memory data structure approaches

In this branch of work, the implementations try to make the least amount of stores to persistent memory. As a consequence, most of the hash table designs provide different approaches to avoid a full table rehash operation.

One of the earliest works introduced Path Hashing [44]. This approach presents a technique called position sharing for dealing with hash collisions. This technique involves the structuring of the storage cells of the hash table to be organized as an inverted complete binary tree. In this design, all the leaf nodes are the ones that can be addressed by the hash functions, while all the other nodes are considered as shared standby positions to deal with hash collisions.

Another technique introduced was Level Hashing [45]. The Level Hashing technique introduces a two-level structure comprised of a top-level, where buckets can be addressed by the hash functions, and the bottom level, which is used to provide standby positions for any collisions. Their resizing technique takes an interesting approach, where they add a new level. The new level is then placed on top of the previous top-level. Then the resize operation only involves the rehashing of all the buckets on the lowest level, meaning the resize will involve only 1/3 of the buckets in the hash table.

The next approach is CCEH [39], which adapted the original extendible hashing technique [47]. The main motivations in their work involved avoiding a full table rehash. They state that this is an expensive operation to perform on persistent memory. Moreover, they intend to minimize the cache line accesses and satisfy recoverability without the need for

explicit logging. This technique presents a three-level structure comprised of a global directory which points to segments, and buckets. In this implementation, segments have a defined number of buckets, which can be addressed by the lower bytes of the hash code, while the upper bytes are used to determine the directory entry to the segment to be addressed. This technique avoids full table rehashing by first trying to perform split operations on the segments. In the case a split can not be performed, the resize function is triggered. Given the design of CCEH, the table resize only involves the creation of a new directory with updated pointers to the segments, eliminating the cost of rehashing all stored values.

One of the newest hash table algorithms is DASH [8]. This work proposes a holistic approach to building dynamic and scalable hash tables for persistent memory, proposing a set of design principles and load balancing techniques to create hash tables for persistent memory.

The design principles of DASH are as follows:

- **Avoid unnecessary reads and writes to persistent memory:** this is done with the intention of conserving bandwidth and alleviating the impact of high end-to-end read latency.
- **Lightweight concurrency:** simple concurrency control to reduce overhead, avoiding things such as read locks for search operations.
- **Full functionality:** not sacrificing important features such as instant recovery or variable-length keys.

Dash proposes a custom bucket layout, which includes a metadata section to optimize the probing and the load factor of the table. This metadata includes a version lock (for concurrency control), allocation bitmap, membership bitmap, counter, and a fingerprint. The counter is used to store the number of key-value pairs stored in the bucket. The allocation bitmap indicates which slot stores valid key-value pairs. The membership bitmap is used to determine if the current key-value pair belongs to the bucket or if it was placed there by one of their load balancing approaches. The fingerprint is used to reduce unnecessary probing of key-value pairs in its different operations. In their results it was noted that the fingerprinting technique dramatically benefits negative search operations by only requiring the checking of fingerprints and avoiding probing.

To balance the load of the buckets, DASH proposes a set of techniques:

- **Balanced insert:** Consists of probing the counter metadata in the bucket $b$ and $b+1$, where $b = hash(key)$ and inserting the record on the bucket which is less full.
- **Displacement:** If both $b$ and $b+1$ are full, the operation will try to move a record from $b+1$ to $b+2$. If this is not possible, it will try to move a record from $b$ to $b-1$. To identify which records were to be originally inserted to the bucket, the membership bitmap metadata is checked, and this accelerates the displacement.
- **Stashing:** As a final resort, if after a balanced insert and displacement, the record cannot be stored, it will be placed in a stash bucket.

For concurrency, Dash uses an optimistic bucket level locking scheme, where insert operations need to acquire the lock to access the bucket. It is important to note that the lock is only one bit, while the remaining bits store the version number. This technique allows the insert operation to release the lock and update the version number atomically. The version number is used by read operations to make sure they are reading the correct values in the bucket. Search operations take a snapshot of the version number and will verify the snapshot matches the value held in the version number to make sure no concurrent write operation has modified the bucket.

## 2.6 Cache line hash table

The work presented further in this thesis extends the cache line hash table lock-based version (CLHT-LB) [62]. Throughout this section, an overall description of CLHT-LB will be given. CLHT-LB was designed based on the asynchronized concurrency (ASCY) paradigm [61]. This paradigm presents four programming patterns in order to develop portable and scalable implementations of concurrent search data structures (CSDSs). The patterns presented by ASCY persuade developers into designing CSDSs that resemble their sequential implementation counterparts. It was identified that some existing CSDS algorithms already follow some of the proposed design patterns. While following some of the design patterns of ASCY will benefit the performance of CSDSs algorithms, they have also identified that to achieve even better performance, the collective use of these design patterns is needed. The patterns proposed by ASCY are the following [61].

- **Design pattern 1:** The search operation should not involve any waiting, retries, or stores.

- **Design pattern 2:** The search phase of an update, locating an empty slot in case of an insert or locating the key to be removed for a remove operation, should not perform any stores other than for cleaning-up purposes and should not involve any waiting, or retries.
- **Design pattern 3:** An update operation whose search phase is unsuccessful should not perform any stores, besides those used for cleaning-up.
- **Design pattern 4:** The number of memory stores in a successful update should be close to those of a sequential implementation.

The implementation of CLHT-LB was initially developed in C and is comprises three structs which serve as the building blocks of the hash table. In this thesis, these building blocks are identified as three types of items: the bucket item, the hash table item, and the root item. Below in Figure 2.2 we can observe a simplified graphical representation of CLHT-LB.

CLHT-LB implements a three-layered structure. The root item serves as the entry point to the data structure; it holds a pointer to the current in-use hash table and contains the global lock used for resizing. The hash table item contains information of itself, such as a pointer to the current contiguous array of bucket item, the size of the table, a threshold, which determines how many overflow buckets can be placed before triggering a full table resize, an overflow bucket counter, and variables which are used to coordinate threads during a full table resize. The bucket item is used to store the key-value pairs; these contain a lock used for concurrency control, three key-value pairs, and a pointer to an overflow bucket in case the current bucket is full. While the implementation does not directly support variable-length keys, storing pointers to variable-length keys can add this functionality. This approach has been used by several other implementations such as [8, 20, 25, 39, 45]. To avoid multiple cache line transfers, the size of the bucket item is fixed to that of a cache-line. It is also important to note that only the locks contained in the bucket item in the contiguous array are used for concurrency control. If a bucket contains a linked overflow bucket when the lock is acquired, the whole chain would be locked.

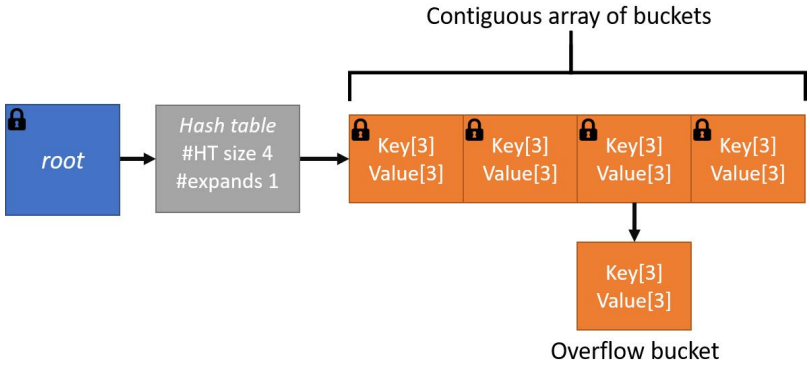


Figure 2.2: Simplified CLHT-LB structure.

The search operation will not wait for other operations; hence it will not acquire a lock, and instead performs the search operation without the need for synchronization. Instead of naively traversing the key-value pairs, throughout the traversal of the key-value pairs, the search operation takes an atomic snapshot of the value of each key-value before checking for a key match during traversal. After a matching key is found, the snapshot then is compared to the current value held at the key-value pair. This ensures that if a key is found, the value returned will correspond to that of the key and not of a concurrent modification by other operations. It is important to note that this requires that the same value cannot be reused by a concurrent operation throughout the lifespan of the search. If the previous condition is not met, a search operation may read a value *val*, compare the key and return *val*, where the value *val* was removed and re-inserted concurrently during the search operation. This restriction in the search operation will lead to linearizability violations in cases where concurrent operations are using the same value. This problem can be fixed, for example, by using version locking, as used in DASH [8]. After the traversal of the key-value pairs in the current bucket has completed, the search operation will proceed to access the next linked bucket if it exists.

The insert operation accesses the current bucket where the key-value pair is to be inserted and searches if the current key is already present in the bucket. If the key is present, the operation returns false. In case the key was not present, then the lock of the bucket is acquired, and the key-value pairs are traversed. During the traversal, if the key is found, the operation will return false, or else it will keep track of an empty location to place the key-value pair to be inserted. Inserts order their writes using SFENCE instructions and will first write the value and then the key; this guarantees that any concurrent search will only find the key-value pair when both the key and value are present in the bucket. In case the bucket is full, a new bucket will be allocated. The key-value pair will be written to the new bucket, to then be atomically linked using a compare-and-swap (CAS) operation on the current full bucket next pointer. This guarantees that any concurrent search will either see an empty pointer or one pointing to a bucket.

The remove operation, in the same way as the insert operation, searches if the current key is present on the bucket. If the key is not found, the operation returns false. If the key is present, the lock to the bucket is acquired, and the key-value pairs are traversed. When found, the key is removed.

To avoid long chains of buckets that would hurt the performance, CLHT-LB keeps track of the number of linked buckets. When the number of linked buckets surpasses a defined threshold, a full table resize is triggered. A global lock is used to provide access to the resizing code. The process that acquires this lock will traverse all the buckets while acquiring their lock in resize mode and rehashing the key-value pairs to the new table. Searches will be unaware of the full table resize and will be able to complete accordingly, while insert and remove operations may still proceed if the current bucket has not yet been locked in resize mode. CLHT-LB implements a helping mechanism where if an insert or remove operation finds the current bucket is locked in resize mode, it will aid in the resizing of the table if possible, or wait for the resize operation to finish.

## 2.7 Summary

The literature around the development of data structures for persistent memory can be summarized into three different approaches. One of them is the conversion of previous DRAM implementations. Often these approaches work on mature codebases, which can guarantee the correctness of their implementations by providing just a conversion methodology to follow. Still, these conversion approaches sacrifice some of their performance, given that these designs were optimized to work with DRAM instead of persistent memory.

Then, we have the tailormade approaches which focus on minimizing the number of stores to persistent memory. Designs that store all the data structure in persistent memory focus on providing specialized algorithms that further reduce the stores to persistent memory. These implementations provide a faster recovery time to that of hybrid implementations as shown in Table 2.1. Hybrid designs aim to remove unnecessary data from the persistent domain, which can be later rebuilt or reloaded from secondary storage. These implementations often sacrifice recovery time to gain performance on failure-free scenarios. However, at the same time, both of these approaches can be complex compared to that of DRAM implementations.

We have identified that most of these approaches use multi-purpose memory allocators. While this is convenient for developers, these allocators need to handle crash recovery as well. Thus, they need to write metadata and implement recovery mechanisms that can hurt the performance of the developed data structures by incurring additional stores to persistent memory. These multi-purpose allocators can allocate data of any given size. Thus, their implementation needs to handle all possible cases. An allocation mechanism specifically designed for a data structure can reduce the allocation complexity by knowing a predefined set of data types to be stored. Then, only handling the defined data type allocations will be necessary.

Most of the existing implementations tend to avoid a full table rehash operation, while none of the previously mentioned approaches has explored the benefits of parallelization of a full table rehash operation. The presented work of this thesis intends to reduce the cost of allocation by providing a custom allocation technique, explore the impact of parallelizing the full table rehash operation, and maintain a simple design that is similar to DRAM implementations.

| Implementation | Pros | Cons |
|---|---|---|
| RECIPE [54] | Simple conversion approach that specifies where in the code the addition of fences and flushes are needed. | Only implementations that meet the specified characteristics can be converted. |
| Pronto [2] | Can convert any sequential or concurrent data structure to a persistent memory implementation. | Needs to have background threads to take care of the logging, thus reducing the available threads in the system. |
| HiKV [15] | Ports the properties of trees and improves the performance of scan operations, while retaining the efficiency of single key operations of hash tables. | After a crash, the volatile tree structure needs to be rebuilt, making the recovery slower. |
| Bucket hash [69] | Simple design that maintains a sorted linked list in persistent memory and provides a volatile indexing structure for operations. | After a crash, the volatile indexing structure needs to be rebuilt, making the recovery slower. |
| Path Hashing [44] | Simple design that proposes a hash table built as an inverted tree. | Does not propose a resize mechanism. |
| Level Hashing [45] | Simple design that proposes a two-layered hash table structure. | 1/3 of the buckets need to be rehashed during a table rehash. |
| CCEH [39] | Avoids a full table resize by instead splitting small segments of the table when necessary. | Memory leaks are possible in the event of a power failure during a segment split. Slow PMDK transactions can solve the issue with the addition of overhead. |
| DASH [8] | Uses metadata to reduce the probing and load factor of the table. | The load balancing techniques add complexity to the design. |

Table 2.2: Pros and cons for the different approaches.

# Chapter 3

# Converting CLHT-LB Into a Persistent Memory Hash Table

Throughout this chapter, an adapted design of the lock-based Cache Line Hash Table (CLHT-LB) [62] is presented as the embedded allocation persistent hash table (EA-PHT). This work is an extension of RECIPE [54], which has proven that CLHT-LB meets the requirements of their condition one, mentioned in subsection 2.5.1. Thus this implementation is guaranteed to be crash-consistent if the conversion methodology of condition one is applied. The limitation of the work presented by [54] is that the used library for allocation is libvmmalloc [24]. This library creates a memory pool in persistent memory and uses it as a heap replacement. In libvmmalloc, the allocated pool is reclaimed upon termination of the program or a crash. As stated previously, this implementation would not be able to recover the memory used. Allocators such as the one provided in the libpmemobj library [23], Ralloc [65], NV-Heaps [26], and Makalu [31] allow developers to allocate objects dynamically and prevent memory leaks in case of power failure. Implementations such as DASH and the versions of Level Hashing and CCEH presented in [8] use the libpmemobj allocator for persistent memory. While these allocators for persistent memory are great as general-purpose allocators, they incur additional writes with metadata to ensure their recovery mechanisms are able to recover after a crash or power failure. When designing data structures that aim for performance, the overhead of these allocators can become a bottleneck [34].

A common goal throughout the field of developing data structures for persistent memory is to reduce the number of stores. Approaches such as DASH [8], CCEH [39], and Level Hashing [45] focus on reducing the stores the data structure needs to perform its functions, but they disregard the stores and overhead of the allocators used. For this specialized

conversion of CLHT-LB to EA-PHT, a custom allocation mechanism is embedded into the data structure itself, to further reduce the number of stores to persistent memory. We prove that this approach can outperform current state-of-the-art persistent memory hash tables, which have implemented specialized designs for persistent memory, while still maintaining a similar structure to its DRAM counterpart.

## 3.1 Memory-mapped files on PMEM

Libraries such as libpmemobj [23] and libpmem [21] allow developers to create memory-mapped files in persistent memory, which are then used to store and retrieve data. The libpmemobj library offers a set of functions that allow developers to allocate objects dynamically and perform transactions in a fail-safe manner, among other functionalities. The libpmemobj library abstracts the complexity of ensuring consistency for data-structures placed in persistent memory in the event of a crash or a power failure. Implementations such as DASH and the adapted versions of level-hashing and CCEH presented in [8], use libpmemobj as the base for handling persistent memory.

In contrast, libpmem offers low-level persistent memory support, which means ensuring consistency of the data placed in persistent memory, and the allocation is left to the developer. For this specific design of EA-PHT, it is assumed that the memory-mapped file is exclusively used to place data that comprises EA-PHT and that no other data structure has access to this file. While it is possible to have multiple instances of EA-PHT, each one requires an independent memory-mapped file. To handle persistent memory in this implementation, the libpmem library [21] is used. This means that mechanisms to ensure data consistency need to be developed for this implementation. Further explanation and details on how this is achieved are described in the following sections.

## 3.2 Offset placement in memory-mapped files

As mentioned in the previous section, libpmem [21] only offers low-level support for persistent memory and provides no allocation functions. The function *pmem_map_file()*, allows the developer to create a new memory-mapped file or reload a previously created one based on the given parameters. Every time a memory-mapped file is created or loaded, a different virtual address space mapping is provided. Storing pointers then becomes a problem; the change of the virtual address space will make all previously saved pointers invalid, which will likely cause a segmentation fault in the program.

To avoid invalid pointers across executions, saving offsets is a viable solution, and this technique has been used in allocators for persistent memory such as Ralloc [65] and NV-Heaps [26]. Instead of using pointers, the offsets in relation to the beginning of the memory-mapped file can be saved. To get a direct pointer, the offset needs to be added to the virtual address that points to the beginning of the memory-mapped file. This way, we can ensure that the items allocated on persistent memory can be located across executions.

## 3.3 Classification of CLHT-LB building blocks

Before deciding where and how the items of EA-PHT are going to be placed in persistent memory. A further understanding of how CLHT-LB allocates its items is needed. In CLHT-LB, the allocation of the root item is performed by a single thread upon the creation of the data structure. The hash table item requires dynamic allocation during resizing, which is performed by a single thread. The bucket items are allocated in two different scenarios: as a contiguous array of buckets during table resize by a single thread, or as single buckets by multiple concurrent threads during execution. Given the previously stated, we have three scenarios where dynamic allocation is necessary and only one scenario where the allocation must be thread-safe.

As mentioned in the previous section, EA-PHT will "live" inside a memory-mapped file. Similarly to libpmemobj, the notion of a root object needs to be defined, which we will call the entry point item. This entry point item should contain all the necessary data needed for recovery and will be the first item the data structure will access after a crash or power cycle. It is clear that the root item should be part of the entry point item. Allocation of a hash table item is needed only during a full table resize. During a full table resize, a minimum of two hash table item instances are needed, one pointing to the old table and one pointing to the new table. Predefining a number of these items to reside as part of the entry item will remove the complexity of dynamically allocating these items on persistent memory. Arguably, the hash table items are mostly read and will be modified only when a table resize needs to be performed. Memory degradation due to writes to the same memory location will not be a problem due to the infrequency of these items needing to be modified. By setting the root item and a predefined number of hash table items as part of the entry item, only the dynamic allocation of buckets in its two different scenarios needs to be handled, reducing the complexity of dynamic allocation.

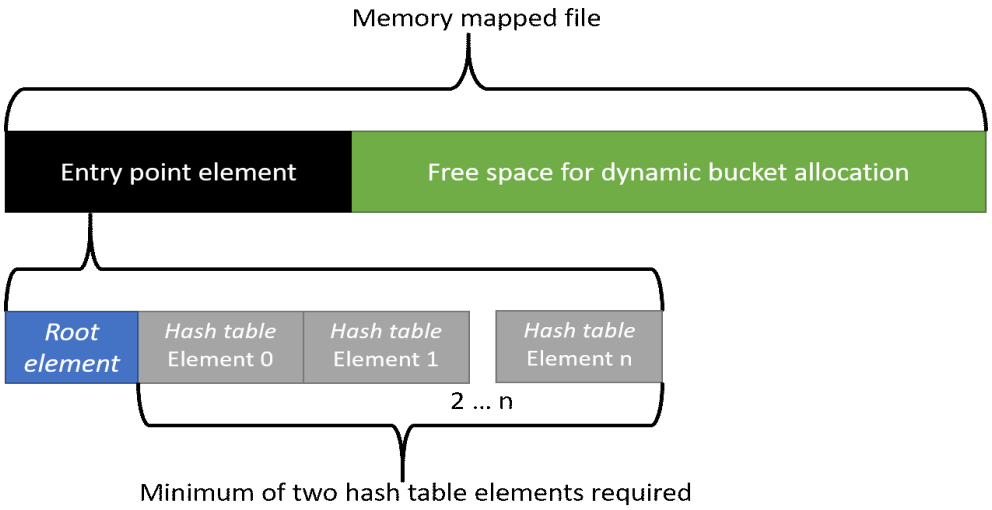


Figure 3.1: Entry item layout on the memory mapped file.

## 3.4 Offset-based allocation of EA-PHT

In this section, the allocation technique and internal composition of the items that comprise EA-PHT are described. As shown in Figure 3.1, the entry point item will be comprised of the root item and a predefined number of $n$ hash table items, where $n$ needs to be set to a value equal or greater than two before compilation. These items are statically allocated at the beginning of the memory-mapped file. The free space left in the memory-mapped file is used to allocate the bucket items dynamically during run-time. As mentioned in DASH [8], it is known that DCPMM modules have limited bandwidth in comparison to DRAM modules, around 3-14 times lower. To reduce the overhead of performing expensive writes to initialize allocated items during run-time, the initial creation of the memory-mapped file will follow a *pmem_memset_persist()* function call, which will initialize the whole file to zero. As an example, when initializing a memory-mapped file of 8.5GB to zero, it takes roughly 5 seconds. While this can take several seconds during initial creation, the initial zeroing of the memory-mapped file will only be done once and simplifies the recovery process by removing the need to handle uninitialized items. It is also reasonable to assume that developers using persistent memory data structures intend to create these data structures once and maintain them across executions. Based on the previous statements, it seems reasonable to remove some of the recovery complexity at the cost of additional overhead for initial creation.

In EA-PHT, the root item is comprised of an id used to locate the hash table item, a resize lock, an offset that indicates the first available space after the entry item and a status variable used to determine if the data structure exited gracefully, or if it was interrupted during execution. The hash table item contains an offset to a chunk, the size of the hash table, the hash value, a version number, the number of allowed expands, the expand counter, and four variables used for the table resize mechanism of the original CLHT-LB implementation. A chunk contains a contiguous array of bucket items plus the overflow buckets, further explained on Subsection 3.4.2. The size of the bucket items is restricted to 256 bytes, which is the access granularity size of DCPMM modules. The buckets are comprised of a lock for concurrency control, fifteen key-value pairs, and a next bucket offset. The locks in bucket items use one bit as the update lock, one bit as the resize lock, these locks are mutually exclusive, meaning only one can be acquired at a time. The remaining six bits are used as a version number, as shown in Figure 3.3. Each time a lock is released, its version number is simultaneously incremented by one or rolled back to zero. For this design, only six bits are used as a version number for concurrency control. If a larger version number is required, we have two options: use the MSB of the next bucket offset to store the resize lock, or remove a key-value pair and add a separate resize lock variable. An in-depth overview of the use of the variables inside these items will be given throughout this chapter.

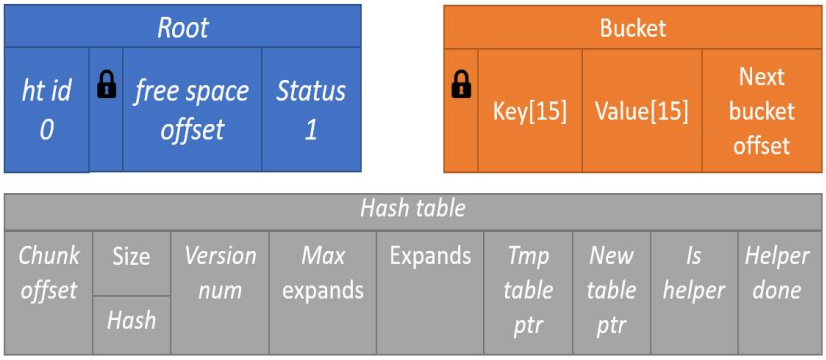


Figure 3.2: EA-PHT items.

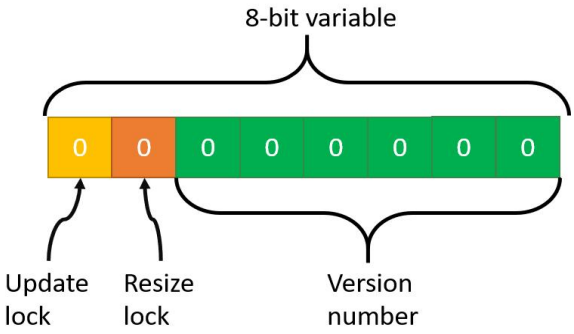


Figure 3.3: Bucket item internal lock structure.

### 3.4.1 Access of statically allocated items

Accessing the items contained in the entry point item is straightforward. The first item stored in the persistent memory-mapped file is the root item; thus, every time we call the function *pmem_map_file()* to create or re-open the memory-mapped file, a direct pointer to the root item is provided. To get a direct pointer to a hash table item, the *get_ht_ptr()* function is used. This function uses the root item id to calculate a direct pointer as shown in Figure 3.4 below.

```
1       ptr_ht=root+sizeof(root_element)+(root->id*sizeof(ht_element));
```

Figure 3.4: Get direct pointer to hash table item (ptr_ht) from id.

Upon initial creation of EA-PHT, the root item id will be set to zero. As full table resizes are performed during run-time, the id value in the root item will increment or roll back to zero in a similar manner to that of a circular buffer to change between hash table items.

### 3.4.2 Allocation and access of dynamically allocated items

To handle the dynamic allocation of bucket items, the notion of a chunk is introduced. A chunk is comprised of a contiguous array of bucket items and individually allocated overflow bucket items, as shown in Figure 3.5. A chunk's size will change during run time

as we start placing overflow buckets in the next free memory space after the contiguous array.

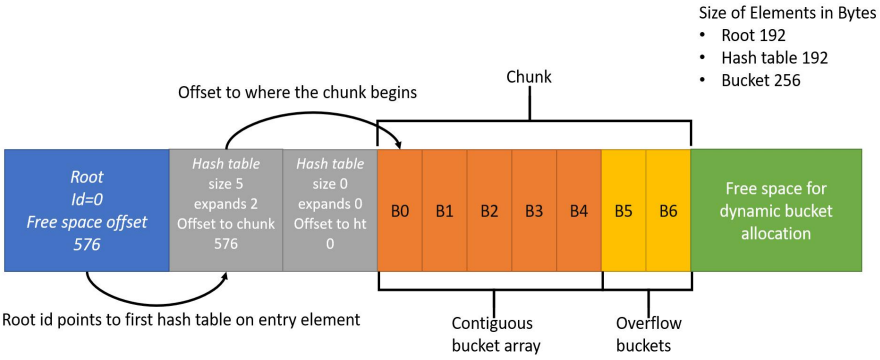


Figure 3.5: Representation of EA-PHT.

To allocate overflow buckets, the *allocate_bucket()* function is used. This function uses the expand counter variable as an index that specifies the overflow bucket's location after the contiguous array. It is important to note that the allocation of overflow buckets needs to be thread-safe since several threads may be trying to allocate an overflow bucket concurrently. The expanded counter is atomically incremented with an *_sync_add_and_fetch()* instruction in every allocation of a new overflow bucket. The returned value minus one will be used to calculate the reserved offset for the newly allocated bucket. Then the offset of the overflow bucket can be calculated as shown in Figure 3.6, where the index value is the returned value by the *_sync_add_and_fetch()* and minus one, since the previous value of the expand counter is the reserved index for the new bucket. To get a direct pointer to the overflow bucket offset, the offset_to_ptr() function is used. This function adds the provided offset to the root pointer. To access a bucket contained in the contiguous array of buckets, the *get_bucket_ptr()* function is used. In this function, the direct pointer is calculated as shown in Figure 3.7, where the index refers to the array location of the bucket to be accessed.

```
1       bucket_off=ht->chunk_off+(ht->size*sizeof(bucket))+(index*sizeof(
           bucket));
```

Figure 3.6: Offset to overflow bucket.

```
1       ptr_bucket=root+ht->chunk_off+(index*sizeof(bucket));
```

Figure 3.7: Direct pointer to bucket in contiguous array.

When a full table resize is triggered, the thread resizing the table will mark the chunk offset, similarly to that of pointer tagging [59], where the MSB bit of the chunk offset is set to one. Marking this offset prevents other threads from allocating new overflow buckets, thus allowing the thread performing the resize to calculate the placement of the new chunk. The new chunk is placed just after the last overflow bucket of the previous chunk. A more in-depth explanation of this mechanism is described in Section 3.5.4, where the resize is explained.

## 3.5 EA-PHT operations

This section describes how the different operations of EA-PHT are performed, as well as detailing the combination of fences and flushes used by these operations to impose an order when performing stores to persistent memory. As mentioned in Section 2.2, 8-byte stores in persistent memory followed by a flush are guaranteed to be power-fail atomic, meaning that either the new value or the old value will remain in persistent memory after a crash during the store operation. It is implied that all variables that are mentioned in this section are 8-byte aligned, which means all stores are guaranteed to be power-fail atomic. When a variable is written, it is implied that this is followed by a *Flush()* instruction, which consists of an MFENCE, CLWB, and another MFENCE; this is based from the flush instruction implemented by RECIPE [54]. In EA-PHT, the resize lock and bucket lock variables are neither flushed nor persisted. Upon recovery or restart, all these locks are reinitialized. In the case of the bucket locks, their embedded version number is also reinitialized.

### 3.5.1 The search operation

Similarly to CLHT-LB, EA-PHT implements a read operation without the need for synchronization. In Algorithm 1 line 6, the search operation locates the bucket in the contiguous array to be searched and in line 8, it takes a snapshot of the version number contained in its lock. Then the search operation proceeds to traverse the key-value pairs while taking snapshots, (see line 11), of the value for each key-value pair accessed. If the traversal of the key-value pairs in the current bucket has completed and the key-value pair was not

found, the search operation will proceed to access the next linked bucket if it exists, line 23. After a matching key is found, (line 13), the snapshot of the value is compared to the current value held at the key-value pair, followed by the comparison of the version number snapshot and the current version number held in the lock. If these conditions are true, the operation returns the value. If the snapshot of the value does not match, then the operation returns zero, since a concurrent operation removed the key-value pair. If the snapshot of the version number does not match, it means a concurrent operation has modified the bucket, and the search operation restarts.

**Algorithm 1** Search function

```
1:  procedure SEARCH(KEY)
2:      Retry :
3:      hash_table = get_ht_ptr(root→id)
4:      chunk = get_chunk_ptr(hashtable→chunk_off)
5:      bin = hash(hashtable→hash, KEY)
6:      bucket = get_bucket_ptr(chunk, bin)
7:      entry=bucket
8:      version = (bucket→lock)&VERSION_MASK
9:      do
10:         for (j=0; j<ENTRIES_PER_BUCKET; j++) do
11:             snapshot = bucket→val[j]
12:             LFENCE()
13:             if bucket→key[j] == KEY then
14:                 if bucket→val[j] == snapshot then
15:                     if version == ((entry→lock)&VERSION_MASK) then
16:                         return snapshot
17:                     else
18:                         goto Retry
19:                 else
20:                     return 0
21:         b=offset_to_ptr(bucket→next)
22:         bucket=b;
23:     while bucket ≠ null
24:     return 0
```

### 3.5.2 The insert operation

An insert operation can be performed in three different ways: a lock-free insert, where the key is already contained in the bucket; a simple insert, where there is no need to allocate an overflow bucket; and an overflow insert, where the allocation of an overflow bucket is needed. A lock-free insert will search the bucket without acquiring a lock, using the *bucket_exists()* function, finding the key, and returning false, Algorithm 2 line 7, without the need to perform any stores to persistent memory. A simple insert will not find the searched key and will acquire the update lock. After acquiring the update lock, an empty space is located in the current bucket, (lines 20 - 26); then, in the following order, the value is written, then the key is written to ensure that a search operation can only find a key when its value is present, (lines 42 - 45). After the key is written, the update lock is released, and at the same time, the version number is updated, (line 46).

An overflow insert will not find the searched key. It will acquire the update lock and determine that the current bucket has no free entry for the key-value pair, (line 29). Then, the *create_bucket()* function will be called. If the chunk offset is marked, (line 31), the *create_bucket()* function will return and signal the insert operation to restart the operation, since allocation to the chunk is restricted by a resize operation. If the chunk offset is not marked, the expand counter is atomically incremented/written by an *_sync_add_and_fetch()* instruction. The returned value of the *_sync_add_and_fetch()* instruction minus one is then used to calculate the allocated bucket offset, as shown in Figure 3.6. After receiving an offset to the newly allocated bucket, from the *create_bucket()* function, the direct pointer to this bucket can be calculated with the offset. In the first key-value pair of the newly allocated bucket, the value is written, followed by the writing of the key (lines 35 - 38). Then the offset to the newly allocated bucket is written to the next bucket offset of the previously full bucket, (line 39). Finally, the update lock is released, and at the same time, the version number is incremented, (line 46). For both the simple insert and overflow insert, after acquiring the lock, if during the traversal of the key-value pairs the key to be inserted is found, the operation releases the lock and returns false.

**Algorithm 2** Insert function

```
1: procedure INSERT(KEY, VALUE)
2:     Retry:
3:     hash_table = get_ht_ptr(root→id)
4:     chunk = get_chunk_ptr(hashtable→chunk_off)
5:     bin = hash(hashtable→hash, KEY)
6:     bucket = get_bucket_ptr(chunk, bin)
7:     if bucket_exists(bucket,KEY) == true then return false
8:     lock = &bucket→lock
9:     while LOCK_ACQ(lock, hashtable)=0 do // wait until lock acquired
10:        if if_chunk_lock(hashtable→chunk_off)=true then
11:            goto Retry
12:        hash_table = get_ht_ptr(root→id)
13:        chunk = get_chunk_ptr(hashtable→chunk_off)
14:        bin = hash(hashtable→hash, KEY)
15:        bucket = get_bucket_ptr(chunk, bin)
16:        lock = &bucket→lock
17:    key_location = null
18:    val_location = null
19:    do
20:        for (j=0; j<ENTRIES_PER_BUCKET; j++) do
21:            if bucket→key[j] == KEY then
22:                lock_REL(lock)
23:                return false
24:            else if key_location == null and bucket→key[j] == 0 then
25:                key_location= & bucket→key[j]
26:                val_location=& bucket→val[j]
27:        resize_flag = 0
28:        if bucket→next == null then // if bucket not linked
29:            if key_location == null then // if there is no space in bucket
30:                bucket_offset = allocate_bucket(hashtable, &resize_flag )
31:                if bucket_offset == LOCKED_CHUNK then
32:                    lock_REL(lock)
33:                    goto Retry
34:                b=offset_to_ptr(bucket_offset)
35:                b→val[0]=VAL
36:                Flush(&bucket→val[0])
```

```
37:                 b→ key[0]=KEY
38:                 Flush(&bucket→key[0])
39:                 bucket→next=bucket_offset
40:                 Flush(&bucket→next[0])
41:             else
42:                 *val_location=VAL
43:                 Flush(val_location)
44:                 *key_location=KEY
45:                 Flush(key_location)
46:             lock_REL(lock)
47:             if resize_flag = true then
48:                 full_table_resize()
49:             return true
50:         b=offset_to_ptr(bucket→next)
51:         bucket=b;
52:     while true
```

### 3.5.3 The remove operation

A remove operation can be performed in two different ways. A lock-free remove, where the key is not contained in the bucket, and a locking remove, where the key is contained in the bucket and the operation proceeds to remove it. A lock-free remove will search the bucket without acquiring a lock, using the *bucket_exists()* function, not finding the key, and returning false without the need of performing any stores to persistent memory, (Algorithm 3 line 6). A locking remove will find the searched key and will acquire the update lock. After acquiring the update lock, it will traverse the key-value pairs. If the matching key is found, (line 16), then, in the following order, the key is overwritten with zero, then the value is overwritten with zero to ensure that a search operation can only find a key when its value is present. Finally, the update lock is released, and at the same time, the version number is updated, line 22. If the matching key is not found after acquiring the lock and traversing the key-value pairs, the operation releases the lock and returns false.

**Algorithm 3** Remove function

```
1: procedure REMOVE(KEY)
2:     hash_table = get_ht_ptr(root→id)
3:     chunk = get_chunk_ptr(hash_table→chunk_off)
4:     bin = hash(hash_table→hash, KEY)
5:     bucket = get_bucket_ptr(chunk, bin)
6:     if bucket_exists(bucket,KEY) = false then return false
7:     lock = &bucket→lock
8:     while LOCK_ACQ(lock, hash_table)=0 do// wait until lock acquired
9:         hash_table = get_ht_ptr(root→id)
10:        chunk = get_chunk_ptr(hash_table→chunk_off)
11:        bin = hash(hash_table→hash, KEY)
12:        bucket = get_bucket_ptr(chunk, bin)
13:        lock = &bucket→lock
14:    do
15:        for (j=0; j<ENTRIES_PER_BUCKET; j++) do
16:            if bucket→key[j] == KEY then
17:                bucket→key[j]=0
18:                Flush(bucket→key[j])
19:                val=bucket→val[j]
20:                bucket→val[j]=0
21:                Flush(bucket→val[j])
22:                lock_REL(lock)
23:                return val
24:        b=offset_to_ptr(bucket→next)
25:        bucket=b;
26:    while bucket ≠ NULL
27:    lock_REL(lock)
28:    return 0
```

### 3.5.4 The resize operation

When a full table resize is required, a new hash table with a size of a constant multiplied by the size of the old hash table will be created. The resize operation is performed as follows. After a successful overflow insert, if the thread finds that the expand counter surpasses the max allowed expands, it will try to acquire the resize lock. If the thread finds the lock to be acquired, then the insert operation completes successfully and returns since another

thread has already begun the resize operation. A thread that acquires the resize lock will proceed to call the *create_hash_table()* function, which will select the new hash table item to be used and initialize its variables.

The *create_hash_table()* function first marks the old hash table item chunk offset by setting the MSB of the chunk offset, using a CAS operation, (see Algorithm 4 line 3). After that, no other thread may allocate overflow buckets in the old chunk. Then, the root id is used to determine which is the id of the new hash table item to be used, (lines 4 - 8), and the direct pointer to the new hash table item is obtained as shown in line 9. If the size variable in the new hash table item contains a value greater than zero, this item already points to a previous chunk; then the previous chunk is deallocated by issuing a *pmem_memset_persist()* to set the entire chunk to zero, (see lines 10 - 13).

After clearing the previous chunk referenced by the new hash table item, if it existed, the size variable is then written with the new size of the table, line 14. Then, the chunk offset is calculated and written. To calculate the chunk offset for the new hash table, the old hash table chunk offset, table size, and expand counter are used, as shown in line 17. Then the expand counter is written to zero. Finally, the max expands, hash, is helper, helper done, table new, and table temp variables are initialized, without requiring flushing since these variables are always initialized upon recovery. (See Section 3.6 for more details).

After the initialization of the new hash table item and just before beginning the rehash operation, the thread performing the resize needs to re-check the calculated offset, given that there could be a thread $t1$ which sees the chunk unlocked, then the thread $t2$ performing the resize marks the chunk offset and reads the expand counter prior to the increment by the thread $t1$ allocating the overflow bucket. This way, it can be ensured that there are no overlapping chunks.

**Algorithm 4** Allocation of new hash table items

```
1: procedure CREATE_HASH_TABLE(NUM_BUCKETS)
2:     ht_root = get_ht_ptr(root→id)
3:     while LOCK_ACQ(lock_chunk())=0 do // wait until chunk lock acquired
4:     id=ht_root→id
5:     if id== (NUMBER_OF_PREALLOCS - 1) then
6:         id=0
7:     else if id < (NUMBER_OF_PREALLOCS - 1) then
8:         id=id+1
9:     ht_new = get_ht_ptr(id)
10:    if ht_new→size ≠ 0 then
11:        chunk = get_chunk_ptr(ht_new→chunk_off)
12:        size =(ht_new→size+ht_new→expands)*sizeof(bucket_t)
13:        pmem_memset_persist(ht_new,0,size))
14:    ht_new→size=num_buckets
15:    Flush(ht_new→size)
16:    old_chunk_size+=(ht_root→size+ht_root→expands)*sizeof(bucket_t)
17:    new_chunk_off= ht_root→chunk_off+ old_chunk_size
18:    ht_new→chunk_off=new_chunk_off
19:    Flush(ht_new→chunk_off)
20:    ht_new→expands=0
21:    Flush(ht_new→expands)
22:    ht_new→max_expands = (EXPAND_FACTOR*num_buckets)
23:    ht_new→hash= num_buckets-1
24:    ht_new→is_helper = 1
25:    ht_new→helper_done = 0
26:    ht_new→table_new = null
27:    ht_new→table_temp = null
28:    return ht_new
```

After the *create_hash_table()* function has initialized the new hash table, the rehashing will be performed. The rehashing algorithm is the same as the one provided by CLHT [61]. The rehashing consists of traversing the buckets along with any overflow buckets linked and copying all their key-value pairs to the new hash table. To begin the rehash of the accessed bucket, the resize lock must be acquired first. This lock is acquired and never released since we do not want any other process to be able to modify a bucket that is or has been rehashed to the new table. The resize lock and update lock are mutually exclusive, so a resize lock will only be acquired if neither lock is set. The thread performing the resize will begin the rehash operation by traversing all the buckets in the old table from the first bucket to the last, while a helping thread will start from the last bucket to the first bucket. Both will stop if they encounter a bucket locked in resize mode.

During a full table resize, search operations can be executed normally. Any concurrent threads performing an insert or remove that do not find the current bucket resize lock acquired will be able to proceed with its operation. If the bucket resize lock is set, the thread will help to resize, if possible, or wait for the resize operation to finish.

When the rehashing is finished, the contiguous array of bucket items along any overflow buckets allocated during resize in the new chunk are flushed. After the data is flushed, the old hash table item version number plus one is written to the new hash table item. Then the root id is atomically swapped/written for the id of the new hash table item. Finally, the resize lock is released, and the operation completes. As mentioned previously, the hash table item uses four variables during the table resize operations, and their use is explained as follows.

- **Tmp table pointer:** Upon initialization, this variable is set to null. During table resize, a direct pointer to the next hash table is set. A helping thread during resize uses this pointer to access the new table. It is important to note that for a thread to begin helping the resize of the table, it needs to encounter a bucket with the resize lock set, which means that the thread performing the resize has previously set the direct pointer to the new table in this variable.

- **New table pointer:** When a full table resize is triggered, this pointer is set to null. When the new table has been rehashed and is ready, a pointer to the new table is assigned to this variable, signaling all the waiting threads that the resize operation has completed.

- **Is helper:** Upon initialization, this variable is set to zero. A thread trying to enter the resize help function will use a *_sync_sub_and_fetch()* instruction on this variable. If the value returned is greater than or equal to zero, the thread will then proceed to enter the resize help function. Only one thread will be able to help the resize operation.
- **Helper done:** Upon initialization, this variable is set to zero. This variable is set to one by the helping thread when it finishes helping, and signals the main thread performing the resize that the helper has finished.

## 3.6 Recovering from a crash

To determine if the program exited gracefully, the status variable contained in the root item is used. If the status variable value is two, it means that the program exited gracefully. If the status variable value is one, then the program crashed during execution, and executing the recovery function is needed. If the status variable value is zero or any other value, then the program crashed before the initial creation of the hash table. The recovery function consists of three steps that are performed in the following order: re-initialization and fixing incomplete operations, linking of buckets, and detecting if the program crashed during a full table resize.

### 3.6.1 Possible inconsistent states left by EA-PHT operations

Operations such as a simple insert, overflow insert, locking remove, or full table resize, perform stores to persistent memory. If these operations were to be interrupted by a crash or power failure, the memory could be left in an inconsistent state. This subsection describes the different inconsistent states that can remain in the EA-PHT in its different operations.

- **Simple insert:**
  - Successful insert: both the value and key were persisted.
  - Partial insert: only the value was persisted.
  - Failed insert: neither the value nor the key were persisted.

- **Overflow insert:**
  - Successful insert: the expand counter, the key-value pair, and the offset were persisted.
  - Unliked successful insert: the expand counter, key, and value pair were persisted, the offset was not persisted to the previous full bucket offset variable.
  - Unlinked partial insert: only expand counter, and the value were persisted.
  - Unlinked failed insert: the expand counter was persisted, the insert never took place.
  - Failed bucket allocation: the increment of the expand counter was not persisted.
- **Locking remove:**
  - Successful remove: both the value and key were persisted.
  - Partial remove: only the key was persisted.
  - Failed remove: neither the value nor the key were persisted.

If the program crashes during a full table resize operation, it can be tricky to determine the exact point where this happened. To avoid complex algorithms that may solve this issue, a lazy approach is taken to separate the analysis of the resize operation into five possible states. These states are determined by the root item id variable, and the hash table items, size, chunk offset, and version number variables. All of these variables are persisted in a specific order during a full table rehash operation; thus, the full table rehash operation can be split into the described states below. This approach dramatically simplifies the recovery process, while sacrificing recovery time.

- **Full table resize:**
  - Did not crash during a resize operation: A previous resize operation took place successfully.
  - Resize never started: The hash table item referenced by the root id has an expand counter value that surpasses the expand threshold, the chunk offset might be marked, and none of the variables of the new hash table item to be used were persisted.
  - Resize deallocated old chunk: persisted the new size on the new hash table item.

– Resize might have begun rehash: persisted the new size and new chunk offset in the new hash table item.
– Resize finished, missed swap of root id: persisted the new size, new chunk offset, and updated version number in the new hash table item, but did not persist the new id to the root item variable.

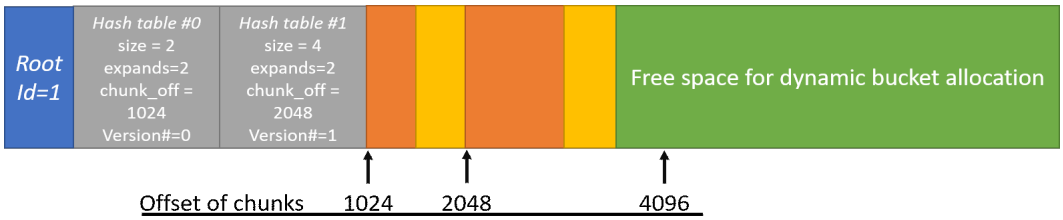


Figure 3.8: Did not crash during a resize operation.

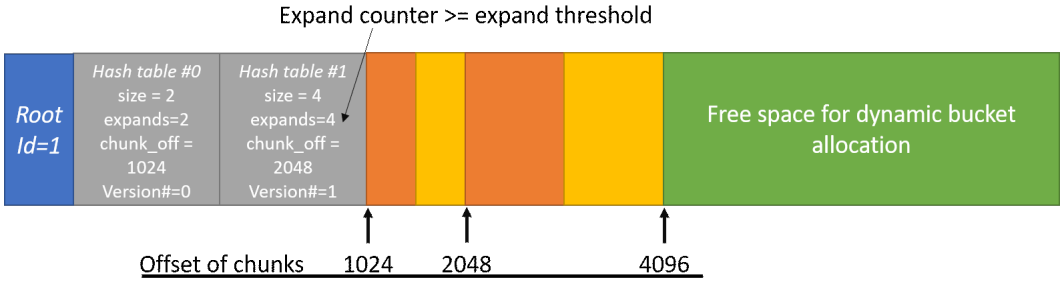


Figure 3.9: Resize never started.

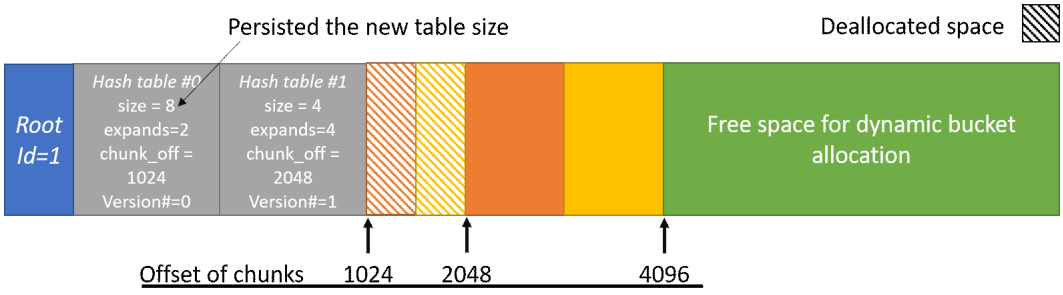


Figure 3.10: Resize deallocated old chunk.

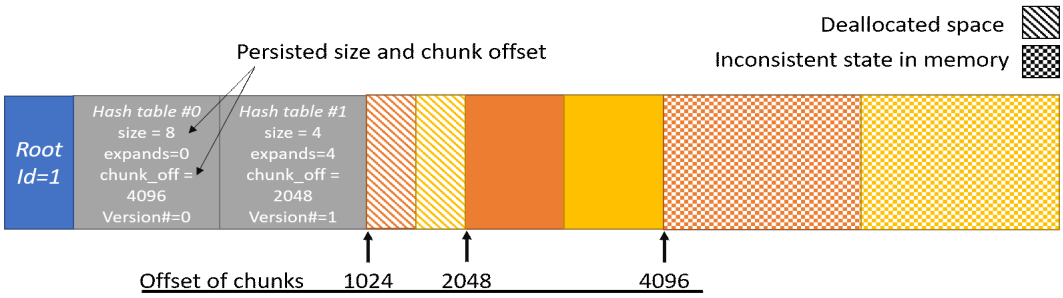


Figure 3.11: Resize might have begun rehash.

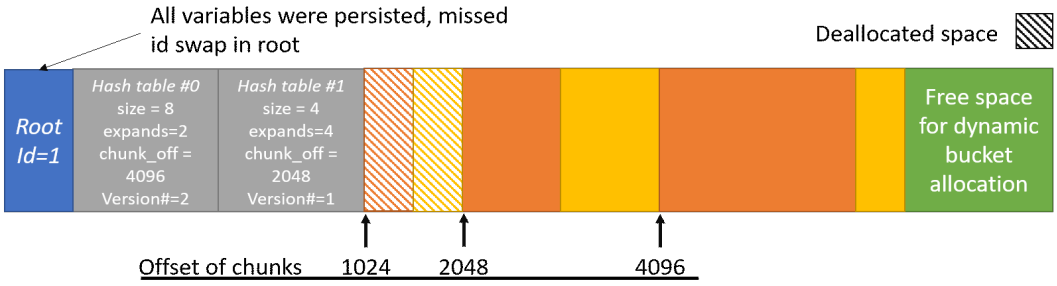


Figure 3.12: Resize finished, missed swap of root id.

### 3.6.2 Re-initialization and repairing incomplete operations

The *re_init_and_fix()* function is used to fix incomplete operations and re-initialize variables of the data structure. The root item resize lock is set to zero, then the hash table referenced by the root id is accessed, and finall the max expands, and hash variables are recalculated. Given that these are not manually flushed and may contain invalid values, they are calculated using the size variable of the hash table item, which is manually flushed during a resize operation and is guaranteed to contain a valid size for the current hash table. Also, the *is_helper*, *helper_done*, *table_new*, and *table_temp* variables are initialized to its default. Then the buckets contained in the current chunk referenced by the hash table item are traversed. The size variable and expand counter are used to determine the number of buckets contained in the chunk, (see Algorithm 5 line 9). Each bucket is then accessed, its lock is set to zero, and the key-value pairs are traversed. If a key-value pair in the bucket contains a value different from zero, and a key equals to zero (partial insert, unlinked partial insert or partial remove), then the value is set to zero, (line 14). In the case of a partial insert and unlinked partial insert, these operations will be aborted. In the case of a partial remove, this operation will be completed.

**Algorithm 5** Re-initialization and repairing incomplete operations

```
1:  procedure RE_INIT_AND_FIX()
2:      ht_root = get_ht_ptr(root→id)
3:      chunk = get_chunk_ptr(ht_root→chunk_off)
4:      root→resize_lock = LOCK_FREE
5:      ht_root→max_expands = (EXPAND_FACTOR*ht_root→size)
6:      ht_root→hash= ht_root→size-1
7:      ht_root→is_helper = 1
8:      ht_root→helper_done = 0
9:      for (i = 0; i<(ht_root→size + ht_root→expand_cnt); i++) do
10:         bucket = get_bucket_ptr(chunk, i)
11:         bucket→lock = LOCK_FREE
12:         for (j = 0; j < ENTRIES_PER_BUCKET; j++) do
13:             if (bucket→val[j] ≠ 0) AND (bucket→key[j]==0) then
14:                 bucket→val[j]=0
15:                 Flush(bucket→val[j])
```

### 3.6.3 Linking of buckets

All the overflow buckets are traversed. If both the first key, and value are equal to zero (unlinked failed insert), this bucket might be "lost" in the sense that the hash table might not be able to access this bucket. Still, when the deallocation of the chunk is performed later, this bucket will be reclaimed along with all the other buckets in the chunk. If both the key and value are different from zero, the key-value pair will be searched in the same manner as the search operation, (see Algorithm 6 line 9). If the key-value pair is found (successful insert), the bucket is already linked correctly to the data structure. If the key-value pair is not found (unliked successful insert), the next bucket offset variable of the last bucket accessed during the search operation will be assigned the offset of the bucket containing the not found key-value pair, thus completing an overflow insert operation, (line 13).

**Algorithm 6** Link buckets

```
1:  procedure LINK_BUCKETS()
2:      ht = get_ht_ptr(root→id)
3:      // begin holds pointer to first overflow bucket
4:      begin= get_chunk_ptr(ht→chunk_off)+(ht→size*BUCKET_SIZE)
5:      // end holds pointer to last overflow bucket
6:      end= begin+(ht→expand_cnt*BUCKET_SIZE)
7:      for (i = begin; i<end; i=i+BUCKET_SIZE) do
8:          bucket = i
9:          if (bucket→key[0] ≠ 0) AND (bucket→val[0] ≠ 0) then
10:             result= search(bucket→key[0])
11:             if (result = 0) then
12:                 offset= ptr_to_offset(bucket)
13:                 link(ht, bucket→key[0], offset)
```

**Algorithm 7** Link function

```
1: procedure LINK(HT,KEY,OFFSET)
2:     chunk = get_chunk_ptr(HT→chunk_off)
3:     bin = hash(HT→hash, KEY)
4:     bucket = get_bucket_ptr(chunk, bin)
5:     do
6:         if bucket→next = 0 then
7:             bucket→next=offset
8:             Flush(bucket→next)
9:             return 1
10:        b=offset_to_ptr(bucket→next)
11:        bucket=b
12:    while bucket ≠ null
13:    return 0
```

### 3.6.4 Detecting if the program crashed during a resizing

To detect whether the program crashed during a full table resize, the preallocated hash table items are scanned; based on their position, the id's of the hash table item containing the maximum size and maximum version numbers are obtained. For example, if the first preallocated hash table item contains the maximum size compared to all the other preallocated hash table items, the id of the maximum size will be zero.

If both the maximum version id and maximum size id match the id contained in the root item, the program did not crash during a resize. Still, the hash table item expand counter is read to check if it surpasses the maximum expands. If this is the case (resize never started), the MSB of the chunk offset is cleared, and the resize function is executed with a single thread. If the maximum version id and the maximum size id are equal, but the root id is different, the program crashed after a successful rehash. Then the root id needs to be changed to the maximum version id (resize finished, missed swap of root id). If the maximum version id and the root id are equal, but the maximum size id is different, the program crashed during resizing. The hash table item referenced by the root id is used to calculate the next chunk offset for the new hash table item. If the computed chunk offset does not match the chunk offset contained in the hash table item referenced to by the maximum size id (resize deallocated old chunk), the rehash did not take place. If the chunk offset calculated from the hash table item referenced by the root id matches the one of the hash table item referenced by the maximum size id (resize might have begun

rehash), the crash might have happened while rehashing. If so, the chunk is reinitialized to zero using a *pmem_memset_persist()*, to remove any incomplete operations that were interrupted during the rehash, and then the resize function is called.

## 3.7 Experimental Setup

The testbed used for the results presented in Chapters 3, 4, and 5 was an Intel Xeon Gold 6230 server with 20 cores and Intel Optane Persistent Memory.

The Yahoo! Cloud Serving Benchmark (YCSB) [6] was used for evaluating the hash tables. The benchmark consists of a load phase and a workload phase. The load phase inserts 64M key-value pairs to grow the hash table. During the load phase, the hash table is expected to change in size. This phase will capture the impact of the resizing mechanism used for the implementations tested.

The workload phase focuses solely on measuring the performance of insert and search workloads. Each workload consists of 64M operations. Four types of workloads were tested: 50/50, 80/20, 95/5, 100/0 read/write. The experiments were run five times, and hyperthreading was used in executions with more than 20 threads.

## 3.8 Performance evaluation

Implementations such as DASH [8], CCEH [39], and Level Hashing [45], focus on reducing the overall number of stores to persistent memory performed by their operations, without considering the overhead of using general-purpose memory allocators. These designs avoid the full table rehash operation and propose different alternatives to handle collisions in the data structure to reduce the number of allocations needed. On the other hand, EA-PHT focuses on reducing the number of fences and flushes needed by the search, insert (including the fences and flushes needed for allocation) and remove operations only, overlooking the cost of a full table rehash. Shown below are the number of flushes and fences needed for each operation of EA-PHT.

- **Search:** uses 1 LFENCE per key-value pair traversed.
- **Simple Insert:** uses 4 MFENCE and 2 CLWB.
- **Overflow Insert:** uses 8 MFENCE and 4 CLWB.
- **Locking Remove:** uses 4 MFENCE and 2 CLWB.

In EA-PHT, overflow buckets (256 bytes) are the most frequently allocated items. A comparison in the cost of allocation using libpmemobj, which is used by the other implementations, in contrast to the embedded allocation technique, is shown in Figure 3.13. The throughput for both techniques was measured, and the results show that the embedded allocation technique can be up to 2.9 times faster in comparison to libpmemobj.

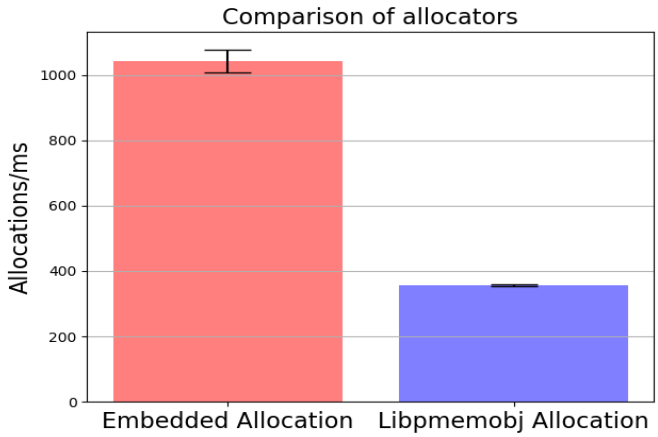


Figure 3.13: Average time to allocate 256 bytes (single thread).

For the workload graphs, each point represents the average of five different executions where the error bars are defined as the standard deviation of the set of measurements. DASH [8] takes into account the access granularity of DCPMM devices and sets the size of buckets to be 256 bytes, while CCEH and Level Hashing, use 64 bytes for their bucket size. This specific design decision in DASH is intended to benefit from data locality. EA-PHT borrows the idea of DASH by setting the size of its buckets to 256 bytes. Shown below in the workload results, a significant gap between designs that use a bucket size of 256 bytes and designs that use a bucket size of 64 bytes and can be seen.

The reduced cost of allocation and taking into consideration the access granularity of DCPMM devices in the design of EA-PHT allows it to surpass the other implementations in all four workloads. The difference in performance between EA-PHT and CCEH and Level Hashing is related mainly to data locality, since EA-PHT uses 256 byte buckets. On the other hand, for write-heavy workloads 50/50 (read/write), Figure 3.14 and 80/20 (read/write), Figure 3.15, EA-PHT can outperform DASH given its low cost of allocation, being up to 22% faster than DASH in the 50/50 (read/write).

For read-heavy workloads where no allocation is needed, 100/0 (read/write) Figure 3.17 and 95/5 (read/write), Figure 3.16, EA-PHT still can outperform DASH. The gap between DASH and EA-PHT in the read-heavy workloads is related to how the implementations store and manage their key-value pairs. DASH uses several techniques to balance the load in the hash table and a fingerprinting technique to avoid unnecessary probing. While the fingerprinting technique reduces the need for probing all buckets, their load balancing techniques add extra steps to their search operation. A search operation needs to check the bucket where the key is to be stored and the next adjacent bucket. In the presence of key-value pairs in the stash bucket, this too needs to be checked. In contrast, EA-PHT needs to check only one bucket most of the time and only needs to check extra buckets if there is a chained bucket. This allows EA-PHT to be up to 33% faster than DASH in the read-only workload C 100/0 (read/write).

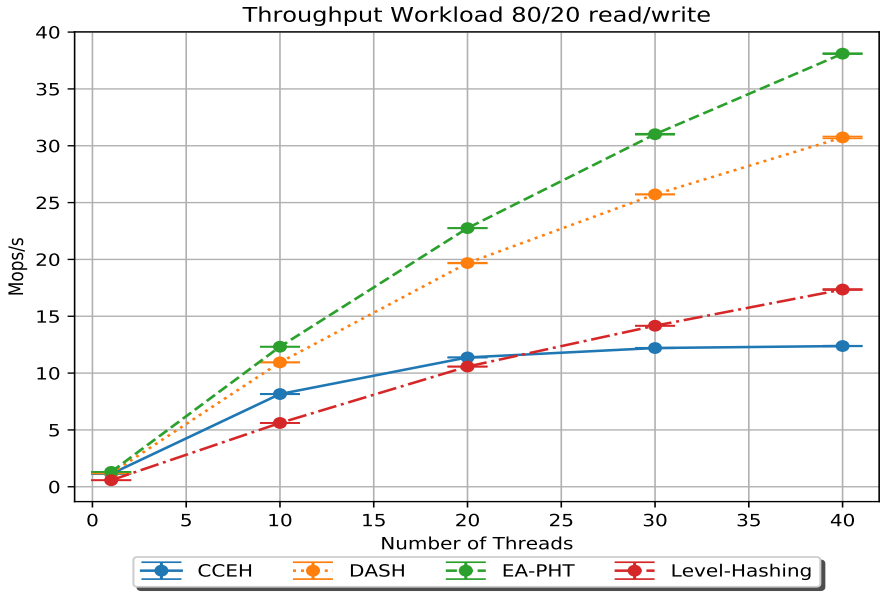


Figure 3.15: 80/20 read/write workload.

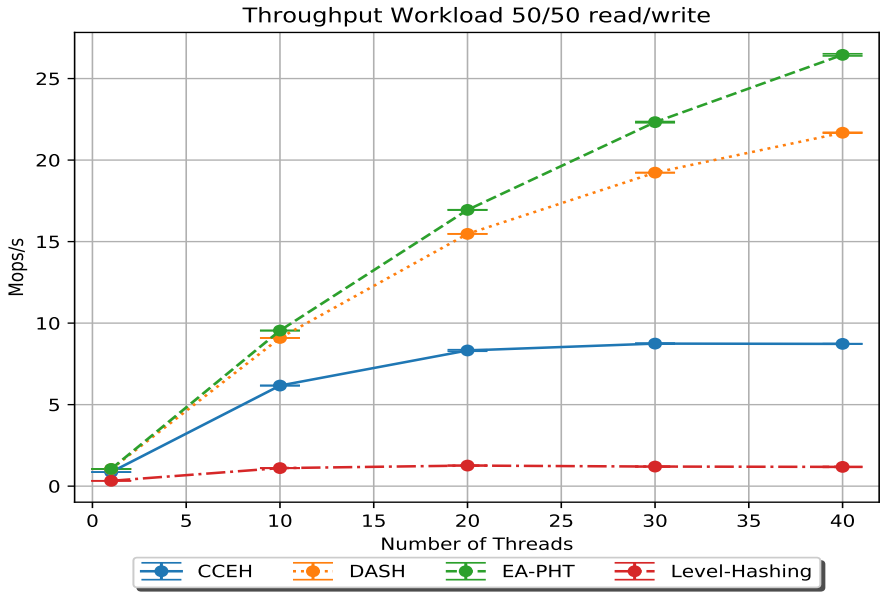


Figure 3.14: 50/50 read/write workload.

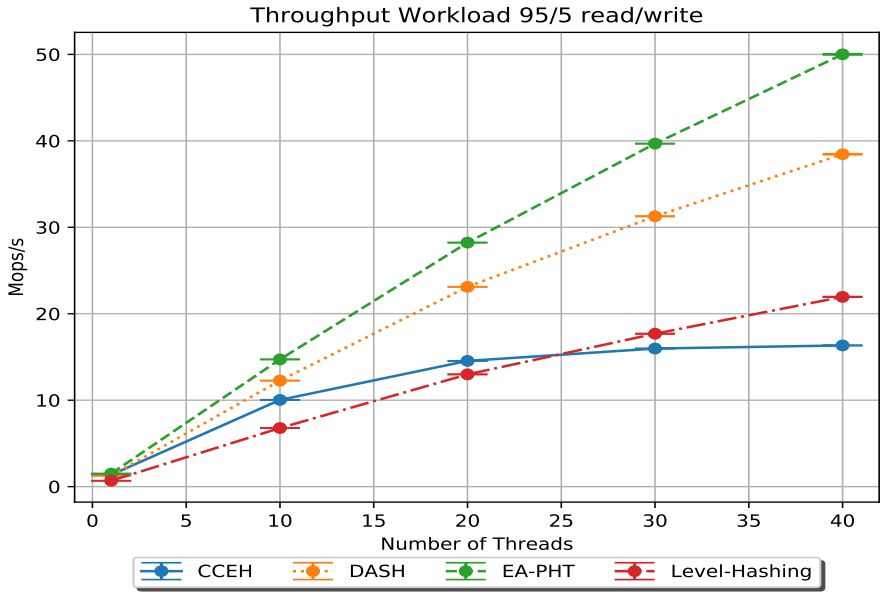


Figure 3.16: 95/5 read/write workload.

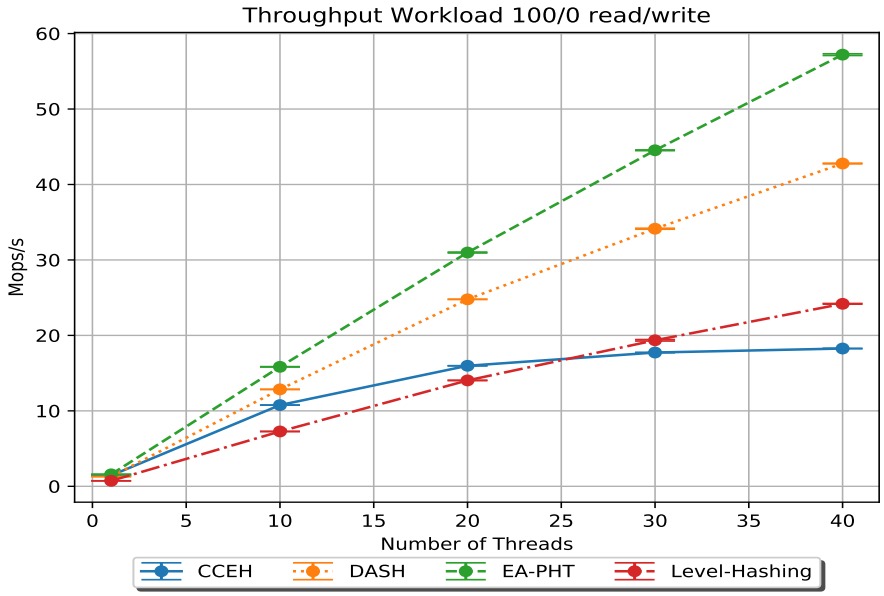


Figure 3.17: 100/0 read/write workload.

As shown in Figure 3.18, DASH performs better during the load phase, being 22% faster than EA-PHT. This was expected since EA-PHT needs to perform a full table rehash during resizing. The performance during the load phase between CCEH and EA-PHT is similar. Even though CCEH implements an alternative to avoid a full table rehash, EA-PHT takes advantage of the data locality by setting its bucket size to be 256 bytes. Level Hashing shows the lowest performance during the load phase since this implementation performs the rehashing of 1/3 of its buckets during a full table resize, and uses buckets with a size of 64 bytes.

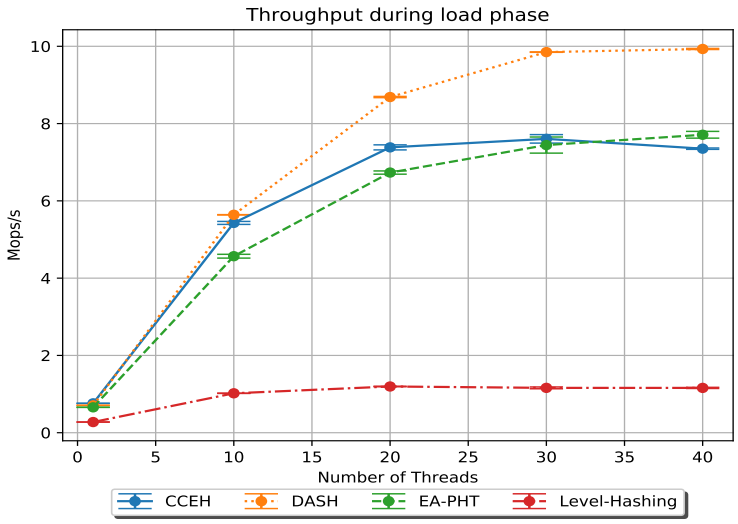


Figure 3.18: Load phase throughput.

| Implementation | M key-value pairs/s |
|---|---|
| DASH | Constant 57ms |
| CCEH | 2169 |
| EA-PHT | 49 |
| Flat Store | 25 |
| Level Hash | 0.57 |

Table 3.1: Recovery time performance comparison.

As shown in Table 3.1, the recovery time of EA-PHT is higher in comparison to implementations such as DASH [8], and CCEH [39], which have built specialized recovery mechanisms. This was expected since EA-PHT takes a greedy single-threaded approach when performing recovery, which requires the scanning of every key-value pair in every bucket contained in the current chunk. Compared to hybrid implementations such as HiKV [15], and FlatStore [68], which need to rebuild a portion of the data structure upon recovery, EA-PHT is faster.

When designing data structures for persistent memory, taking into consideration the access granularity of these devices is essential. Implementations such as DASH and EA-PHT, which consider the access granularity of these devices in their designs, perform

considerably better in all workloads. The overhead of a full table rehash in EA-PHT was not as high as expected as seen in Figure 3.18. During the load phase, EA-PHT manages to perform similarly to CCEH, which avoids a full table rehash by just rehashing small portions of the hash table when needed. The parallelization of the full table rehash operation of EA-PHT could further reduce its overhead.

As well, we have a hypothesis regarding the impact of the use of locks stored in persistent memory. Using the Linux Perf profiler tool, it was observed that in a 40 thread execution of workload A 50/50 read/write, 76% of all the cache references were cache misses. Flushing instructions are used to guarantee persistence, but as mentioned in Section 2.2, these instructions can invalidate the flushed cache line. A single-threaded benchmark comparing the different flushing instructions CLWB, CLFLUSH, CLFLUSHOPT was executed. The benchmark consisted of 1M iterations of a load and a store followed by one of the different flush instructions wrapped by MFENCE instructions.

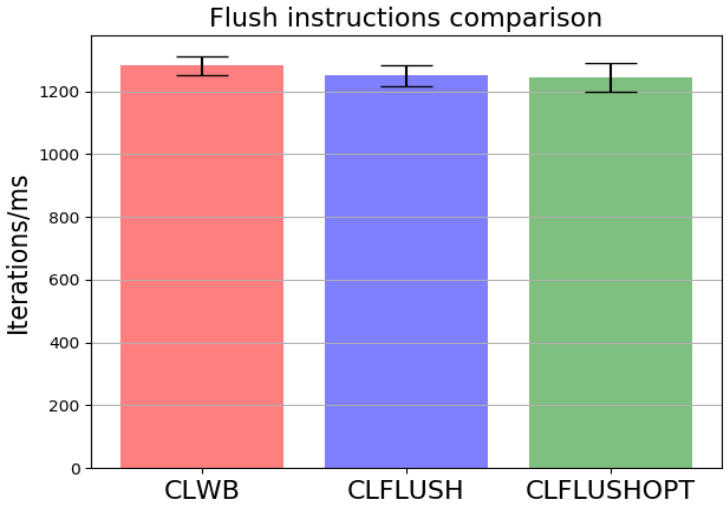


Figure 3.19: Comparison of flushing instructions.

The comparison of CLWB, CLFLUSH, CLFLUSHOPT shows that they perform similarly. Based on the previous results, we infer that containing locks in persistent memory can increase the number of cache misses if we consider the following scenario. If a process is waiting for the lock to be released by continuously reading it, and the thread that has acquired the lock flushes a key-value pair contained in the same cache line as the lock, it will force the waiting thread to re-load the cache line.

After the evaluation of EA-PHT, two areas of opportunity to improve its performance are identified. The first one is the migration of the bucket item locks to volatile memory. The second one is the design of a new resize mechanism, which allows multiple helpers to work in parallel during rehashing.

# Chapter 4

# Volatile Concurrency Control Variables

This chapter presents the changes made in the design of E-PHT to migrate the bucket item locks to volatile memory. This new design is called the embedded allocation persistent hash table hybrid version, EA-PHT-H.

## 4.1 EA-PHT-H design

As previously mentioned in the original description of CLHT-LB, only the locks in the contiguous array of buckets are used for concurrency control. Then the migration of the locks in the bucket items to volatile memory is straightforward since the size of the hash table item can be used to create a contiguous array of volatile locks.

The design of EA-PHT-H, includes a new item stored in volatile memory called the access item. This item holds direct pointers to a hash table item, the chunk pointed by the hash table item, and its corresponding contiguous array of volatile locks. The access item then serves as the entry to the data structure for the search, insert and remove operations, by providing all the necessary pointers needed. Also, the variables used for resizing were removed from the hash table items and now are part of this class's member variables, given that these variables are not needed during recovery.

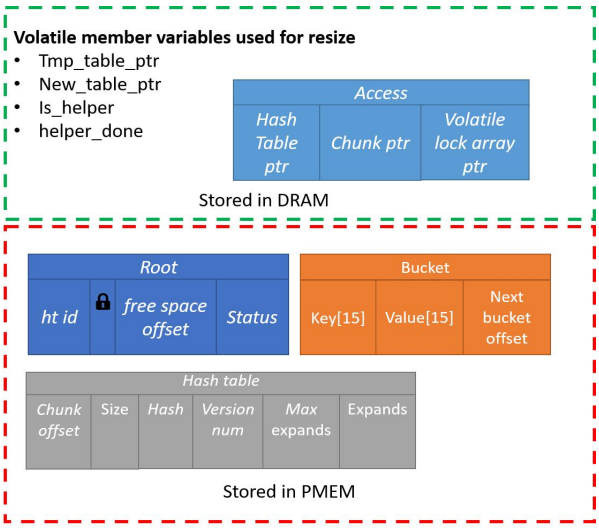


Figure 4.1: EA-PHT-H items and member variables.

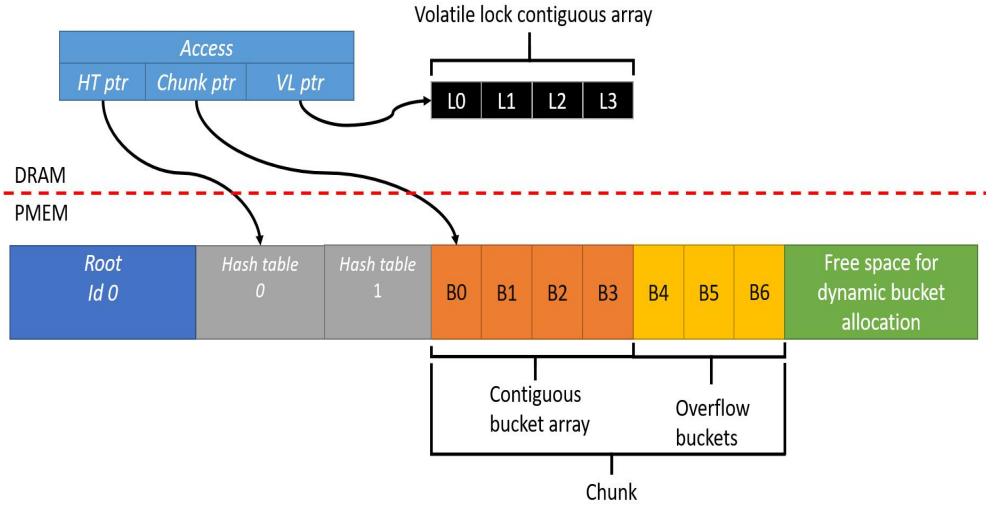


Figure 4.2: Representation of EA-PHT-H.

The design changes to incorporate volatile locks into EA-PHT-H involved minor changes in the code, and these are described below. Upon initialization or recovery, a new contiguous array of volatile locks needs to be allocated and initialized, as well a new access item needs to be allocated, and its direct pointers need to be assigned. The search, insert and

remove operations will no longer use the root item to access the data structure, and instead will use the provided pointers contained in the access item to perform their operations. For a full table resize, before beginning the rehashing of the table, a new contiguous array of the size of the new table needs to be allocated and initialized. After the rehashing is completed and the table is ready, but before finalizing the full table resize operation, a new access item is created, and its pointers are assigned. The pointer to the old access item will be atomically swapped for the new one, then the old volatile array of locks and access item are deallocated.

## 4.2 Performance evaluation

The experimental setup for the following results is the same as the one described in Section 3.7. In addition to the YCSB workload results, the Linux perf profiler tool was used to count the total cache references and the percentage of cache misses in the execution of workload A with 40 threads. It was observed that EA-PHT-H had 10% fewer cache misses in comparison to EA-PHT. These results support the hypothesis mentioned in Chapter 3, which states that having locks stored in persistent memory that share a cache line with key-value stores, increases the number of cache misses.

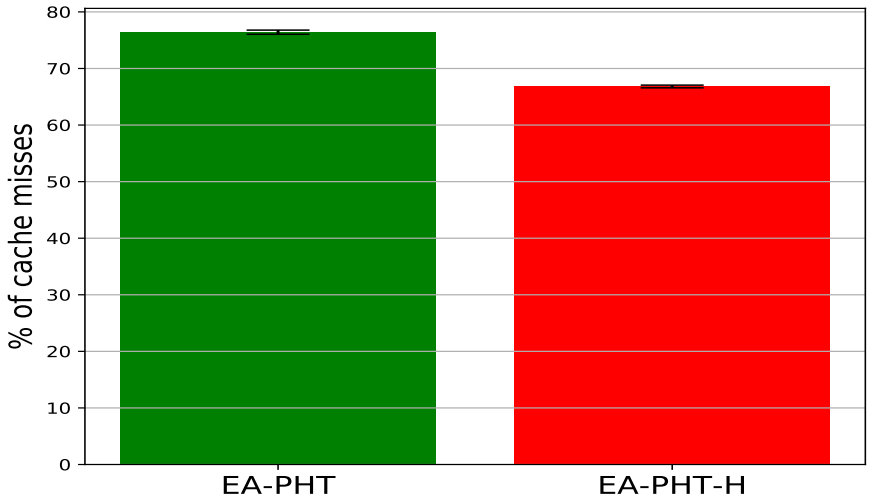


Figure 4.3: Percentage of cache misses in a 40 thread execution of workload A.

In Figure 4.4, we observe that EA-PHT-H benefits from having the bucket item locks in volatile memory in write-heavy workloads. In the 50/50 read/write workload, EA-PHT-H is up to 20% faster than EA-PHT and 46% faster than DASH. In the 80/20 read/write workload, EA-PHT-H is up to 7% faster than EA-PHT and 32% faster than DASH. These results further confirm that the scenario described next is the reason EA-PHT has lower throughput than EA-PHT-H. If a process is waiting for the lock to be released by continuously reading it, and the thread that has acquired the lock flushes a key-value pair contained in the same cache line as the lock, it will force the waiting thread to re-load the cache line.

The decision to migrate the locks to volatile memory does not come free of cost. For read dominated workloads, EA-PHT-H performance is lower than EA-PHT. Search operations in EA-PHT-H need to load data from both persistent and volatile memory before starting the operation. In this case, EA-PHT benefits from data locality in these workloads. In the 100/50 read/write workload, EA-PHT is up to 11% faster than EA-PHT-H, and in the 95/5 read/write workload EA-PHT, is up to 5% faster than EA-PHT-H.

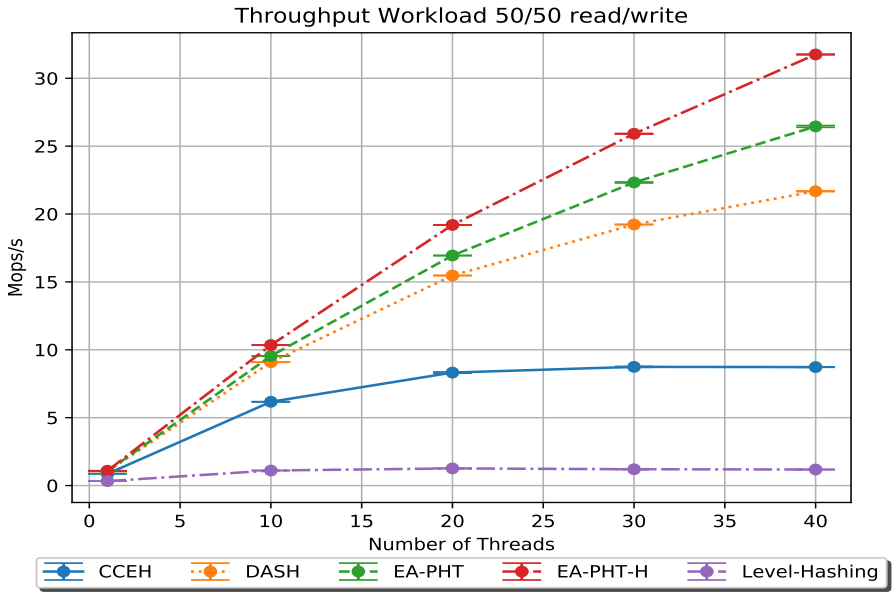


Figure 4.4: 50/50 read/write workload.

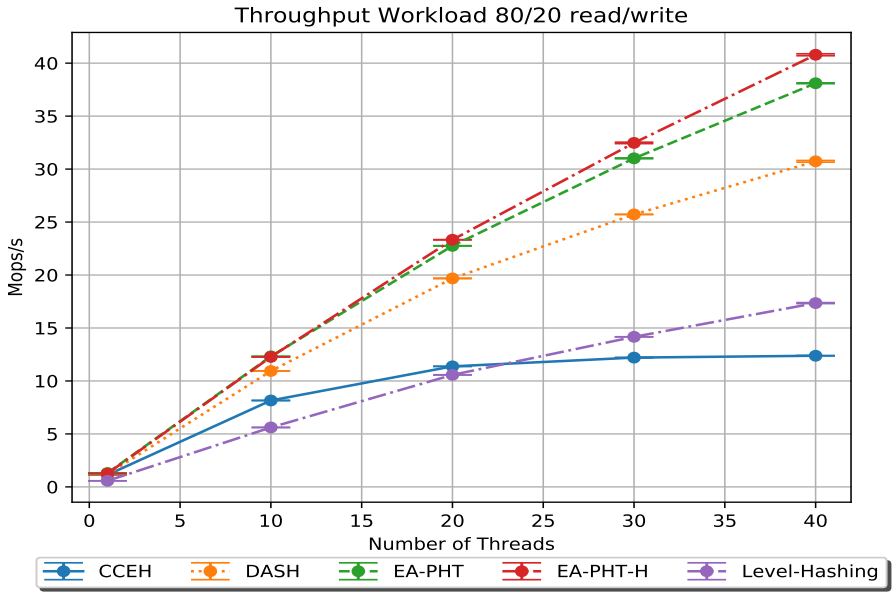


Figure 4.5: 80/20 read/write workload.

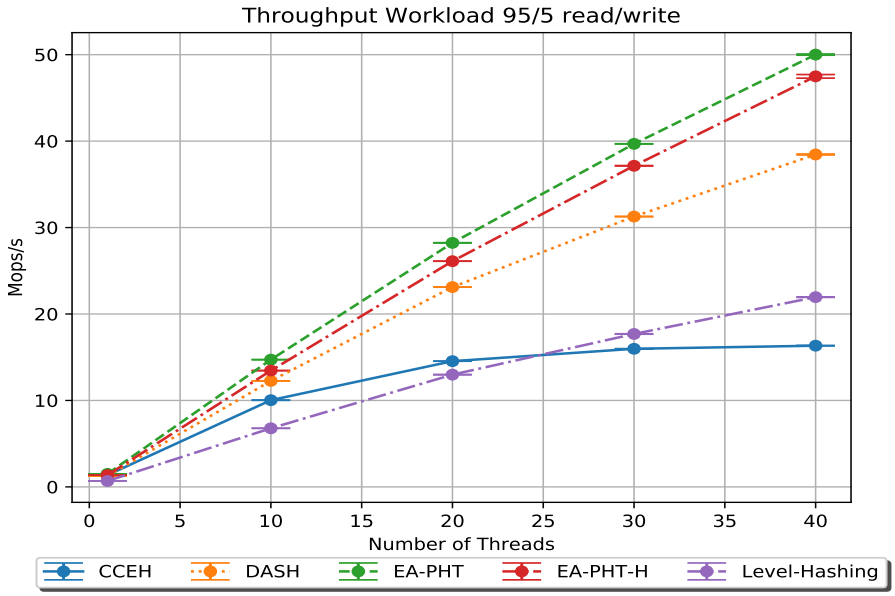


Figure 4.6: 95/5 read/write workload.

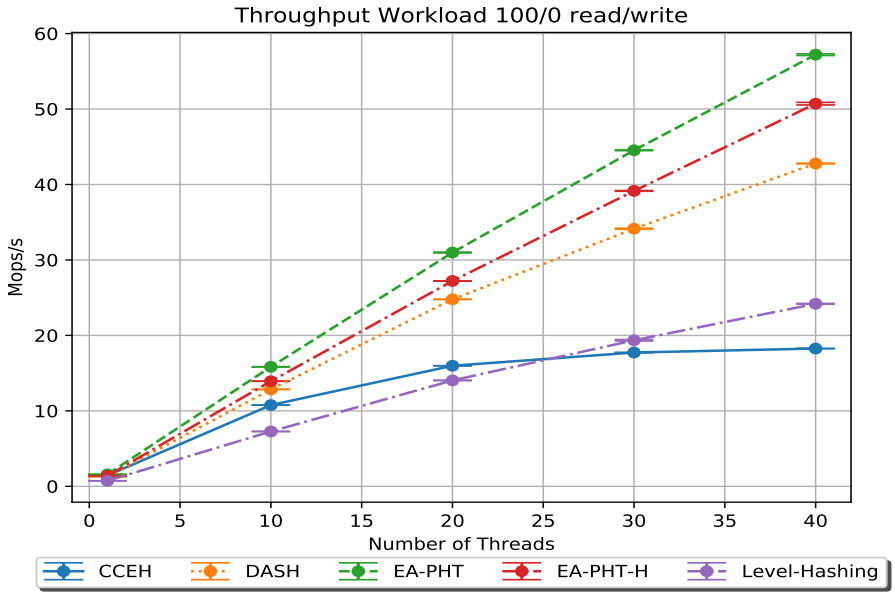


Figure 4.7: 100/0 read/write workload.

Presented below in Figure 4.8, the average time for the rehash operation in relation to the number of buckets contained in the hash table for EA-PHT and EA-PHT-H is shown. As the table size gets grows, the performance of EA-PHT-H degrades in comparison to EA-PHT, given that the thread performing the resize will need to initialize the whole contiguous array of volatile locks before performing the rehash of the table. Interestingly, the results shown in Figure 4.9, still show that EA-PHT-H performs better during the load phase, which is correlated to the reduced number of cache misses. EA-PHT-H is 10% faster than EA-PHT and 14% slower than DASH during the load phase.

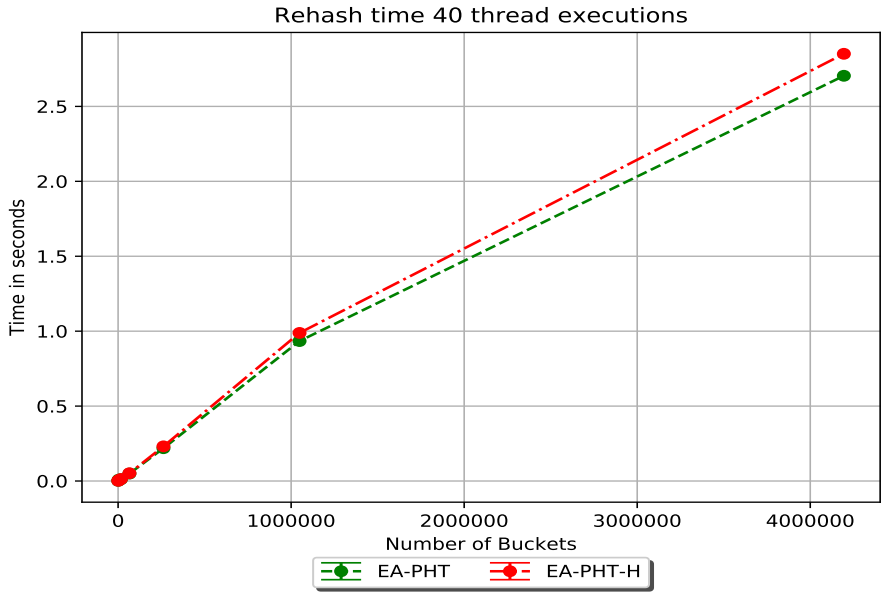


Figure 4.8: Average time in seconds spent in a full table rehash (lower is better).

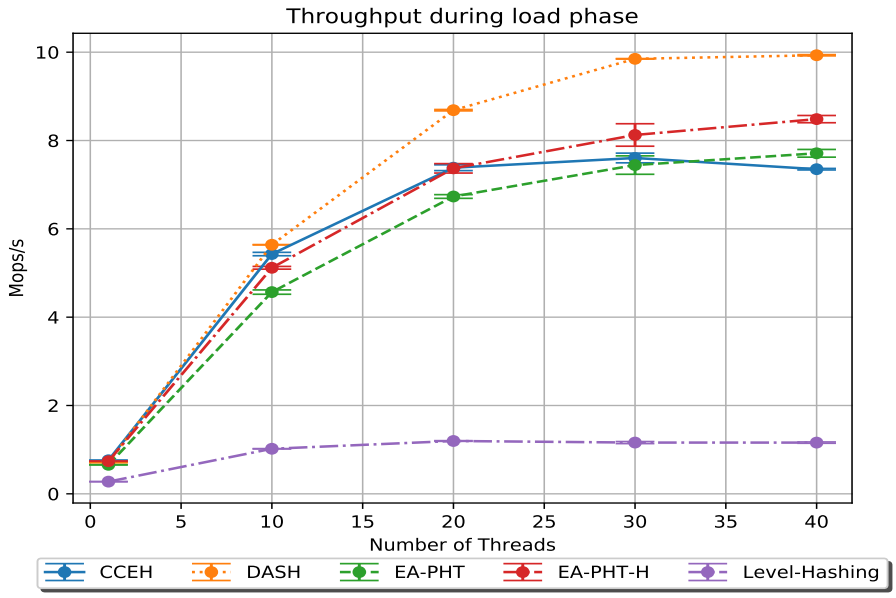


Figure 4.9: Load phase throughput (higher is better).

## 4.3 Conclusion

When implementing lock-based data structures for persistent memory, special care needs to be taken regarding the placement of locks. Storing locks in persistent memory alongside data that will be concurrently flushed by other processes will increase the number of cache misses by waiting threads, which are continuously checking if the lock is available. This specific case becomes a bottleneck for throughput in write-heavy workloads. While in the implementation of EA-PHT-H the solution was to migrate the locks to volatile memory, padding the locks to be the size of a cache line can be a viable solution, but comes with the drawback of wasted space. These results indicate that even though the CLWB flushing instruction allows some cache lines to remain valid in the cache after flushing them, in a concurrent shared memory scenario, this instruction can still have undesired effects.

The initialization of the contiguous array of volatile locks before the full table rehash operation increases the overhead of the table resize operation. Still, the improvement in performance for write-heavy workloads allows EA-PHT-H to perform better than EA-PHT during the load phase. Still, EA-PHT-H is outperformed by DASH during the load phase. The next chapter proposes a new resize mechanism to reduce the overhead of the rehash operation, which allows the parallelization of the rehash operation.

# Chapter 5

# The Parallel Resize Mechanism

This chapter presents the embedded allocation persistent hash table with hybrid parallel resize (EA-PHT-HPR). The proposed resize mechanism allows multiple threads to aid in the full table resize operation and performs the initialization of its volatile locks and the rehashing of the table in parallel.

## 5.1 EA-PHT-HPR design

For this specific implementation, the size, the maximum number threads allowed to help rehash the persistent section, and the maximum number threads allowed to help initialize the volatile section need to be a power of two. This is because the new resize mechanism needs to be able to provide equally sized sections, further explained in Section 5.2, and having all of these three variables be a power of two simplifies the problem. This design incorporates the addition of a second access item, which is used by the helper threads during resize. As well, a new set of member variables is introduced for the new resize mechanism. A brief description of the member variables is shown below, and an in depth explanation of their use during the resize operations is included in Section 5.2.

- **dram_section_id:** Used by helper threads to acquire a section of the volatile lock array.
- **pmem_section_id:** Used by helper threads to acquire a section of the table.
- **active_pmem_resizers:** Used to signal that threads are currently helping the rehashing of the table.

- **active_dram_resizers:** Used to signal that threads are currently helping the initialization of the volatile lock array.
- **tmp_lock_arr_ptr:** Provides access to the volatile lock array for threads helping in the initialization of the volatile lock array.
- **tmp_size:** Used to calculate the size of the section by threads helping in the resize of the volatile lock array.
- **flag_dram_resize:** Notifies threads that a new volatile array of locks needs to be initialized.
- **finish_flag:** Signals that the resize operation has finished.

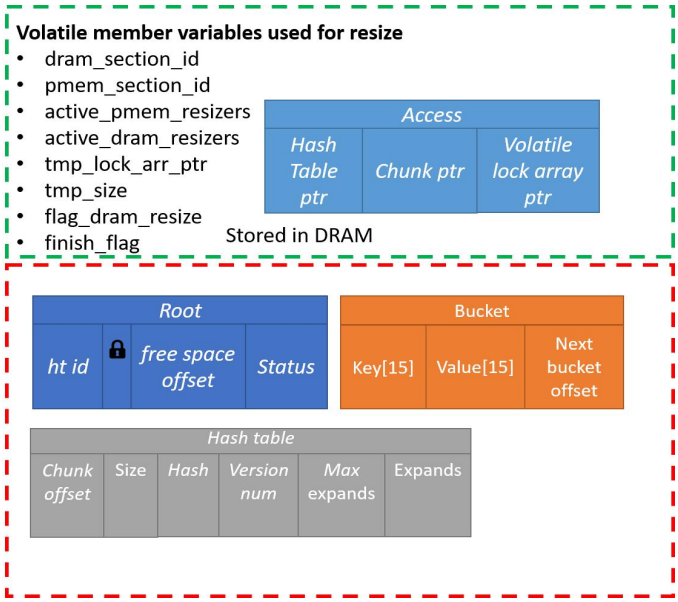


Figure 5.1: EA-PHT-HPR items and member variables.

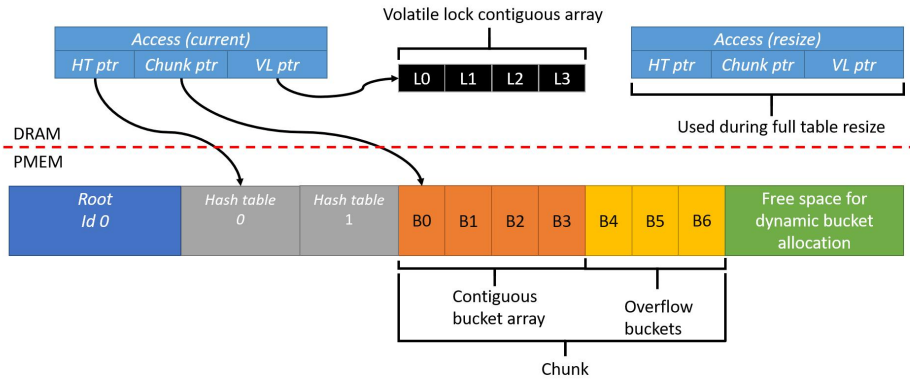


Figure 5.2: Representation of EA-PHT-HPR.

## 5.2 Parallel resize

When a full table resize is required, a new hash table with a size of a constant multiplied by the size of the old hash table will be created. The basic idea behind the resize mechanism is to recruit a small number of user threads for the initialization of the volatile locks, and a large number of user threads to rehash the hash table. In this implementation, there can be up to eight threads helping in DRAM, and up to thirty two helping in PMEM. This is a similar approach to the one presented in [63], in which servicing threads are allowed to aid the full table rehash during run-time. To designate reserved portions of the volatile lock array and the table, the notion of sections is introduced. These sections represent slots where a single thread can perform either the initialization or rehash operation.

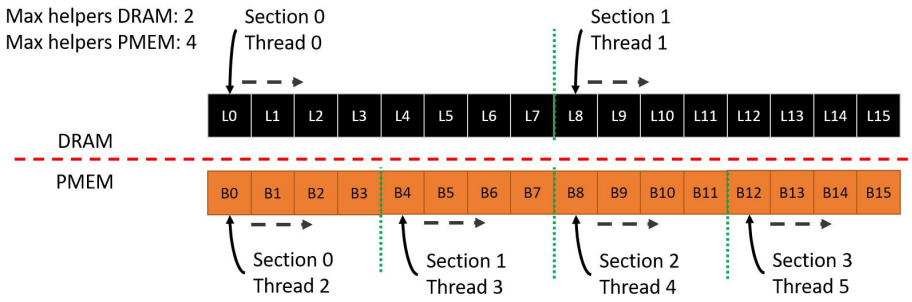


Figure 5.3: Section-based approach representation.

The resize operation is performed as follows. After a successful overflow insert, if the thread finds that the expand counter surpasses the max allowed expands, it will try to acquire the resize lock. If the thread finds the lock to be acquired, then the insert operation completes successfully and returns since another thread has already begun the resize operation. A thread that acquires the resize lock will proceed to initialize its member variables to zero, and deallocate any previous resize access item, if it exists. Then, it will proceed to call the *create_hash_table()* function, which will select the new hash table item to be used and initialize its variables. The *create_hash_table()* function first marks the old hash table item chunk offset by setting/writing the MSB of the chunk offset to one. After that, no other thread may allocate overflow buckets in the old chunk. Then the root id is used to determine which is the id of the new hash table item to be used, and the direct pointer to the new hash table item is calculated from the id. If the size variable in the new hash table item contains a value greater than zero, this item already points to a previous chunk; then the previous chunk is deallocated by issuing a *pmem_memset_persist()* to set the entire chunk to zero.

After clearing the previous chunk pointed by the new hash table item, if it existed, the size variable is then written with the new size of the table. Then, the chunk offset is calculated and written. To calculate the chunk offset for the new hash table, the old hash table chunk offset, table size, and expand counter are used. Then, the expand counter is written to zero. Finally, the max expands, and hash variables are initialized, without requiring flushing since these variables are always initialized upon recovery.

After the initialization of the new hash table item and just before beginning the rehash operation, the thread performing the resize needs to re-check the calculated offset, because there could be a thread which sees the chunk unlocked and proceeds to allocate. Then, the thread performing the resize marks the chunk offset and reads the expand counter before the increment by the thread allocating the overflow bucket. This way, it can be ensured that there are no overlapping chunks. After the *create_hash_table()* function has initialized the new hash table, the size of the new table is written to the tmp_size variable, and a contiguous array of volatile locks of said size is allocated. This new contiguous array of volatile locks will be pointed to by the tmp_lock_arr_ptr variable. Then, the flag_dram_resize variable is set to one to signal threads to begin the initialization of the volatile lock array.

The thread performing the resize will continue without helping with the initialization of the volatile lock array. Any thread that tries to acquire a lock will first check the flag_dram_resize variable before continuing with its operation, (Algorithm 8 line 2). If the variable is set to one, then the active_dram_resizers variable is atomically incremented with a *_sync_add_and_fetch()* instruction, and the *help_init_locks()* function is called (Algorithm 8 lines 3 - 4). Inside the *help_init_locks()* function, a helping thread will try to

acquire a volatile section to initialize. To do this, the dram_section_id variable is used. The dram_section_id variable is protected by a lock. When the lock is acquired, if the dram_section_id variable value is lower than the maximum allowed helpers for DRAM, the dram_section_id variable is incremented by one, and its previous value represents the section to be initialized, (Algorithm 9 lines 5 - 7). A thread that finishes initializing its section will try to acquire a new section if possible, (Algorithm 10 line 10). If the value of dram_section_id is equal to the maximum allowed helpers for DRAM, then all sections are currently being initialized by other threads, and the thread proceeds to write a zero to the flag_dram_resize, signaling that all the sections are taken, and no further help is needed, (Algorithm 10 line 12). After exiting the *help_init_locks()* function, the active_dram_resizers variable is atomically decremented using a *_sync_sub_and_fetch()* instruction, (Algorithm 8 line 5).

Meanwhile, the volatile locks are being initialized, the thread performing the resize will allocate a new resize access item and proceed to assign the direct pointers of the new table. The thread performing the resize will use the pmem_section_id variable to acquire a section to rehash. The pmem_section_id variable is protected by a lock. When the lock is acquired, if the pmem_section_id variable value is lower than the maximum allowed helpers for PMEM, the pmem_section_id variable is incremented by one, and its previous value represents the section to be rehashed, (Algorithm 9 lines 5 - 7). If the value of pmem_section_id is equal to the maximum allowed helpers for PMEM, then all sections are currently being rehashed by other threads. The main thread performing the resize operation will wait for the active_dram_resizers and active_pmem_resizers variables to be zero, which will signal that both the initialization of locks and rehashing of the new table has finished.

After finishing rehashing its current section, the thread performing the resize operation will try to acquire a new section to rehash if possible. During the rehashing, search operations can be executed normally. Any concurrent threads performing an insert or remove that do not find the current bucket resize lock acquired will be able to proceed with its operation. If the bucket resize lock is set, (Algorithm 8 line 15), the thread will help to rehash, if possible, or wait for the resize operation to finish. To begin helping in the rehashing, the active_pmem_resizers variable is atomically incremented using an *_sync_add_and_fetch()* instruction, and the *help_rehash_table()* function is called. In the *help_rehash_table()* function, helping threads will use the pmem_section_id variable to acquire a section to rehash in the same way as previously described above. After exiting the helping function, the active_pmem_resizers variable is atomically decremented using a *_sync_sub_and_fetch()* instruction, and the thread waits for the resize operation to complete (Algorithm 8 lines 18 - 22). After finishing rehashing its current section, the helping thread will try to acquire a

new section to rehash if possible, (Algorithm 11 line 12). The rehash operation traverses the buckets from the reserved section along with any overflow buckets linked and copying all their key-value pairs to the new hash table. To rehash a bucket, the resize lock must be acquired first. This lock is acquired and never released. The resize lock and update lock are mutually exclusive, so a resize lock will only be acquired if neither lock is set. When the rehashing is finished, the contiguous array of bucket items along with any overflow buckets that were allocated during rehash in the new chunk are flushed. After the data is flushed, the old hash table item version number plus one is written to the new hash table item. Then, the root id is atomically swapped for the id of the new hash table item. Then, a new access item is allocated and assigned the pointers of the hew hash table. The pointer of the access item used by search, insert and remove operations is atomically swapped for the updated access item pointer. Then, the old access item is deallocated along with its old referenced volatile lock array. Finally, the finish_flag is set to one, signaling the waiting threads that the resize operation has finished, and the resize lock is released, thus completing the resize operation.

**Algorithm 8** Acquire bucket lock function

```
1: procedure LOCK_ACQ(LOCK,HT)
2:     if flag_dram_resize then
3:         _sync_add_and_fetch(active_dram_resizers)
4:         help_init_locks(tmp_lock_arr_ptr,tmp_size)
5:         _sync_sub_and_fetch(active_dram_resizers)
6:     lock_s = *lock
7:     lock_free= lock_s& VERSION_MASK
8:     lock_update = (lock_s & VERSION_MASK) | UPDATE_LOCK
9:     while !CAS(lock, lock_free,lock_update) do
10:        if lock_s & RESIZE_LOCK then
11:            break
12:        lock_s = *lock
13:        lock_free= lock_s & VERSION_MASK
14:        lock_update = (lock_s & VERSION_MASK) | UPDATE_LOCK
15:    if lock_s & RESIZE_LOCK then
16:        _sync_add_and_fetch(active_pmem_resizers)
17:        help_rehash_table(root_access_item→ht)
18:        _sync_sub_and_fetch(active_pmem_resizers)
19:        while finish_flag=0 do
20:            pause() // Intel intrinsic instruction _mm_pause()
21:            MFENCE()
22:        return 0
23:    return 1
```

**Algorithm 9** Acquire Section

```
1: procedure GET_SECTION()
2:     section = 0
3:     while get_section_lock()==0 do
4:         // repeat until section lock acquired
5:     if section_id < MAX_HELPERS then
6:         section = section_id
7:         section_id++
8:     else
9:         section = MAX_HELPERS
10:    release_section_lock()
11:    return section
```

**Algorithm 10** Lock initialization helper function

```
1: procedure HELP_INIT_LOCKS(LOCK_ARR_PTR,SIZE)
2:     get_section:
3:     section_id = get_volatile_section()
4:     if section_id < MAX_HELPERS_DRAM then
5:         start = section_id*(SIZE/MAX_HELPERS_DRAM)
6:         end = (section_id+1)*(SIZE/MAX_HELPERS_DRAM)
7:         for (i=start; i<end; i++) do
8:             LOCK_ARR_PTR[i].lock = 0
9:         if section_id < MAX_HELPERS_DRAM then
10:            goto get_section
11:    else
12:        flag_dram_resize = 0
```

**Algorithm 11** Rehashing helper function

```
1: procedure HELP_REHASH_TABLE(OLD_HT)
2:     get_section:
3:     section_id = get_persitent_section()
4:     if section_id < MAX_HELPERS_PMEM then
5:         start = section_id*(OLD_HT→size/MAX_HELPERS_PMEM)
6:         end = (section_id+1)*(OLD_HT→size/MAX_HELPERS_PMEM)
7:         chunk = get_chunk_ptr(OLD_HT→chunk_off)
8:         for (i=start; i<end; i++) do
9:             bucket = get_bucket_ptr(chunk, i)
10:            rehash_bucket(bucket,resize_acces_item, i)
11:        if section_id < MAX_HELPERS_PMEM then
12:            goto get_section
```

## 5.3 Performance evaluation

The experimental setup for the following results is the same as the one described in Section 3.7. In addition to the YCSB workload results, the Linux perf profiler tool was used to count the total cache references, and the percentage of cache misses in the execution of workload A with 40 threads. Since the changes to the search insert and remove operations between EA-PHT-HPR and EA-PHT-H are virtually none, their performance is essentially the same.

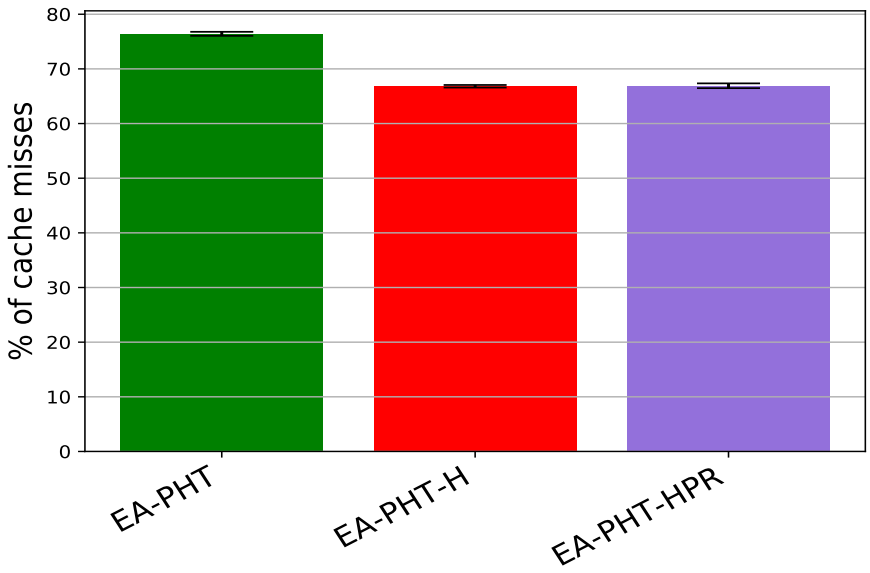


Figure 5.4: Percentage of cache misses in a 40 thread execution of workload A.

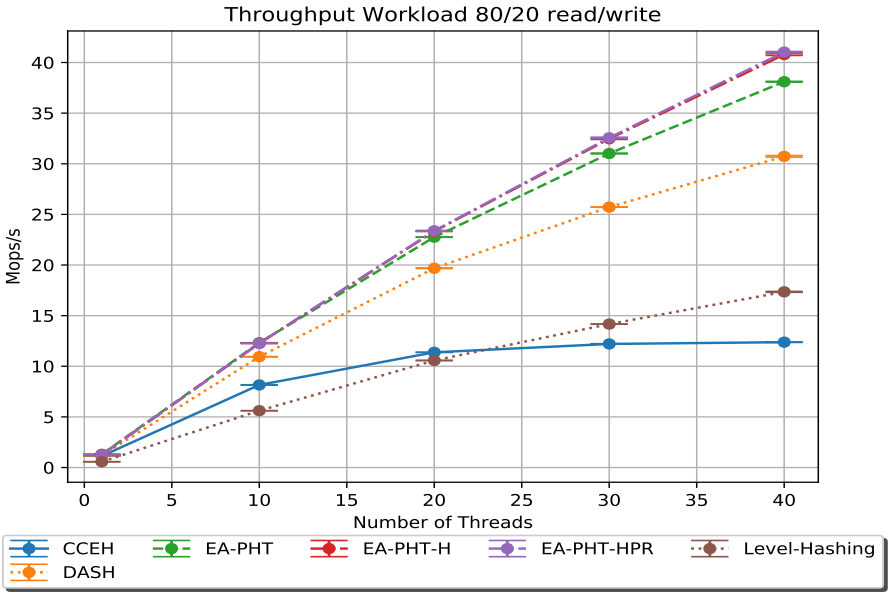


Figure 5.5: 80/20 read/write workload.

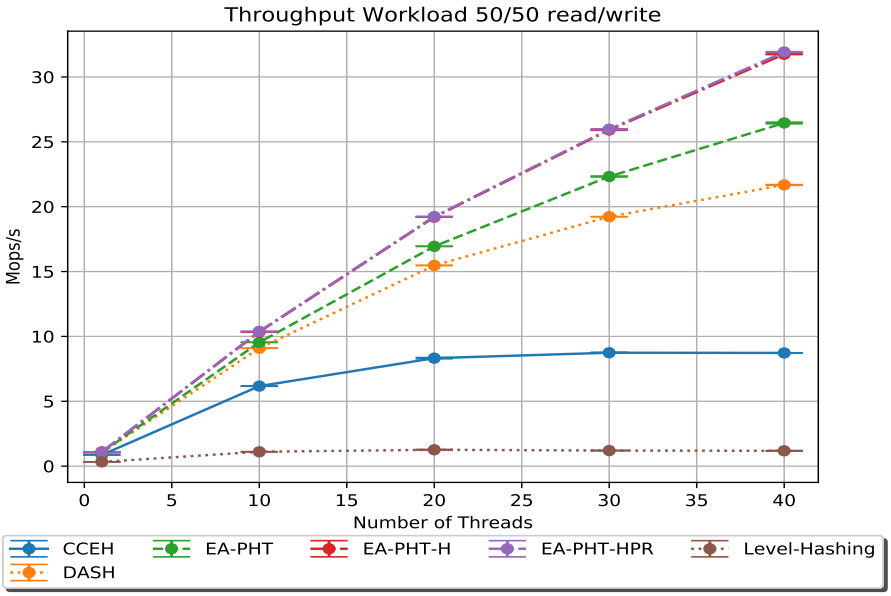


Figure 5.6: 50/50 read/write workload.

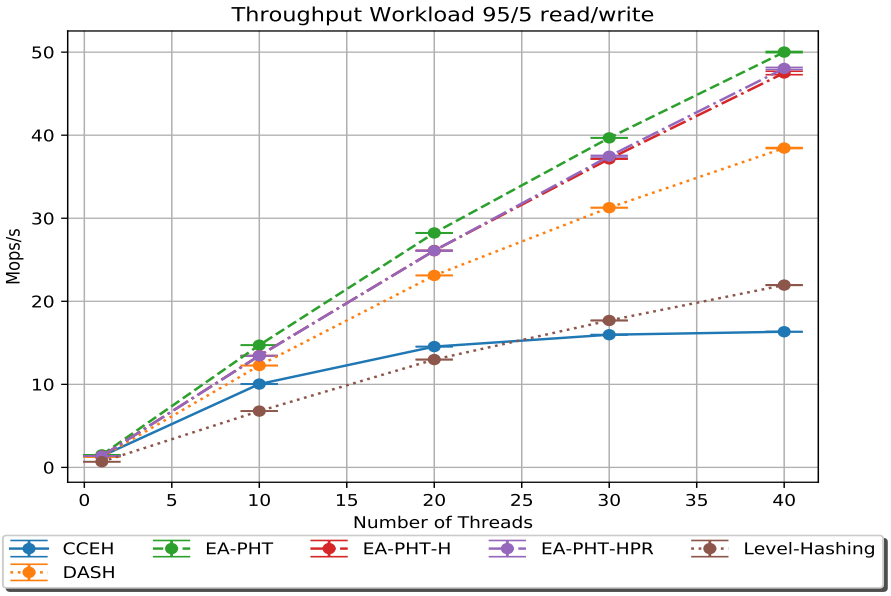


Figure 5.7: 95/5 read/write workload.

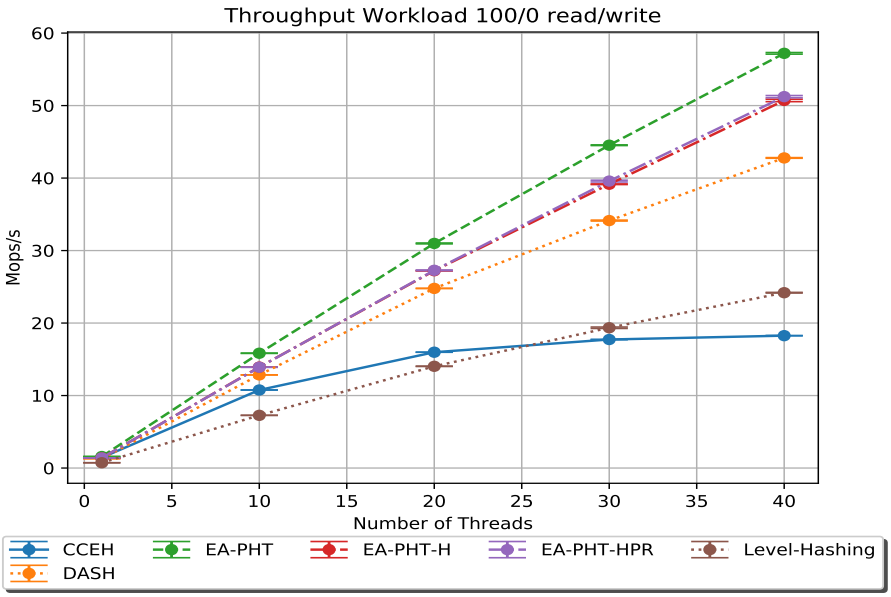


Figure 5.8: 100/0 read/write workload.

Shown in Figure 5.9, the overhead of the rehash operation is drastically reduced by the parallel resize mechanism, which allows multiple threads to aid in the full table resize operation and performs the initialization of the volatile locks and the rehashing of the persistent table in parallel. For the shown results, up to eight threads can help in the initialization of volatile locks, and up to thirty-two threads can help in the rehashing of the table. The parallelization of the resize mechanism allows EA-PHT-HPR to perform a full table resize on a table containing a little over 4 million buckets in roughly 1 second, while EA-PHT and EA-PHT-H take over 2.5 seconds. The reduced overhead of the full table rehash operation allows EA-PHT-HPR to be up to 29% faster than DASH, 51% faster than EA-PHT-H, and 66% faster than EA-PHT, during the load phase.

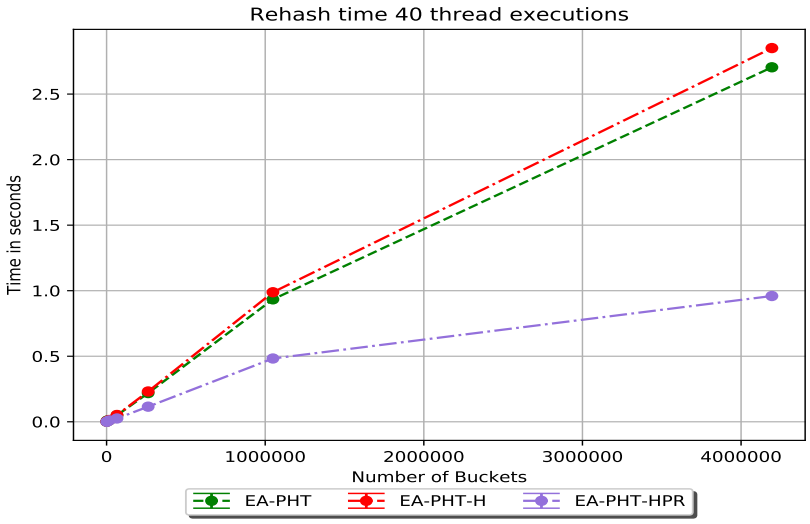


Figure 5.9: Average time in seconds spent in a full table rehash (lower is better).

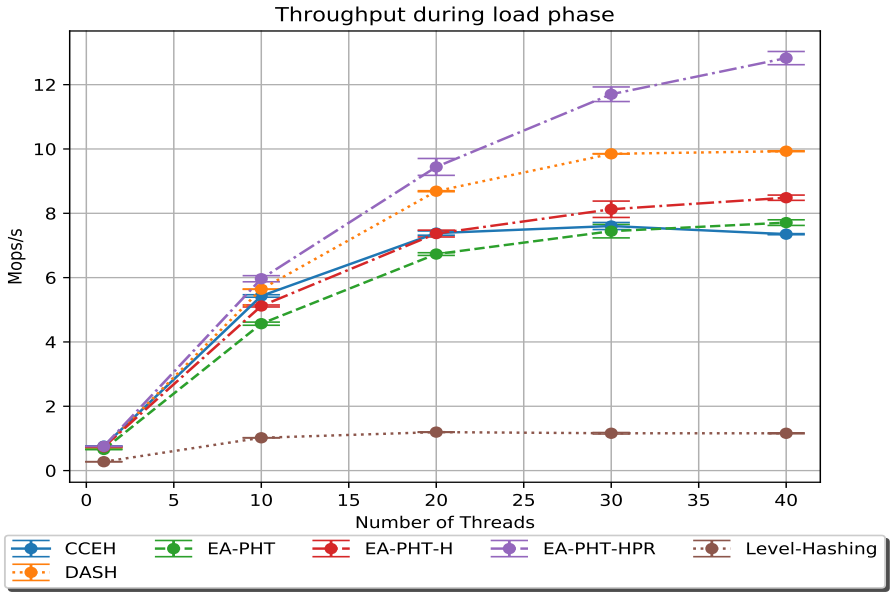


Figure 5.10: Load phase throughput (higher is better).

The performance of the recovery function for the different implementations is shown in table 5.1. There is a slight increase in the performance of EAPHT-H and EAPHT-HPR in contrast to EA-PHT. This is the result of migrating the locks to volatile memory, which reduces the number of stores to persistent memory during recovery.

| Implementation | M key-value pairs/s |
|---|---|
| DASH | Constant 57ms |
| CCEH | 2169 |
| EA-PHT-HPR | 52 |
| EA-PHT-H | 52 |
| EA-PHT | 49 |
| Flat Store | 25 |
| Level Hash | 0.57 |

Table 5.1: Recovery time performance comparison.

## 5.4 Conclusion

The design and implementation of EA-PHT-HPR, which is based on CLHT-LB [61, 62], achieves high performance and scalability while maintaining a similar design to that of its original DRAM implementation. EA-PHT-HPR, being the successor of EA-PHT-H, uses embedded allocation and volatile concurrency control with the addition of a hybrid parallel resize mechanism. The use of an embedded allocation technique reduces the number of stores to persistent memory and reduces the overhead of allocation. The use of volatile locks reduces the cache misses in write-heavy workloads, by isolating the locks from the cache lines containing data that needs to be flushed. Finally, the parallel resize mechanism is used to reduce the overhead of the full table resize. This mechanism demonstrates that procedures that would be considered expensive in persistent memory, in our case, a full table rehash, can benefit considerably from parallelization.

# Chapter 6

# Linearizability testing

This chapter will cover the experimental setup and mathematical principles used for linearizability testing, and the results obtained. The EA-PHT, EA-PHT-H, and EA-PHT-HPR implementations follow the correctness property of strict linearizability [38], which requires that pending operations complete or abort as of the crash. To follow strict linearizability, the recovery mechanisms in these implementations ensure that any pending operations left after the crash are completed or aborted, as described in Section 3.6. The linearizability tests include both failure-free and power failure scenarios. The specifics on how the power failure logs are generated and how tests are analyzed are explained in Section 6.2.

## 6.1 Experiment setup

The experimental setup for linearizability testing is composed of four elements:

- **Benchmark:** Executes multithreaded workloads on the hash tables to be tested.
- **Logger:** Used to capture the invocations and responses of operations. The logger used must be power-fail safe, meaning that in the event of a power failure the log and all its entries can be recovered.
- **Linearizability checker tool:** Given an execution history of concurrent operations, the tool must be able to determine if this execution is linearizable.
- **Crash script:** Used to trigger a power failure by power cycling the server.

The benchmark used to generate the different workloads for the hash tables is based on the benchmark provided by the libcuckoo library [37]. This benchmark allows us to define the percentage of read, insert, and delete operations, the total number of operations to be performed, and the initial capacity of the table.

The benchmark is instrumented to capture all the invocations and responses of operations performed on the hash table in a log file. For invocations, we capture: the timestamp, process id, invocation indicator, and the key. For insert operations, the value is also captured. For responses, we capture the timestamp, the process id, the response indicator, and a true/false statement stating if the operation succeeded. A read operation will look for the requested key and return the current stored value, if and only if the key-value pair has been previously inserted in the table. Thus a read can succeed and find a value that was previously inserted, or fail, not finding the requested key and returning false. Only for a successful read operation, the found value is captured in the response.

Efficient and fast logging is essential when testing for linearizability, as the overhead induced by logging operations can affect the interleaving of steps and mask some concurrency bugs. To avoid contention among threads when appending entries to a single log, a partitioned logger approach is used. Each thread creates a private log of its own invocations and responses. Since some of the linearizability tests will require the system power cycle, we require the logs to be persistent. To achieve this the libpmemlog library [22] is used. This library equips us with fail-safe appends to the log. In addition, this library uses Direct Access (DAX), which eliminates the use of the page cache and allows us to write persistent memory directly without involving the kernel, further reducing the overhead induced by logging. The per-thread logs are then merged and sorted by a secondary program after the instrumented code finishes executing, using the event timestamp to determine the order of events.

The creation of histories relies on timestamps to order the entries of the log partitions. Using instructions that read the processor timestamp counter, such as RDTSC and RDTSCP, does not suffice for our requirements, since these instructions may be reordered by the processor. To ensure that the instructions reading the processor timestamp counter will not be reordered, we need to use memory fence instructions. We used the timer library from the HighwayHash project [29], which conveniently implements this functionality.

The linearizability checker tool used is Ahorn [1]. It is important to note that this tool does not handle logs that contain a crash. The crash events need to be externally handled, as explained in Section 6.2, in order to provide a log which can then be analyzed by the tool. To trigger a power failure, a remote script is executed via SSH within the benchmark. This script power cycles the server. The script communicates with the Integrated Dell Remote

Access Controler (iDRAC). The iDRAC is an embedded computer independent of the server, and module allows the server to be power cycled remotely.

## 6.2 Handling power failures in logs

When generating logs with a power failure, each thread creates two logs: the first log prior to the crash and the second log after the crash. The first log will likely have some pending operations, meaning a process only appended an invocation to the log and is missing a response. Since the linearizability checker is not equipped to handle the following scenario, some extra steps need to be taken to handle pending operations prior to merging both logs. Following the correctness property of strict linearizability [38], described in Section 2.3, we need to determine if the pending operations took effect before the crash, in order to complete or abort these operations.

Read and insert workloads were tested, where each value inserted to the table is unique. We ensure that every process uses unique values for all insert operations across both executions, before the crash and after recovery. With these unique values captured in the logs, determining if an incomplete insert completed or aborted in the presence of a power failure is reduced to locating the unique values for pending insert operations on the second log, as described below. To generate unique values, we divide the 8 bytes of the value into three different variables, as shown in Figure 6.1: the process id, the test type, and a per-thread counter.

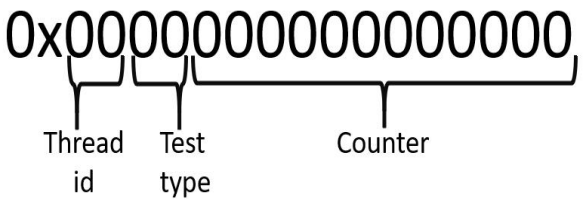


Figure 6.1: Unique value composition.

The logs capture the invocation and response events of all operations performed by the threads. Thus, we capture a history of invocations and responses in a log. For all the given threads in the history, their last invocation or response appended to the first log needs to be obtained. To test the entire history for strict linearizability, for all threads whose last append to the first log was an invocation, we need to determine if the operation completed

or aborted, and either add the response event or remove the invocation. Any pending read operations will be removed from the log. Since read operations do not modify the data structure, it is safe to remove them, which is similar to the approach taken by [10]. For a pending insert operation, the value proposed by the insert operation is searched in the second log. If the value is found, then a matching response event is added just before the crash. If not, then we abort the operation and remove it from the log.

As mentioned above, all pending read operations are aborted, eliminating the case of pending reads that are affected by a pending insert. Then we only need to determine if pending inserts are aborted or completed. Since inserts are only able to insert a key-value pair if the key has not previously been inserted, it is not possible to have two pending inserts on the same key succeed. Based on the previously stated, it is known that none of the pending operations will depend on another pending operation. Since we will only add response events to the log, then the order of insertion of matching response events for the incomplete operations does not matter. The generated log can now be parsed into sub-logs. Each sub-log contains all the operations for a subset of keys, which are selected by range, for example, a sub-log containing all operations for keys in the rage of 1 to 100. This is done since Ahorn [1] can only analyze a limited keyspace.

Since linearizability [40] is a local property, we can state that if all of the generated sub-logs are linearizable, the generated log is also linearizable. The generated log hides the presence of a crash by eliminating the crash event and determining which pending operations are completed (added response) or aborted (removed from the log), giving the illusion of a single continuous execution. The construction of the generated log ensures that if the history represented by the generated log is linearizable, then the original execution containing a crash is strictly linearizable. This statement can be formalized and proved rigorously, but it is out of the scope of this thesis.

## 6.3 Results

The testbed used for the following results was an Intel Xeon Gold 6230 server with Intel Optane Persistent Memory. The linearizability checking tool Ahorn [1] was used to analyze both failure-free and power failure scenarios.

For all the performed tests, the hash table initial size is set to 128 buckets, and the range of keys to be inserted, deleted, or removed is set to be 50K. This will force the implementations to resize during runtime. To analyze each test, sub-logs are generated from the log by selecting a range of keys. Then these sub-logs are analyzed with the

linearizability checking tool Ahonr [1] separately. If all logs are linearizable, then we can conclude the execution is strictly linearizable.

For the failure-free executions, a mixed workload of read, insert, and remove operations was tested. The test was performed ten times and contained over 1M operations. The linearizability checker results are shown in Table 6.1 below. We found that the Level Hashing [45] version used in [8] provides non-linearizable executions.

In an attempt to determine if delete operations introduced the linearizability bug, the Level Hashing [45] implementation was also tested for insert and read workloads. Still, the results provided non-linearizable executions; finally, the implementation was tested on a smaller keyspace, which does not trigger any resizes during execution. The final results showed that insert, delete, and read workloads histories are still non-linearizable, while an insert and read workload provided linearizable executions. This leads us to believe that the implementation may have two concurrency bugs, one in the table resize code and another in the delete operation.

To ensure the linearizability checker provides valid results, a bug was introduced to the EA-PHT implementation to determine if the analyzer can accurately detect linearizability violations. To add this bug that will cause non-linearizable executions, we removed the locking functions from the insert and delete operations. Then we ran ten linearizability tests, and the analyzer determined all of them non-linearizable, further reinforcing the finding of a linearizability bug in the Level Hashing implementation.

| Implementation | 50%R,25%I,25%D | # of tests | Non-Linearizable |
|---|---|---|---|
| EA-PHT | ✓ | 10 | 0 |
| EA-PHT-H | ✓ | 10 | 0 |
| EA-PHT-HPR | ✓ | 10 | 0 |
| Level-Hashing | X | 10 | 10 |
| CCEH | ✓ | 10 | 0 |
| DASH | ✓ | 10 | 0 |

Table 6.1: Failure-free executions, a ✓denotes all tests produced linearizable executions while a X denotes one or all tests produced non-linearizable executions.

| Implementation | 75%R,25%I | # of tests | Non-Linearizable |
|---|---|---|---|
| EA-PHT | ✓ | 30 | 0 |
| EA-PHT-H | ✓ | 30 | 0 |
| EA-PHT-HPR | ✓ | 30 | 0 |

Table 6.2: Simulated power failure executions, a ✓denotes all tests produced linearizable executions while a X denotes one or all tests produced non-linearizable executions.

For the power failure executions, a mixed workload of read and insert operations was tested. Each execution of the workload includes a power failure. The power failure entails the benchmark itself calling the remote shutdown script, causing the benchmark to be interrupted. The histories are created, as explained in Section 6.2. All tests on the three different given implementations provided linearizable histories in the presence of power failures. The results of this section indicate that EA-PHT, EA-PHT-H and EA-PHT-HPR can produce linearizable executions in both failure-free and power failure scenarios.

The allocation mechanism design allows the implementation to be free of memory leaks, as mentioned in Subsection 3.6.3, since any allocated buckets that remain detached for the data structure will still reside in the chunk and be reclaimed eventually when the chunk is deallocated. The recovery mechanism explained in Section 3.6 is executed when we detect that the previous execution terminated unexpectedly, and ensures liveness by verifying the table is in a consistent state and by fixing partial operations and reinitializing concurrency control variables. The implemented embedded allocation technique reduces the complexity of allocation since the type and size of the items that need allocation is known. In addition to the defined set of items, the only dynamically allocated items are buckets; thus, the cases where we can have a memory leak are specified and handled by the recovery mechanism.

# Chapter 7

# Conclusions and future work

This is the last thesis chapter covering the conclusions of the research and several areas of opportunity that can be explored to refine the provided implementations.

## 7.1 Conclusion

When designing data structures for persistent memory, taking into account the access granularity of these devices, 256 bytes, is essential. As shown in the previous results in Sections 3.8, 4.2, 5.3, implementations that take into consideration the access granularity of these devices tend to have better performance than implementations that do not. When designing data structures for persistent memory, the placement of data also needs to be handled with care. Variables that are accessed by multiple processes that share a cache line are prone to increase the number of cache misses if data is being flushed on the same cache line. This was the case with our lock in the design of the hash table. The volatile concurrency approach described in this thesis solves the previously stated problems but adds a new level of indirection when performing operations. Finally, parallelization of operations, in this case, a full table resize, was shown to be a viable way to alleviate the latencies of write-heavy operations considerably, in our case the full table resize operation. The embedded allocation technique has been proven to be a viable solution when trying to reduce the overhead of allocation in persistent memory. Still, one of the drawbacks of this technique is that it needs to be implemented differently depending on the data structure.

## 7.2 Future work

Future work in this research towards the development of persistent data structure may include the following directions:

1. Exploration of different data structure designs that could incorporate and benefit from embedded allocation and parallel operations in persistent memory.
2. Exploration of NUMA-aware embedded allocation, as the current version is not designed to account for NUMA-awareness.
3. Adding support for variable-length keys. This can be accomplished by storing pointers to variable-length keys—an approach that has been used by several other implementations such as [8, 20, 25, 39, 45].
4. The addition of the fingerprinting technique used in DASH [8], to improve the negative search operation latency.

# References


[1] A. Horn, D. Kroening. Faster linearizability checking via P-compositionality. In *Proc. of the International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, FORTE'15, pages 50–65, April 2015.

[2] A. Memaripour, J. Izraelevitz, and S. Swanson. Pronto: Easy and fast persistence for volatile data structures. In *Proc. of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 789–806, March 2020.

[3] A. Rudoff. Persistent memory programming. *login: The USENIX Magazine*, 42(2):32–40, 2017.

[4] A. Sharma and K. Roy. 1T Non-Volatile Memory Design Using Sub-10nm Ferroelectric FETs. *Proc. of the IEEE Electron Device Letters*, 39(3):359–362, March 2018.

[5] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proc. of the International Conference on Management of Data*, SIGMOD'18, pages 275–290, May 2018.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154, June 2010.

[7] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 371–384, April 2013.

[8] B. Lu, X. Hao, T. Wang, and E. Lo. Dash: Scalable Hashing on Persistent Memory. *arXiv e-prints*, March 2020. arXiv:2003.07302.

[9] Intel Corporation. *Intel 64 and IA-32 architectures software developer's manual*, 2016.

[10] D. Cepeda, S. Chowdhury, N. Li, R. Lopez, X. Wang, and W. Golab. Toward Linearizability Testing for Multi-Word Persistent Synchronization Primitives. In *Proc. of the 23rdh International Conference on Principles of Distributed Systems*, OPODIS'19, pages 1–17, December 2019.

[11] D. Chakrabarti, D. Chakrabartin, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'14, pages 433–452, October 2014.

[12] D. Hwang, W. Kim, Y. Won, and B. Nam. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *Proc. of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 187–200, February 2018.

[13] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. NVC-Hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proc. of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, IMDM'15, pages 4:1–4:8, August 2015.

[14] F. Nawab, D. Chakrabarti, T. Kelly, and C. B. Morrey. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. In *Proc. of the 18th International Conference on Extending Database Technology*, EDBT'15, pages 689–694, December 2015.

[15] F. Xia, D. Jiang, J. Xiong, and N. Sun. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proc. of the USENIX Annual Technical Conference*, ATC'17, pages 349–362, July 2017.

[16] PostgreSQL Global Development Group. PostgreSQL, 2020. https://www.postgresql.org/.

[17] H. J. Boehm and D. Chakrabarti. Persistence programming models for non-volatile memory. In *Proc. of the ACM SIGPLAN International Symposium on Memory Management*, ISMM'16, pages 55–67, June 2016.

[18] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proc. of the International Conference on Management of Data*, SIGMOD'17, pages 21–35, May 2017.

[19] H. S. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, K. E. Goodson. Phase Change Memory. *Proc. of the IEEE Electron Device Letter*, 98(12):2201–2227, March 2010.

[20] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proc. of the International Conference on Management of Data*, SIGMOD'16, pages 371–386, June 2016.

[21] Intel. Persistent memory development kit, libpmem library, 2018 [accessed 2020-04-20]. https://pmem.io/pmdk/manpages/linux/master/libpmem/libpmem.7.html.

[22] Intel. Persistent memory development kit, libpmemlog library, 2018 [accessed 2020-04-20]. https://pmem.io/pmdk/manpages/linux/master/libpmemlog/libpmemlog.7.html.

[23] Intel. Persistent memory development kit, libpmemobj library, 2018 [accessed 2020-04-20]. https://pmem.io/pmdk/manpages/linux/master/libpmemobj/libpmemobj.7.html.

[24] Intel. Persistent memory development kit, libvmmalloc library, 2018 [accessed 2020-04-20]. https://pmem.io/vmem/manpages/linux/master/libvmmalloc/libvmmalloc.7.html.

[25] J. Arulraj, J. Levandoski, U. F. Minhas, and P. Larson. BzTree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. of the VLDB Endowment*, 11(5):553–565, January 2018.

[26] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, pages 105–118, March 2011.

[27] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Proc. of the 30th International Symposium on Distributed Computing*, DISC'16, pages 313–327, September 2016.

[28] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv e-prints*, page arXiv:1903.05714, March 2019.

[29] J. Alakuijala J. Wassenberg. Fast strong hash functions: Siphash/highwayhash, 2018 [accessed 2020-04-20]. https://github.com/google/highwayhash/.

[30] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proc. of the 15th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181, February 2015.

[31] K. Bhandari, D. R. Chakrabarti, and H. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proc. of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'16, pages 677–694, October 2016.

[32] K. J. Norris, J. Zhang, S. Musunuru, M. Zhang, K. Samuels, J. J. Yang, and N. P. Kobayashi. TEM and EELS Study on TaOx-based Nanoscale Resistive Switching Devices. *Proc. of the Materials Research Society*, 1805, January 2015.

[33] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *Proc. of the International Conference on Management of Data*, SIGMOD'16, pages 1539–1551, June 2016.

[34] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm. Evaluating persistent memory range indexes. *Proc. of the VLDB Endowment*, 13(4):574–587, December 2018.

[35] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. In *Proc. of the 14th USENIX Conference on File and Storage Technologies*, FAST'16, pages 133–148, February 2016.

[36] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *Proc. of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'18, pages 28–43, April 2018.

[37] M. Goyal, B. Fan, X. Li, D. G. Andersen, and M. Kaminsky. libcuckoo a high-performance, concurrent hash table, 2014 [accessed 2020-06-08]. https://github.com/efficient/libcuckoo.

[38] M. K. Aguilera, S. Frølund. Strict linearizability and the power of aborting. November 2003. Technical Report https://www.hpl.hp.com/techreports/2003/HPL-2003-241.pdf.

[39] M. Nam, H. Cha, Y. Choi, S. H. Noh, B. Nam. Write-optimized dynamic hashing for persistent memory. In *Proc. of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, pages 31–44, February 2019.

[40] M. P. Herlihy, J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[41] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *Proc. of the IEEE 30th International Conference on Data Engineering*, ICDE'14, pages 604–615, March 2014.

[42] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, and K. Lim. Meet the Walkers: Accelerating index traversals for in-memory databases. In *Proc. of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO'13, pages 468–479, December 2013.

[43] Oracle. MySQL, 2020. https://www.mysql.com/.

[44] P. Zuo and Y. Hua. A write-friendly hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 29(5):985–998, May 2018.

[45] P. Zuo, Y. Hua, and J. Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proc. of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'19, pages 461–476, October 2018.

[46] R. Berryhill, W. Golab, and M. Tripunitara. Robust shared objects for non-volatile main memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems*, OPODIS'15, pages 1–17, December 2015.

[47] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[48] R. Guerraoui, R. R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Proc. of the 24th International Conference on Distributed Computing Systems*, ICDCS'04, pages 400–407, March 2004.

[49] Redis Labs. Redis, 2020. https://redis.io.

[50] S. Chen and Q. Jin. Persistent B+-trees in non-volatile main memory. *Proc. of the VLDB Endowment*, 8(7):786–797, February 2015.

[51] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *Proc. of the 5th Biennial Conference on Innovative Data Systems Research*, CIDR, pages 21–31, April 2011.

[52] S. George, K. Ma, A. Aziz, X. Li, A. Khan, S. Salahuddin, M. Chang, S. Datta, J. Sampson, S. Gupta, and V. Narayanan. Nonvolatile memory design based on ferroelectric fets. In *Proc. of the 53nd ACM/EDAC/IEEE Design Automation Conference*, DAC'16, pages 1–6, 2016.

[53] S. Ghemawat and J. Dean. LevelDB, 2020. https://github.com/google/leveldb.

[54] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram. RECIPE: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proc. of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 462–477, October 2019.

[55] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *Proc. of the 15th USENIX Conference on File and Storage Technologies*, FAST'17, pages 257–270, February 2017.

[56] S. Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Apress Media LLC, January 2020.

[57] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of the 11th USENIX Conference on File and Storage Technologies*, FAST'11, pages 65–75, February 2011.

[58] Storage Networking Industry Association. NVDIMM messaging and FAQ, 2014 [accessed 2020-04-20]. https://www.snia.org/sites/default/files/NVDIMM%20Messaging%20and%20FAQ%20Jan%2020143.pdf.

[59] T. Chen, D. Chisnall. Pointer tagging for memory safety, 2019 [accessed 2020-04-20]. https://www.microsoft.com/en-us/research/uploads/prod/2019/07/Pointer-Tagging-for-Memory-Safety.pdf.

[60] T. David, A. Dragojević, R. Guerraoui, and I. Zablotchi. Log-free concurrent data structures. In *Proc. of the USENIX Annual Technical Conference*, ATC'19, pages 373–386, July 2018.

[61] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized Concurrency: The secret to scaling concurrent search data structures. In *Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 631–664, March 2015.

[62] T. David, R. Guerraoui, T. Che, and V. Trigonakis. Designing ascy-compliant concurrent search data structures. 2014. Technical Report https://infoscience.epfl.ch/record/203822.

[63] T. Maier, P. Sanders, and R. Dementiev. Concurrent Hash Tables: Fast and General. *Proc. of the ACM Transactions on Parallel Computing*, 5(4), March 2019.

[64] T. Wang, J. Levandoski, and P. Larson. Easy lock-free indexing in non-volatile memory. In *Proc. of the IEEE 34th International Conference on Data Engineering*, ICDE, pages 461–472, April 2018.

[65] W. Cai, H. Wen, H. A. Beadle, M. Hedayati, and M. L. Scott. Understanding and optimizing persistent memory allocation. In *Proc. of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'20, pages 421–422, February 2020.

[66] W. Hu, G. Li, J. Ni, D. Sun, and K. L. Tan. $\mathsf{B}^p$ - Tree : A predictive $\mathsf{B}^+$ - Tree for reducing writes on phase change memory. *IEEE Transactions on Parallel and Distributed Systems*, 26(10):2368–2381, January 2014.

[67] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proc. of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 465–477, October 2014.

[68] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proc. of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 1077–1091, March 2020.

[69] Z. Ma, E. H.-M. Sha, Q. Zhuge, R. Zhang, and S. Gu. Towards the design of efficient hash-based indexing scheme for growing databases on non-volatile memory. *Future Generation Computer Systems*, 105:1–12, April 2020.