הטכניון
מכון טכנולוגי לישראל
TECHNION
Israel Institute of Technology

ספריות הטכניון
The Technion Libraries

בית הספר ללימודי מוסמכים ע"ש ארווין וג'ואן ג'ייקובס
Irwin and Joan Jacobs Graduate School

Concurrent Data Structures for Non-Volatile Memory

Michal-Evgenia Korenberg (Friedman)

Concurrent Data Structures for Non-Volatile Memory

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Michal-Evgenia Korenberg (Friedman)

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tishri 5782 Haifa September 2021

This research was carried out under the supervision of Prof. Erez Petrank, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period, the most up-to-date versions of which being:

Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, volume 53, pages 28–40. ACM, 2018.

Naama Ben-David, Guy Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory,. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.

Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy Blelloch, and Erez Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: Making lock-free data structures persistent. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1218—1232, 2021.

Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy Blelloch, and Erez Petrank. Flit: A library for simple and efficient persistent algorithms, 2021.

Acknowledgements

I would like to express my deepest appreciation and gratitude to my amazing advisor, Erez Petrank, for his guidance and support. I've had a huge privilege of working with an incredible researcher and adviser who taught me a huge set of professional and personal skills during my PhD. I would like to thank Erez for his constant belief in me and for being much more than an advisor to me.

I had the pleasure to work with my co-authors from which I have learned a great deal, had a lot of fun, and inspired me a lot: Naama Ben-David, Guy Blelloch, Nachshon Cohen, Maurice Herlihy, Jim Larus, Virendra Marathe, Erez Petrank, Pedro Ramalhete, Gali Sheffi, Yuanhao Wei, and Yoav Zuriel.

I thank my candidacy exam committee, Yehuda Afek, Roy Friedman, Yoram Moses, and Assaf Schuster for their time, wisdom, and advice. I also thank my final exam committee, Yehuda Afek and Roy Friedman, for their insightful comments.

I would also like to thank David Bacon who hosted me for an internship at Google New York in the summer of 2017. This internship, although not directly related to the main line of my thesis, had contributed a lot to my skill set.

The Computer Science Department was my second home and I am thankful to its administrative and academic staff for that. I thank my incredible friends that I've met along the way. I am blessed to have you in my life.

I am thankful to my family and in particular my mother for her encouragement to excel and her endless belief in me. And a last and unique thank you to my one and only, my life partner, my husband Amit. I could not have done this without you. You provide me with motivation, advice, endless love and support, which helps put everything into perspective. I dedicate this work to you.

The Technion's and Azrieli's Foundation funding of this research is hereby acknowledged.

Contents

List of Figures

Abstract

Memory and storage are considered major performance bottleneck of almost all computer systems. The role of computer storage, particularly secondary memory such as SSDs or HDDs, has become even more critical with the advance in technology and system design. Fortunately, the emergence of non-volatile main memory (NVMM), which provides persistence, byte-addressability, near-DRAM performance and large capacity, revolutionizes the way we use secondary memory. By using NVMM, data can be read-/written directly from memory, without the need of marshaling it into a block-based format, and is crash-resilient at a minimal performance cost. Several challenges still remain, however, since caches and registers remain volatile. Consequently, the most updated application's data may be located in the cache, and is lost upon a crash. Accordingly, data structures must be designed in a way that a recovery procedure will be able to retrieve a consistent memory state following a crash. Concurrent data structures increase the difficulty of the design, as achieving correctness, efficiency, and scalability simultaneously becomes more challenging, and the involvement of experts in concurrency and durability is required.

This dissertation aims at designing efficient, durable and concurrent data structures and algorithms for NVMM. It first presents novel implementations of a concurrent lock-free queue. These implementations illustrate algorithmic challenges in building persistent lock-free data structures with different levels of durability guarantees. Additionally, we present a new notion of *detectable execution*. A data structure provides detectable execution if it is possible to tell at the end of a recovery phase whether a specific operation was executed. Interestingly, there was no such requirement prior to this work. Next, we introduce two general techniques for building lock-free data structures, *NVTraverse* and *Mirror*. Presented as NVM libraries, they reduce the programming effort required to add persistence to volatile data structures. Our evaluation shows that these constructions outperform other techniques for designing durable data structures by a large margin. NVTraverse is a general transformation that takes a lock-free data structure from a general class called a *traversal data structure* and automatically transforms it into an implementation of the data structure for the NVMM setting that is provably correct and highly efficient. Mirror is a simple, general automatic transformation that adds durability to any lock-free data structure, with a low performance overhead. Moreover, Mirror exploits the hybrid system to substantially

improve performance. The next contribution in this dissertation is *FliT*, a C++ library that facilitates writing efficient persistent code. Using the library's default mode makes any linearizable data structure durable with minimal changes to the code. FliT avoids many redundant flush instructions by using a novel algorithm to track dirty cache lines. Finally, our last contribution is the development of a general construction that takes any concurrent program and makes it persistent. The converted algorithm has a constant computational delay (preserves instruction counts on each process within a constant factor), as well as a constant recovery delay (a process can recover from a fault in a constant number of instructions).

Chapter 1

Introduction

A critical bottleneck in a program's performance is accessing the slow long-term storage unit, such as HDDs or SSDs. Intel's recent introduction of the Optane memory platform, has made a large byte-addressable memory that does not lose its content upon a crash (i.e., non-volatile) and has access speeds comparable to DRAM available. The memory can serve as a very large main memory, opening up a whole new world of opportunities in designing modern programs and systems.

Numerous recent studies have proposed various schemes to exploit the benefits of non-volatile main memory. A substantial fraction of these research projects attempts to design persistent data structures, allowing them to be manipulated by the program with no need to save their content to a secondary persistent storage, in an augmented format. Data structures form the basic blocks for any software design and facilitates better storage systems, databases, key-value stores, etc. Since traditional storage media have been block-based, all these applications persist these data structures by marshaling them into a block-based format. As a result, the in-memory representation and on-disk (SSD) representation of these data structures are quite different. Byte-addressable persistent memory can be used to create a unified persistent representation. Designing data structures and algorithms for non-volatile main memory is hard because caches and registers are still volatile, and their content, which reflects the most updated data, is lost upon a crash. Thus, there might be inconsistencies in the state of the main memory, following a crash, since it might be missing recent data writes not yet flushed from the cache into the main memory. There are, however, special instructions that flush data to the memory, in order to ensure that important data is available in the (persistent) memory after a crash. These instructions allow the programmer to make sure that certain data is written to memory, securing the state of the execution. Such flushing of data from cache to memory, nevertheless, is costly, and good algorithms use these instructions sparingly. The difficulty of the design increases in the presence of concurrent data structures as correctness, efficiency, and scalability become harder to achieve simultaneously, requiring the involvement of experts on concurrency and durability.

Interestingly, a well-studied class of programs called *lock-free algorithms* ensures that memory is always in a consistent state, even during long updates. In a nutshell, lock-freedom requires that processes be able to execute operations on the shared state regardless of a slow progress or even failure of other processes in the system. This means that even if a process is swapped-out part way through its update, other processes can continue their execution. Accordingly, lock-free algorithms are a very natural fit for use in NVMM.

Thus, the main question investigated in this thesis is: How to design efficient and durable data structures and programs for NVMM?

Researchers have tried to reduce the difficulty of designing durable data structures with an assortment of techniques that fall into three broad categories: hand-tuned constructions, general transformations and persistent transactions.

1.1 Persistent Transactions

Persistent transactions can take a sequential implementation of a data structure and provide a persistent and concurrent execution of the data structures' methods (e.g., [9, 23, 70, 10, 66, 61, 60, 46, 84]).

Several prior works proposed transactional updates to persistent memory that guarantee *failure atomicity* – a collection of persistent data updates all occur or none do across failure boundaries [20, 23, 43, 61, 68, 88]. The generality of persistent transactions usually comes with a significant performance price tag. While these approaches work, they trade off performance for consistency in the face of failures – transaction runtimes incur significant bookkeeping overheads to consistently manage transaction metadata. This involves creating persistent logs to either undo or redo partial transactions [25, 30, 67, 89]. Moreover, although some of these techniques are capable of guaranteeing non-blocking progress, they all serialize write operations, which severely limit scalability in write-heavy workloads.

1.2 Hand-Tuned Constructions

An interesting alternative strategy to transactional updates is to build libraries of high performance persistent data structures [37, 32, 41] that are heavily optimized using ad hoc techniques informed by the data structure architecture and semantics. Hand-tuned constructions can achieve different correctness guarantees while providing great performance [32, 41, 37]. Performance is the main benefit in building such data structures. On the other hand, hand-tuned constructions are very specific, and do not yield a general way of designing durable data structures. One line of our research, which is presented in Chapter 3, focused on building a library of highly optimized persistent data structures using ad hoc techniques informed by the data structure architecture and semantics. We built three different designs of a persistent lock-free queue, based

on the queue of Michael and Scott [75], and presented some *guidelines* for converting linearizable data structures into durable ones. These guidelines informally specify where flushes and fences need to be inserted in an algorithm: after initializing new nodes, before completing an operation, and to ensure dependencies of operations. These queues were the first designs of highly concurrent, nonblocking data structures, optimized for persistent memory [41] in the literature. In subsequent work, David et al. [32] implement hand-tuned durable data structures for sets. David et al. [32] achieve this by using similar guidelines while carefully understanding each data structure to find its dependencies.

1.3 General Constructions

Another line of our work presents general transformations, which is another approach in designing durable data structures. These transformations are typically applied to specific class of data structures, making them durable using an automatic transformation applicable by any programmer. As indicated in previous work [59, 41, 32, 95], a lock-free algorithm is resilient to thread failures, implying that lock-free data structures are a natural choice because they are adaptable to persistence. Izraelevitz *et al.* [59] presented a mechanized transformation that can be applied to any lock-free data structure, making it persistent. Even though Izraelevitz *et al.*'s transformation is simple, the resulting durable data structures are slow, due to every atomic load or store needing to issue flush and fence instructions. Cohen *et al.* [26] introduce a universal construction that takes any sequential data structure and makes it durably linearizable, with only one flush per operation. This construction, however, suffers from the same drawback as other universal constructions— while they are very general, it is well known that universal constructions are less practical than specialized concurrent data structure designs. Mnemosyne [88] provides a clean programming interface for using persistent memory, through *persistent regions*. It avoids the traditional design effort and program latency involved in persisting data on disk. Atlas [20] provides durability guarantees for general lock-based programs, but does not capture lock-free algorithms. Pronto [72] relies on lock-based concurrency, which presents a technique that has shown that combining DRAM with NVMM can yield high throughput. In Pronto, two replicas of the data structure exist, one in DRAM and the other in NVMM. The data structure code must be modified while following a set of particular rules.

The first construction we designed, NVTraverse [40], is presented in Chapter 4. NVTraverse has improved performance dramatically, albeit with increased complexity and with the transformation restricted to a subclass of the lock-free data structures. NVTraverse is a general transformation that takes a lock-free data structure from a general class called a *traversal data structure* (that we formally define) and automatically transforms it into an implementation of the data structure for the non-volatile memory setting that is provably durably linearizable and highly efficient. The trans-

formation hinges on the observation that many data structure operations begin with a traversal phase that does not need to be persisted, and thus we only begin persisting when the traversal reaches its destination. We take a substantial step in removing the need for expert familiarity with an algorithm to make an efficient durably linearizable version of it.

Transforming a data structure into a traversal form in NVTraverse requires some programming expertise and may sometimes harm performance. The second general construction we established, Mirror [42], in contrast, uses a very different transformation and demonstrates significant improvements on current platforms. Mirror is presented in Chapter 5. Mirror is a simple automatic transformation that is capable of converting any linearizable lock-free data structure into a persistent and lock-free durably linearizable data structure. Moreover, in the current non-volatile main memory configuration, where non-volatile memory operates side-by-side with a standard fast DRAM, Mirror exploits the hybrid system to substantially improve performance. Evaluation shows a significant performance advantage over NVTraverse on current platform, where DRAM resides next to the NVMM, but it requires a larger space overhead.

In this context, in addition to the two general constructions, we also designed a library, Flit [91], which helps programmers easily design efficient persistent code for non-volatile main memory, abstracting away details of flush instructions, and applying optimizations under the hood. The library is described in Chapter 6. Intuitively, this mechanism tries to reduce the number of flushes by indicating that a word was already flushed to avoid repeated flushing. In previous work, multiple reads of the same location could have resulted in multiple flushes in order to make sure that the read value has been read to non-volatile memory. In this work, we add a counter to signify that a word was already flushed. The implementation of FliT keeps track of ongoing stores for each variable. When a store begins, it tags the memory location it operates on by incrementing this counter. A positive counter indicates the possibility that the value in this location is volatile. To deal with concurrent updates, the counter signifies how many updates are currently being applied on this field. After completing the update, the updater decrements the counter. Loads check the counter when accessing a given memory location, and only execute a flush instruction on it if it is positive. In this way, flush instructions are only executed when needed.

Finally, we also suggest a general technique for recovering an entire program after a system failure. This work, presented in Chapter 7, goes beyond data structures, and maintains a consistent state of the full concurrent program [11]. The objective of the work was to present a general construction to make any concurrent program persistent, and show that the persistent version is guaranteed to have at most a constant factor blow-up in both steps and contention. We designed simulators that take any concurrent program and makes it persistent by adding checkpoints to the program. These checkpoints allow whole system crash resiliency by satisfying the detectability definition [41] for the entire program, which is described next.

1.4 Correctness Conditions and Detectability

Careful thought is needed to define what it should mean for a concurrent structure to be correct and durable. In the absence of durability concerns, *linearizability* [53] is perhaps the most common correctness condition for concurrent objects: each operation appears to "take effect" instantaneously at some point between its invocation and response. Linearizability is attractive because it is *compositional*: the joint execution of two (or more) linearizable data structures is itself linearizable. Various definitions were proposed to formalize durability, e.g., [2, 28, 47, 7, 13, 59, 41]. We mainly worked with the definition of *durable linearizability* by Izraelevitz *et al.* [59], but most of the constructions we proposed also satisfy other definitions such as *strict linearizability* [2], etc. In a nutshell, a concurrent data structure is said to be durably linearizable if all executions on it are linearizable once crash events are removed, meaning that the state of a data structure following a crash reflects a consistent sub-history of the operations that actually occurred. This sub-history includes all operations that completed before the crash, and may or may not include operations in progress when the crash occurred. The main tool for achieving durable linearizability for a concurrent data structure is the use of explicit instructions that force volatile cached data to be written to non-volatile main memory. While such instructions enforce correctness, they also carry a performance cost and their use should be minimized. Durable linearizability is compositional: the composition of two durably linearizable objects is itself durably linearizable. Nevertheless, durable linearizability may be expensive, requiring frequent persistence barriers. An alternative – weaker – condition is *buffered durable linearizability* [59], and presented in Chapter 2. Informally, this condition guarantees that the state of the object following a crash reflects a consistent sub-history of the operations that actually occurred, but this sub-history need not include all operations that completed before the crash. Buffered durable linearizability is potentially more efficient, because it does not require such frequent persistence fences. Unfortunately however, buffered durable linearizability is not compositional: the composition of two buffered durably linearizable data structures is not itself buffered durably linearizable. To satisfy buffered durably linearizable objects, a specific linearizable operation, *sync()*, can be issued to force a single-object persistence barrier: a call to an object's *sync()* method renders durable all that object's operations completed before the call, although operations concurrent with the call may or may not be rendered durable. The *sync()* method is provided by the data structure and can be used to synchronize a safe (manual) execution of two (or more) data structures concurrently.

In Chapter 2 we define a new (natural) notion of *detectable execution*. A data structure provides detectable execution if it is possible to tell at the end of a recovery phase whether a specific operation was executed. The work that is described in Chapter 7 satisfies detectable execution. In addition, one of our queue implementations provides durable linearization and detectable execution. If the program that uses the queue

follows a similar procedure for detecting execution, it is possible to tell how much of the execution has completed on recovery from a crash, and program recovery at higher level becomes possible. This condition can be achieved by storing extra metadata beyond what is stored in a non-persisted execution. This term was broadly adopted by subsequent papers, e.g., as [26, 11, 10, 8, 34, 58].

1.5 Contributions

To summarize, the main contributions of this thesis are:

- A new correctness condition – detectability (Chapter 2).
- Novel persistent hand-tuned lock-free queue designs for NVMM (Chapter 3).
- General guidelines for designing lock-free data structures (Chapter 3).
- An automatic transformation for any traversal data structure to make it durable with significantly fewer flushes and fences than previously known general techniques (Chapter 4).
- A library that automatically converts any lock-free data structure into a persistent and efficient one by exploiting the benefits of a hybrid memory (Chapter 5).
- A new technique for tracking dirty cache lines that is fully general (Chapter 6).
- A library, that uses the above technique to issue flush instructions, providing an easy way to design efficient persistent code (Chapter 6).
- A constant-computation and constant-recovery delay simulation of a persistent program that applies to any concurrent program (Chapter 7).
- Proofs for our constructions (Chapters 4, 5, 6, 7).
- Implementation of several data structures using our transformations and evaluations of their performance (Chapters 3, 4, 5, 6, 7).

1.6 Perspective

NVMMs are larger non-volatile main memories. Compared to SSD, they are a lot faster, they allow byte access (instead of block access) and they are of comparable size. Compared to DRAM, they are a bit slower, they are much larger, they provide durability, and they are byte-addressable like DRAM.

We present lock-free concurrent algorithms that are resilient to system crashes. In perspective, there are various algorithms in various communities that obtain such resilience even before non-volatile memory emerged. These algorithms usually write

important stuff to a secondary non-volatile memory, many times in a block-based structure which increases complexity. All these algorithms can be applied as is to NVMM to obtain faster performance as NVMM is faster than SSD. The new NVMM allows better flexibility by not requiring block-based storage. The data structures are kept as is in the non-volatile storage. Small changes in the data structure can be written to memory at a much lower cost compared to writing a full SDD block.

From a different perspective, non-persistent lock-free algorithms existed beforehand and were resilient to thread crashes, as long as the memory view remained stable. A thread can resurrect and proceed in an operation after being dormant for a long time even after many operations by other threads were executed. Lock-freedom ensures that its execution will be linearized. Durable lock-free data structures allow the same resistance to delaying or crashing threads, but adds resistance to memory crash because the view of the memory remains valid when we restart the computer.

Checkpoint-restart is another technique to obtain some resistance to crashes as once in a while a snapshot of the program state is written to persistent memory **??**. The cost is that all operations executed since the last snapshot are lost. First, we should say that using non-volatile memory would make the same algorithms a lot faster simply because NVMM is faster than previous disks or SSDs. Second, buffered durable linearizability attempts to catch this spirit for data structures, allowing snapshots specifically for data structures, thus, sometimes gaining higher performance. We provide a buffered durable queue in Chapter 3.

1.7 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 provides the required background and describes the main definitions and notation used throughout the thesis. In addition, we offer a new correctness condition, denoted *detectable execution*, not found in previous works. In Chapter 3, we provide three novel persistent lock-free queue designs for NVMM. We then present two general constructions for lock-free data structures, NVTraverse and Mirror in Chapters 4 and 5, respectively. In Chapter 6, we provide a library that helps programmers easily design efficient persistent code for non-volatile main memory. Then, in Chapter 7, we present an algorithm to restore an entire program. Finally, we conclude this dissertation in Chapter 8.

Chapter 2

Preliminaries

This thesis considers the shared memory concurrent setting. In this section, we present the model and some definitions, which will form the basis for our work.

2.1 Model

Classic Shared Memory. We consider an asynchronous shared-memory system with n processes $p_1, p_2, ..., p_n$. Processes can access shared memory using *read*, *write*, or read-modify-write (RMW) instructions, like *compare-and-swap (CAS)*, *fetch-and-add (FAA)*, and *test-and-set (TAS)*. We sometimes refer to read instructions as *loads*, and to all other instructions collectively as *stores*. Each memory location is categorized at any given point in time as *shared* or *private*. There is a *root* location in memory, that is always shared. A private memory location can only be accessed by a single process i, which can make that location shared by executing a specific store on some shared location. This store depends on the algorithm; it could be releasing some lock, or swinging a shared pointer to point to this location.

Persistent Memory. The shared memory may be divided into three parts: (1) A volatile cache, which is the fastest memory, but volatile, meaning that its content will not survive a potential crash; (2) a volatile DRAM, which is slower than the cache, but volatile as well and is larger than the cache; and (3) an NVMM, which is slower than the DRAM but faster than traditional non-volatile memories. The NVMM is durable, and we assume that only its content will remain after a crash, as opposed to cache and DRAM.

All accesses are applied to volatile memory. Values in volatile memory can be written back to persistent memory, or *persisted*, in a few different ways; a value could be persisted *implicitly* by the system, corresponding to an automatic cache eviction, or *explicitly* by a process. To make a value that is in volatile memory appear in persistent memory, processes can execute *persistence instructions*, which include *flush* and *fence* (as long as these addresses are mapped to the persistent memory). A *flush* instruction

is non blocking and triggers a write-back for a specific memory location, which is given as a parameter. The value v in memory location ℓ is said to be *flushed* if a flush instruction was executed on ℓ when v was in ℓ. A *fence* instruction orders all preceding writes and flushes executed by that process to become visible to other processes before any writes or flushes executed after the fence. Intuitively, a fence following a flush may be thought of as blocking until all previously flushed locations have reached the memory. We say that a value has been *persisted* by time t if the value reaches persistent memory by time t, regardless of whether it was done implicitly or explicitly. Note that persisting is done on *memory locations*. It is sometimes convenient, however, to discuss *modifications* to memory being persisted. We say that a modifying instruction m on location ℓ is persisted if ℓ was persisted since m was executed. A modifying instruction m is said to be *pending* if it has been executed but not persisted. The specific architecture instructions are presented in Intel, AMD and ARM manuals [57, 54, 4, 5], and described thoroughly in [80, 82].

Execution. We consider programs that execute data structure *operations*. Each operation is implemented as a sequence of instructions. An operation is *invoked* when its first instruction is executed, and its *response* occurs immediately after its last instruction is executed. An invocation of an operation *op* by process p with arguments $args$ is denoted $inv(op, p, args)$, and a response to this operation with return value r is denoted $resp(op, p, r)$. An operation's *implementation* specifies a sequence of instructions for processes to execute. Every operation's invocation and response are considered *events* which are related to the calling process.

In our system, we consider system-wide crashes as events as well, which are not associated with a specific process. A *crash event* erases all values in volatile memory, but leaves the persistent memory intact. Thus, all modifications that were pending at the time of the crash are lost, but all others remain. Furthermore, after a crash event, new processes are spawned. An execution of a concurrent program is modeled by a *history* comprising a finite sequence of invocation and response events of operations by processes, as well as crash events. Note that, as in Izraelevitz *et al.* [59], crash events partition an execution as $E = E_0 C_1 E_1 C_2 ... E_{c-1} C_c E_c$, where c is the number of crash events in E. C_i denotes the i-th crash event, and $ops(E)$ denotes the *sub-execution* containing all events other than crashes. Note that $ops(E_i) = E_i$ for all $0 <= i <= c$. Following to Izraelevitz *et al.* [59], we call the sub-execution of E_i the i-th *era* of E.

A *low-level history* is a sequence of instructions and crash events. A *high-level history* is a sequence of invocations and responses of operations, as well as crash events. Each low-level history maps to some high-level history, and each high-level history can map to a set of low-level histories, representing the different interleavings of instructions that could occur.

For an invocation *inv* in a high-level history H, a *matching response* is the next response in H after *inv* that pertains to the same operation and process. A *method call*

in a high-level history H is a pair consisting of an invocation and the next matching response. An invocation is *pending* in H if no matching response follows the invocation. *complete*(*operation*) in H is an operation that has a matching response follows the invocation. *complete*(H) is the extension of H with all matching responses to pending invocations appended in the end. We use *trunc*(H), to denote the set of histories that can be generated from H by removing some of the pending invocations.

A *subhistory* of a history H is a subsequence of the events of H. For a process p, the *process subhistory*, $H|p$ is the subsequence of events in H whose process names are p. For an object o, the *object subhistory* $H|o$ is similarly defined. Histories H and H' are *equivalent* if for every process p, $H|p = H'|p$. A process or object history is *well-formed* if every response has an earlier matching invocation.

A method call m_0 *precedes* a method call m_1 in a high-level history H if m_0 finished before m_1 started: that is, m_0's response event occurs before m_1's invocation event in H. Precedence defines a *partial order* on the method calls of H: $m_0 <_H m_1$. In other words, operation op_0 *happens before* operation op_1 in H, denoted $op_0 <_H op_1$, if op_0's response is before op_1's invocation in H. A *consistent cut* of a history H is a subhistory $G \subseteq H$ such that, if m_1 is in G, and $m_0 <_H m_1$, then m_0 is also in G.

A high-level history H is *sequential* if the first event of H is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response. If S is a sequential high-level history, then $<_S$ is a total order. A *sequential specification* for an object o is a prefix-closed set of sequential histories called the *legal* histories for o. A sequential history H is *legal* if each object subhistory $H|o$ is legal for o.

2.2 Correctness and Progress Conditions

We start by recalling the standard definition of linearizability and the extension to definitions that also capture system-wide crash events.

Linearizability. Linearizability is a correctness condition for concurrent objects, which is widely used as the basic correctness condition for many concurrent data structure algorithms.

Definition 2.2.1. [53] *Linearizability.* For history H, and partial order $<$ extending $<_H$, H is *linearizable* if there exists a *complete*(*trunc*(H)), H', and there is a legal sequential history S, with no pending invocations, such that

L1 *complete*(*trunc*(H)) is equivalent to S, and

L2 if method call $m_0 <_{H'} m_1$, then $m_0 <_S m_1$ in S.

We refer to S as a $<$-*linearization* of H.

Essentially, a history H of operations of data structure D, with no crash events, is *linearizable* if there is a single point in time during the execution of each operation at

which that operation *takes effect*, such that the sequence of these points adheres to the sequential specification of D.

Durable Linearizability. Since the linearization definition does not capture system-wide crash events, Izraelevitz *et al.* [59] introduced an extended definition of linearizability that also takes into account system-wide crashes. To satisfy *durable linearizability*, the execution model allows a recovery operation, which is called immediately after the crash, before any other operation is made. The recovery operation, however, can be run concurrently with other operations on the data structure.

Definition 2.2.2. [59] *Durable Linearizability.* An object is *durably linearizable* if, a crash and recovery that follows linearizable history H, leaves the object in a state reflecting a consistent cut H' of H such that

DL1 ops(H') is linearizable

DL2 every complete operation of H appears in H'.

In other words, a history is said to be *durably linearizable* if the removal of all crash events from the history still leaves the history linearizable [59]. In particular, this means that the effect of completed operations may not be lost, and operations that were in progress at the time of a crash must either take effect completely, or leave no effect on the data structure. Furthermore, if an operation does take effect, then all the operations it depends on must also have taken effect. Izraelevitz *et al.* [59] show that durable linearizability is compositional.

Buffered Durable Linearizability. We now move to defining *buffered durable linearizability* [59], which is a weaker condition. Let us stipulate that each object provides a *sync()* method, which allows a caller make sure that operations completed prior to the *sync()* are made persistent before operations that follow the *sync()*.

Definition 2.2.3. [59] *Buffered Durable Linearizability.* An object is *buffered durably linearizable* if, a crash and recovery that follows linearizable history H, leaves the object in a state reflecting a consistent cut H' of H such that

BDL1 H' is linearizable

BDL2 every completed *sync()* operation of H appears in H'.

Buffered durable linearizability is not compositional. For example, suppose a process dequeues value x from p, and then enqueues x on q, where p and q are distinct buffered durably linearizable FIFO queues. Following a crash, the thread might find two copies of x, one in each queue, while if the composition of the two queues were buffered durably linearizable, the recovering thread might find x in p but not q, or in q but not p, or in

neither, but never in both.

Detectable Execution. A caller of a data structure operation often needs to be able to tell whether an operation has executed even when a crash occurs. Imagine an operation that adds money to a bank account, or an operation that places an order for a car. One would like to know that such operations execute exactly once. A typical execution scenario is one where a process has a durable list of operations to execute and it executes these operations one by one. Upon recovery from a crash, the process needs to know which of its data structure updates were executed. Providing a mechanism to determine whether an operation completed during a crash is therefore beneficial.

One of our contributions is defining *detectable execution* [41]. We say that a data structure provides *detectable execution* if it provides a mechanism that, upon recovery from a crash, makes it possible to tell whether each operation executed while the crash occurred was completed or aborted. This term was broadly adopted by subsequent papers, e.g., as [26, 11, 10, 8, 34, 58].

In practice, an implementation of such a mechanism can take the following form. An announcement array (similar to [51]) will hold an entry for each thread in which the thread announces an intention to execute an operation, and provides space for a recovery mechanism to write the operation result. Upon recovery from a crash, the recovery process will set a flag in each such entry specifying whether the last intended thread's operation was completed, and deliver the result, if relevant.

Lock-freedom. A property that naturally fits crash survival is *lock-freedom*. An object is *lock-free* if there is progress always being made by at least one process, if processes are run sufficiently long. In other words, a data structure is lock-free if processes make progress in every execution, regardless of the schedule. In particular, even if some processes pause, other processes will be able to continue executing their operations on the object. Furthermore, even if some process halts during an operation, other processes will continue executing, as no process may block another. It also means that a lock-free object always keeps the memory in a consistent state. We exploit this property in our algorithms to guarantee that after a crash, a linearizable lock-free data structure that uses our constructions will be durably linearizable.

Chapter 3

A Hand-Tuned Construction: A Lock-Free Queue

3.1 Introduction

One of the strategy for building libraries of high performance persistent data structures [37, 32, 41] that are heavily optimized is to use ad hoc techniques informed by the data structure architecture and semantics. Hand-tuned constructions can achieve different correctness guarantees while providing great performance [32, 41, 37], which is the main benefit in building such data structures. On the other hand, they are very specific, and do not yield a general way of designing such data structures.

Our first contribution is three novel designs of durable concurrent queues [41]. We study the NVMM challenges by designing a durable version of the lock-free concurrent queue data structure of Michael and Scott [75], which also serves as the base algorithm for the queue in the java.util.concurrent library. It is easy to obtain a durable linearizable queue by adding many persistence barrier operations automatically, but the resulting performance can be very low. In this work, we attempt to minimize the overhead and still achieve robustness to crashes. The first implementation, denoted *durable* queue, provides durable linearization [59]. The second implementation, denoted *log* queue, provides durable linearization and detectable execution we presented in Chapter 2. The third implementation, denoted *relaxed* queue, provides buffered durable linearizability [59] with an implementation of a *sync()* operation. This work was the first to introduce durable lock-free data structure designs for non-volatile main memory, as lock-freedom is a natural fit for a crash-resilient setting. The queue is a fundamental data structure that will find uses in future applications optimized for persistent memory. Several existing messaging systems use a FIFO queue in their core and could benefit from high-performance queues for non-volatile memories. For this study, we chose to extend Michael and Scott's queue due to its portability, simplicity and performance. There exist faster queues that employ the **fetch&add** instruction [76, 93], but they are not portable to platforms that do not support this instruction (e.g., SPARC),

and are significantly more complicated.

When crashes occur during an execution, it is often difficult to tell which operations were executed and which operations failed to execute. Durable linearizability guarantees the completion of all operations that were executed before a crash but does not provide a mechanism to determine whether an operation that executed concurrently with a crash was eventually executed. A persistent queue is not itself sufficient for program recovery. One needs the larger context, and the ability, upon recovery, to determine how much has been executed so far. Without the ability to identify lost operations, it would be difficult to recover the entire program, because in practice it is often important to execute each operation exactly once.

In Chapter 2 we enable a more robust use of the queue by defining a new (natural) notion of *detectable execution*. A data structure provides detectable execution if it is possible to tell at the end of a recovery phase whether a specific operation was executed. The *log* queue provides durable linearization and detectable execution. If the program that uses the queue follows a similar procedure for detecting execution, it is possible to tell how much of the execution has completed on recovery from a crash, and program recovery at higher level becomes possible.

In this context, we proposed several guidelines for designing such lock-free data structures that will be resilient to system failure, and also implemented the three queue designs and measured their performance. As expected, implementations that provide durable linearization have a noticeable cost. Implementations that provide only buffered durable linearizability, however, demonstrate good performance when the *sync()* method is invoked infrequently.

The rest of this chapter is organized as follows. Section 3.2 reviews previous work and hardware instructions on which our work is based. Section 3.3 provides an overview of the three queue versions. We provide the details of the durable, log and relaxed queues in Sections 3.4, 3.5, and 3.6, respectively. A mechanism for memory management is presented in Section 3.7. The experimental evaluation for all three queues is presented in Section 3.8. Section 3.9 discusses related work and Section 3.10 concludes.

3.2 Preliminaries

3.2.1 The MS Queue

Our constructions extend Michael and Scott's queue [75] (denoted the MS queue). As explained in the introduction, we chose this queue because it is highly portable (it only uses the CAS instruction), it is used in the Java concurrency library, and it is adequately complex for a study of durable data structures.

The MS queue is built on an underlying linked-list in which references to the head and tail are held, called, respectively `head` and `tail`. The head points to a dummy node that is not considered part of the queue, and is only there to allow easy handling

of the empty list. The list is initiated to a single (dummy) node referenced by both the head and the tail.

To dequeue an element, the dequeuing thread tries to move the `head` to point to `head->next` using an atomic CAS instruction. Upon success, it retrieves the value in the new node pointed to by the head and upon failure it begins from scratch.

To enqueue an element, a new node is allocated and initialized with the required value and a null next pointer. If `tail->next` is NULL, then the enqueue attempts to let `tail->next` point to its node using an atomic CAS instruction. Upon success, it then moves `tail` to point to `tail->next`. Upon failure to append the node, the operation goes back to inspecting the tail and attempting to append at the end. If `tail->next` is not NULL, this means that the previous operation has not completed and the tail must be fixed to point to the last node. Thus, the current thread attempts to fix the tail and then it starts from scratch.

The linearization point of a dequeue operation is at a successful CAS executed on the head. Enqueuing an element is linearized in a successful CAS appending a node at the end of the queue. Note that the tail can later be fixed by the thread performing the enqueue or by any other enqueue operation that needs to append its node at the end. No appending is attempted before the tail is fixed and pointing to the last node in the queue. A full description of this queue appears in the original paper [75].

3.2.2 Hardware Instructions for Persistence

In the algorithms presented in this work, we use a FLUSH instruction that receives a memory address and flushes the content of this address (together with its entire cache line) to the memory, making it persistent. On an Intel platform this translates to two instructions: CLFLUSH, SFENCE. It has recently been shown that CLFLUSH has store semantics as far as memory consistency is concerned [54], which guarantees that no previous stores will be executed after the CLFLUSH execution. The SFENCE instruction guarantees that the CLFLUSH instruction is globally visible before any store instruction that follows the SFENCE instruction in program order becomes globally visible.

3.3 An Overview of the Three Queue Designs

Our queue builds on Michael and Scott's queue (denoted the *MS queue*) and consists of a linked list of nodes that hold the enqueued values, plus the `head` and the `tail` references. The basic original queue is extended with FLUSH operations to persist memory content required for recovery from crashes, and also with additional information that facilitates recovery.

Our three designs offer varying levels of durable linearization with guarantees provided to the caller. The *durable* version provides durable linearizability, the *log* queue provides both durable linearization and detectable execution, and the *relaxed* version

provides buffered durable linearizability.

3.3.1 The Durable Queue

The durable queue satisfies durable linearizability [59], implying that any operation that completes before a crash must become persistent after the recovery. The first guideline we use in constructing this queue is the *completion guideline*, which states that when an operation completes, its effect is durable. This ensures that when a crash occurs, previously completed operations are bound to persist. In addition, a dependence order must be maintained in this construction between all operations that occur concurrently to a crash. For example, if two dequeues occur concurrently with a crash, linearizability dictates that if the second dequeue completes (the one that dequeued the later value in the queue), then the earlier dequeue must complete as well. Ensuring this requires extra care. Therefore, the second guideline we use is the *dependence guideline* by which each operation must ensure that all *previous* operations become durable before starting to execute. *Previous* here refers to operations that the current operation depends on and that must be linearized before the current operation can be linearized. We recommend this guideline be followed for all future constructions. Finally, we use a third and generally recommended *initialization guideline*, by which all fields of an object are flushed after the object is initialized and before the object is added to (i.e., before it becomes shared) the data structure. An overview on how the above three guidelines are implemented for the queue follows.

The `enqueue` operation of the durable queue starts by allocating a node and initializing it with the enqueued value and with a NULL next pointer. Next, it FLUSHes the node content to memory. This ensures that, before this node is appended to the queue, its durable content becomes updated. Next, recall that the original MS enqueuer attempts to append the node to the end of the queue and then fix the tail to point to the appended node. Appending is only allowed if the tail points to the last node, whose next pointer is NULL. If this is not the case, the enqueuer first fixes the tail and only then tries again to append its own node.

In the extended enqueue of the durable queue, we add a FLUSH instruction after appending the new node and before fixing the tail. This FLUSH persists the pointer from the previous last node to the newly appended node. This flush satisfies the first guideline: at this point the operation is durable. If a crash occurs, the new node is safely persistent in the list.

Next, we turn to satisfying the second guideline. We need to make sure that a previous enqueue is made persistent before a new enqueue operation starts. That should be done when one thread adds a node at the end but pauses before flushing the pointer to the added node (and also before fixing the tail). In this case, extra care should be taken when helping to fix the tail for another operation. If an enqueuer needs to fix the tail following an incomplete previous enqueue operation, then it also flushes

this pointer (that links the added node to the queue) before fixing the tail to point to this node.

The flush operations described above ensure that after enqueuing a node, the node content is durable and the pointer leading to it from the linked list is durable. Namely, all the backbone pointers of the linked-list underlying the queue are durable, except possibly the last updated pointer, whose enqueuing operation has not yet completed. The `tail`, on the other hand, need not be durable. During a recovery we can find its value by chasing the linked-list from the current location of the `head` until the last reachable node.

To make the `dequeue` operation durably linearizable, we need to add more than just flushes. First, we add a node field `deqThreadID`. This field in the queue node points to the thread that dequeued (the value in) this node. The `deqThreadID` field serves two purposes. First, it provides a direction for the recovery procedure to place the dequeued value at the disposal of the adequate dequeuer if a crash occurs. To facilitate such a recovery, we keep a `returnValue[]` array with an entry for each thread, in which a returned value can be placed. Second, it allows one dequeue operation to ensure that a previous dequeue operation completes and is made persistent.

To dequeue a value of a node, a dequeuer attempts to write its thread ID in the `deqThreadID` field of node `head->next` using an atomic CAS instruction. Whether successful or not, it then flushes the `deqThreadID` field to the memory to make sure that the thread which succeeded in this dequeue is recorded in the NVMM. Next, it places the node's value in `returnedValues[deqThreadID]`, flushes this field to the memory to make sure the result is durably delivered to the caller, and updates the head to point to the next node. If the dequeuer did not manage to write its own thread ID into `deqThreadID`, then (after helping) it starts again by trying once more to place its thread ID in the `deqThreadID` field of `head->next`.

These operations provide durability as required. When an operation completes, its effects are persistent, and before an operation starts, it makes the effects of the previous operation persistent. However, this design has performance costs due to the added FLUSHes. Measurements of this cost are given in Section 3.8.

To recover from a crash, we fix the head, making sure that all dequeued values are placed in their intended locations, and place the head over the last node whose `deqThreadID` is non-NULL. We then also fix the tail to point to the last reachable node in durable memory.

The full algorithmic details appear in Section 3.4.

3.3.2 The Log Queue

The second implementation provides durable linearization and also detectable execution. This means that following a recovery after a crash, each thread can tell whether its operation has been executed, and it receives the results of completed operations.

The *log* queue implementation employs a log array for the threads. An operation starts by being announced on a thread log. An operation is assign an operation number that is given by the user invoking thread such that the thread ID and the operation number uniquely identify the operation. The log contains the operation number, a flag that signifies if the operation completed, and an additional field that holds the operation result. If a crash occurs, then it is possible to simply inspect the log entry that contains the relevant operation number after the recovery to determine whether an operation of a crashed thread was executed or needs to be started again. Thus, the program can execute each of its intended operations exactly once.

Our general methodology for combining durability with detectable execution is to start with the durable version of the data structure and extend it with a mechanism to notify that the operation has completed. We demonstrate this approach on the queue. In the log array we maintain, for each thread, a log object on which a thread announces its intent to execute an operation, and on which the result and the operation numbers are written. The algorithmic details appear in Section 3.5. We provide an overview next.

The *log* queue employs two additional fields in the queue nodes. The first field is named `logInsert` and it specifies which thread enqueues this node. More specifically, this field points to the log object that represents the intent to enqueue this node on the queue. The second field added to the node is called `logRemove` and it points to the log object of the dequeuer that dequeues this node.

The `enqueue` operation of the log queue starts by allocating a log object and a new queue node. Both the log object and the queue node are first initialized. The node's `value` is determined by the input, the node's `next` field is set to NULL, and the node `logInsert` field points to the log object. The log object is initialized with a pointer to the new node, with an indication that the operation is enqueued, and with an operation number that is assigned by the invoking thread. The contents of the node and the log object are then flushed to memory. Next, a pointer to this log object is placed in the log array (at the entry of the enqueuer thread) and this array entry is also flushed. Next we try to append the new node at the end of the queue and, if successful, we flush the appending pointer, namely, from the previous last node to the current last node, which has just been appended. Finally, we update the tail. No flushing is required for tail updates.

If the tail is not pointing to the last node, we need to fix the tail. Before fixing the tail, we flush the last pointer and fix the tail. After fixing the tail it is possible to try again to append our node at the end of the queue.

To `dequeue` a node we start by allocating a log object and initialize it to indicate the dequeue operation and the operation number. The log object is then flushed, a pointer to it is placed in the log array, and this entry in the log array is flushed as well. We then try to write a pointer to the log object into the `logRemove` entry of node `head->next`. Upon success, we flush the content of the `logRemove` field. We then put a

pointer in the log to this node (which indicates that the operation has completed) and flush the log content as well. Finally, we advance the head to `head->next`. (The `head` need not be flushed.) A thread that fails to write its log entry into node `head->next` helps complete the dequeue operation (including flushing, linking to the log, flushing, and updating the head) and then tries its operation again.

During recovery, we start from the head and walk the linked list. Whenever we see a pointer to a log, we check whether the operation is completed, and if not, we complete the operation. The head is set to the last node that has a non-NULL `logRemove` field. We then proceed to update the tail to the last element in the list. We also make sure the last enqueue is completed by following the above procedure for marking the completion of the enqueue operation in the relevant log before fixing the tail for the last time. Finally, we go over all log entries and complete all the unfinished operations.

The proposed algorithm inherits from the durable queue the completion, dependence, and initialization guidelines for all of the operations included there. We use the initialization guideline for initializing the log object, while the dependence guideline ensures that previous operations become durable before we execute our own. In addition, we use a *logging guideline*, which ensures that the log with the description of the intended operation and the operation number is flushed before the operation is executed. This in turn ensures that, upon recovery, the operation will be completed.

3.3.3 The Relaxed Queue

The *relaxed queue* implementation provides buffered durable linearization, which is a weaker requirement. Buffered durable linearization only mandates that, upon failure, a proper prefix of the linearized operations take effect after recovery, while the rest of the operations are lost. There is no need to recover all operations that completed before the crash and thus no need to make an operation durable before returning. Hence, we adopt different guidelines, to maximize performance. The implementation needs to provide a `sync()` method that forces previous operations to become durable before later operations become durable. Typically, a caller invokes the `sync()` method to ensure proper compositionality between different data structures; occasionally, it does so to make sure not too many operations are lost when a crash occurs.

We use a design pattern that can be used for other data structures as well. During the execution of a `sync()` operation, we obviously make all previously executed operations durable, but we also save the state of the queue. In case of a crash, we (boldly) discard all operations that followed the last `sync()`, by returning to the saved state from the latest `sync()`. This may seem a painful loss of operations during a crash, but it efficiently satisfies the (weak) requirement of buffer durability and it allows the queue to be saved at different frequencies. The algorithm can completely avoid executing any FLUSH instructions inside the `enqueue` and `dequeue` operations. We only execute FLUSHes in the `sync()` method. This implies low overhead if crashes and `sync`

`()` invocations are infrequent. Buffered durable linearizability is guaranteed because a consistent cut (a proper prefix) of the executed operations is always recovered after a failure. We call this design pattern *return-to-sync*.

To apply this idea to saving a state of the queue, we first note that nodes in the queue are essentially immutable from the moment they are appended to it. So if we look at a current queue state and would like to elide several recent operations and return in time to an earlier state, it suffices to simply restore `head` and `tail` to their previous values at that earlier time (and set `tail->next` to NULL). Keeping this in mind, we add to the queue state two variables, `saved_head` and `saved_tail`, which hold the values of `head` and `tail` the last time `sync()` was called. Whenever a crash occurs, we can set `head` and `tail` back to their saved values. For this to work properly, we need to make all the nodes between `saved_head` and `saved_tail` persistent. The `sync()` method ensures this by performing the required flushes.

The above motivating discussion implies what the `sync()` method should do. This method starts by reading the current `head` and `tail` values. It then flushes the content of all nodes in between these two pointers to the durable memory, and finally, it attempts to replace the previously saved `head` and `tail` with the current ones. The first challenge is to obtain an atomic view of `head` and `tail`, in order to make sure that a consistent cut (i.e., a proper prefix) of the operations is made persistent. A second challenge is to replace the values of `saved_head` and `saved_tail` simultaneously. The third challenge is to coordinate multiple `sync()` operations and make sure that the most updated consistent cut is saved to NVMM.

We solve the first challenge by marking the `tail` pointer, after which it does not change until the `head` and `tail` are saved. It is important that we not mark the `tail` in the middle of an enqueue operation. This can be enforced by helping to complete previous operations. The simultaneity challenge is simply solved by holding `saved_head` and `saved_tail` inside an object that is replaced by a single CAS instruction. The third challenge is solved by obtaining a global number that indicates the order of the `sync()` operations and dealing with races that come up. The full algorithmic details appear in Section 3.6. For the relaxed queue, the completion guideline is irrelevant. The return-to-sync design pattern makes the dependence and the initialization guidelines irrelevant as well.

3.4 Algorithm Details of the Durable Queue

As mentioned above, our queue extends the MS queue. Its underlying data structure includes a linked-list of queue nodes and the `head` and `tail` pointers. The first node in the linked-list is a sentinel node that allows simple treatment of an empty list. The implementations use FLUSH in order to maintain different levels of guarantees. The FLUSH operation consists of two hardware instructions, as discussed in Section 3.2.2. In this section, we provide the details of the `durable` queue.

Figure 3.1: Internal Durable Queue classes

```
1  class Node {
2    T value;
3    Node* next;
4    int deqThreadID;
5
6    Node(T val): value(val), next(NULL), deqThreadID(-1) {}
7  };
8
9  class DurableQueue {
10   Node*  head;
11   Node* tail;
12   T* returnedValues[MAX_THREADS];
13
14   DurableQueue() {
15     T* node = new Node(T());
16     FLUSH(node);
17     head = node;
18     FLUSH(&head);
19     tail = node;
20     FLUSH(&tail);
21     returnedValues[i] = NULL; // for every thread
22     FLUSH(&returnedValues[i]);
23   }
24 };
```

Our queue's underlying representation is a singly-linked list with a sentinel node. It builds on the inner `Node` class, which holds elements of the queue's linked-list. In addition to the standard node fields, i.e., the value and the pointer to the next element, the node class also contains an additional field (line 4): *deqThreadID*. This field holds the ID of the thread that removes the node from the queue.

The durable queue class contains two pointers and an array. The pointers *head* and *tail* point to the first and last nodes of the linked list that implements the queue. The *returnedValues* array is an array of pointers to objects that hold dequeued values (line 12). This array contains an entry for each thread and its size is MAX-THREADS, which is the number of threads that might perform operations on the queue.

The *returnedValues* array entries point to an object that contains a single `value` field. This field either contains a value that has been dequeued from the queue, or one of three special values that are not valid queue values:

1. The special NULL value signifies that the thread is currently idle (this is the initial value).

2. The special *pending* value indicates the intention of the thread to remove a node.

3. The special *empty* value is returned when the queue is empty.

The queue constructor initializes the underlying linked list with one sentinel node. It lets the head and the tail point to this sentinel node, and it also initializes the returned values array with the special NULL value. In order to persist these values, we flush the sentinel node, the head and the tail pointers, and the *returnedValues* array.

Figure 3.2: The enqueue operation of Durable Queue

```
1  void enq(T value) {
2    Node* node = new Node(value);
3    FLUSH(node);
4    while (true) {
5      Node* last = tail;
6      Node* next = last->next;
7      if (last == tail) {
8        if (next == NULL) {
9          if (CAS(&last->next, next, node)) {
10           FLUSH(&last->next);
11           CAS(&tail, last, node);
12           return;
13         }
14       } else {
15         FLUSH(&last->next);
16         CAS(&tail, last, next);
17       }
18     }
19   }
20 }
```

3.4.1 The Enqueue() Operation

The pseudo-code for the enqueue operation is provided in Figure 3.2. The *enqueue* method receives the value to be enqueued and it starts by creating a new node with the received value (line 2). It then flushes the node content (line 3). Next, the thread checks whether the tail refers to the last node in the linked list (lines 7-8). If so, it tries to append the new node after the last node of the list (line 9). Insertion consists of two actions: adding the new node after the last node and updating the tail to reference the newly added node. To ensure proper durability, we add a flush between the two actions. If insertion at the end is successful, we flush the next pointer of the previous node (line 10), and only then update the tail (line 11). Failure in updating the tail means that another thread has helped update it already completed the insertion of the current thread. Failure in the first CAS instruction (line 9) means that another thread has appended a different node and the operation starts from scratch (line 4). In case the next pointer of the last node does not point to NULL (line 14), we help complete the

previous enqueue operation by flushing the previous next pointer (line 15) and fixing the tail (line 16).

The consistent flushing of the next pointer before updating the tail and the flushing of the node content after its initialization yield an important invariant: the entire linked list, up until the current tail, is guaranteed to reside in the volatile memory. This enables the correct execution of the recovery procedure.

3.4.2 The Dequeue() Operation

The dequeue operation receives a thread ID of the dequeuer. It starts by creating and initializing to NULL a new T object, whose purpose is to hold the dequeued value (line 2). Next, it flushes T's content (line 3). Then, it puts a reference to T in the `returnedValues` array and flushes the array entry (lines 4-5). Next, the thread checks that the queue is not empty and that the tail points to the last node (lines 7-18).

If the queue is empty, the method updates the corresponding entry in the `returnedValues` array with the empty value, flushes it and returns (lines 13-15). If the head and tail refer to the same node and the tail must be fixed, i.e., there is some enqueue operation in progress (line 11-12), then the dequeuer helps complete this enqueuing operation. It flushes the next pointer of the previous node (in our case the sentinel), fixes the tail and returns to the beginning of the while loop (lines 17-18). Dequeuing a node consists of following actions: marking the `deqThreadID` field of the node `head->next` with the dequeuer thread ID, writing the dequeued value in the `returnedValue` array, and promoting the head. If the thread succeeds in changing the `deqThreadID` field from NULL to his thread ID (line 21), then it flushes `deqThreadID` (line 22).

Next, it updates its entry in the array with the new value and flushes this result. Finally, it updates the head (line 25). Failure to update the head means that another thread has helped and completed the removal of the current thread. Failure to update the `deqThreadID` field means that another thread has already marked the node with its own threadID, so the dequeuer helps complete this other dequeue before starting again (lines 27-34).

An important invariant here is that before the head is moved, the `deqThreadID` field content (which determines which thread receives the dequeued value) is made durable so that recovery can identify the winning dequeue operation. Also, before the head is advanced, the dequeued value is written to the `returnedValue` array and the returned value is flushed. Therefore, once the head advances in main memory, we know that the dequeuing of all previous nodes can be recovered.

3.4.3 The Recovery() Operation

After a crash, we assume that new thread start executing when the system recovers. There new threads execute the recovery method before starting to execute data structure operations. As a situation where some threads may finish the recovery operation

Figure 3.3: The dequeue operation of Durable Queue

```
1  void deq(int threadID) {
2    T* newReturnedValue = new T();
3    FLUSH(newReturnedValue);
4    returnedValues[threadID] = newReturnedValue;
5    FLUSH(&returnedValues[threadID]);
6    while (true) {
7      Node* first = head;
8      Node* last = tail;
9      Node*next = first->next;
10     if (first == head) {
11       if (first == last) {
12         if (next == NULL) {
13           *returnedValues[threadID] = EMPTY;
14           FLUSH(returnedValues[threadID]);
15           return;
16         }
17         FLUSH(&last->next);
18         CAS(&tail, last, next);
19       } else {
20         T value = next->value;
21         if (CAS(&next->deqThreadID, -1, threadID)) {
22           FLUSH(&first->next->deqThreadID);
23           *returnedValues[threadID] = value;
24           FLUSH(returnedValues[threadID]);
25           CAS(&head, first, next);
26           return;
27         } else {
28           T* address =
29           returnedValues[next->deqThreadID];
30           if (head == first) { // same context
31             FLUSH(&first->next->deqThreadID);
32             *address = value;
33             FLUSH(address);
34             CAS(&head, first, next);
35           }
36         }
37       }
38     }
39   }
40 }
```

before others, it was designed in a way that it can run concurrently to threads that execute data structure operations as well.

We divide the recovery procedure into four logical parts: Let A be the last node that has a non-NULL `deqThreadID` field, and let B be the last node that is reachable from the `head` and whose next pointer is NULL.

1. If A's predecessor is reachable from the `head`, then update the `head` by a CAS operation to point to the predecessor of A. Otherwise, skip to the third step.

2. Flush A's `deqThreadID` field, update the `returnedValues` array to point to the node's value , flush the array entry and update by a CAS operation the `head` to point to A (This becomes the new sentinel node).

3. If B's predecessor is reachable from the `head`, then update the `tail` by a CAS operation to point to the predecessor of B. Otherwise, update the `tail` to point B and skip the fourth step.

4. Flush the `next` pointer of the node the `tail` points to, and then update the `tail` by a CAS operation to point to B.

Note that the `head` and the `tail` are not flushed during any run explicitly, so the recovery operation might go through nodes that was already removed and have a `deqThreadID` field which is not -1. In order to complete operations that were partially executed before the crash, we need to update the `head`, the `tail` and the last node that has a non-NULL `deqThreadID` field may have the array entry not updated. According to that, and the fact that threads that complete the recovery phase may tun concurrently to threads that are still in the recovery phase, we only promote the `head` and the `tail` to one node before the node they suppose to point to. Then, we do the operations that we would usually do during the normal run, which means the normal help operations, that are similar to the second and the fourth step in the recovery. It makes the procedure generic, that does not really has to differentiate between the nodes that have been changed during the previous and the current run.

3.4.4 Correctness

Theorem 3.1. *The durable queue is durably linearizable.*

A formal verification tool has been published by Derrick et al. [34]. Derrick et al. [34] uses this tool in order to formally prove that the Durable queue, which is presented in the work, is durably linearizable [59].

3.5 Algorithm Details of the Log Queue

This section provides details of the *log* queue, which extends the durable queue and provides both durable linearization and detectable execution.

Figure 3.4: Internal Log Queue classes

```
1  class Node {
2    T value;
3    Node* next;
4    LogEntry* logInsert;
5    LogEntry* logRemove;
6
7    Node(T val): value(val), next(NULL), logInsert(NULL),
       logRemove(NULL) {}
8  };
9
10 class LogEntry {
11   int operationNum;
12   Operation operation;
13   bool status;
14   Node* node;
15
16   LogEntry(bool s, NodeWithLog* n, Operation  a, int opNum):
17     operationNum(opNum), operation(a), status(s),  node(n) {}
18 };
19
20 class LogQueue {
21   Node* head;
22   Node* tail;
23   LogEntry* logs[MAX_THREADS];
24
25   LogQueue() {
26     T* sentinel = new Node(T());
27     FLUSH(sentinel);
28     head = sentinel;
29     FLUSH(&head);
30     tail = sentinel;
31     FLUSH(&tail);
32     logs[i] = NULL;  // for every thread
33     FLUSH(&logs[i]); // for every thread
34   }
35 };
```

We start by adding two additional fields to the queue nodes (see the Node class (lines 4-5 in Figure 3.4). The `logInsert` field specifies which thread enqueued this node. More specifically, this field points to the log object that represents the intent to enqueue this node on the queue. The second `logRemove` field points to the log object of the dequeuer thread that dequeued this node. Initially these pointers are NULL. To construct logs, we also introduce an inner class named `LogEntry`. A LogEntry object holds details about an operation that a thread executes. The LogEntry class

Figure 3.5: The enqueue operation of Log Queue

```
1  void enq(T value, int threadID, int operationNumber) {
2    LogEntry* log = new LogEntry(false, NULL, enqueue,
       operationNumber);
3    Node* node = new Node(value);
4    log->node = node;
5    node->logInsert = log;
6    FLUSH(node);
7    FLUSH(log);
8    logs[threadID] = log;
9    FLUSH(&logs[threadID]);
10   while (true) {
11     Node* last = tail;
12     Node* next = last->next;
13     if (last == tail) {
14       if (next == NULL) {
15         if (CAS(&last->next, next, node)) {
16           FLUSH(&last->next);
17           CAS(&tail, last, node);
18           return;
19         }
20       } else {
21         FLUSH(&last->next);
22         CAS(&this->tail, last, next);
23       }
24     }
25   }
26 }
```

contains four fields: the `operationNum` field is a unique identifier for this operation set by the caller. The `operation` field contains either *enqueue* or *dequeue*. The `status` flag indicates whether the operation has been completed, if relevant. Finally, the `node` field, which points to the associated node (which needs to be enqueued, or has been dequeued). The log queue keeps a *logs* array to maintain a log. This array has an entry for each thread, it points to this thread's logEntry, and its size is MAX-THREADS, which is the number of threads that might perform operations on the queue.

The queue constructor initializes the underlying linked list with one sentinel node. It lets the head and the tail point to this sentinel node, and it also initializes the *logs* array with NULL values indicating that no operation is currently ongoing. Finally, it flushes all relevant content: the sentinel node, the head and tail pointers, and the content of the *logs* array.

3.5.1 The Enqueue() Operation

The pseudo-code for the enqueue operation is provided in Figure 3.5. The enqueue method works with a log entry describing the intended operation and a link to the inserted node. After initialization, the node is appended to the list and the log is updated with a flag indicating completion.

The enqueue method receives the value to be enqueued, the ID of the thread that performs the enqueue and the operation ID. It starts by allocating a new log entry to describe the intended operation. It is initialized with an *enqueue* operation, a *completed* status (FALSE), a pointer to a new node with the given value and the operationNumer that specifies the operation ID. Next, it sets the `log` pointer in the new node to point to the log entry. Then, the content of the node and the log are flushed. Next, the thread entry in the log array is set to point to this log entry, and the log array field is flushed. The actual insertion to the queue consists of two actions: appending the new node at the end of the list and updating the tail to reference the newly added node. To ensure proper durability, we add the following flush instructions.

If appending the node is successful, we flush the next pointer of the previous node (line 16). Only afterwards do we update the tail (line 17). In this case we do not bother setting the status flag to true, because flushing the next pointer of the previous node indicates that the operation executed. Failure in the CAS operation to update the tail means that another thread has helped update it and has already completed the enqueue operation of the current thread. Failure in the first CAS instruction (line 15) means that another thread has appended a different node and the operation starts from scratch (line 10). If the next pointer of the last node does not point to NULL (line 20), we help complete the previous enqueue operation by flushing the previous next pointer (line 21). Finally, we fix the tail (line 22).

We maintain an important invariant that all the nodes in the queue, up to the current node pointed to by the tail, are persistent. In addition, all their log entries are fully updated in persistent memory. This ensures a consistent view for the recovery procedure. The last enqueue may not survive a crash, depending on whether the append of its node to the end of the queue persists. But it is easy to tell during the recovery whether a logged operation has completed.

3.5.2 The Dequeue() Operation

The pseudo-code for the dequeue operation is provided in Figure 3.6. The dequeue operation receives the ID of the thread that performs the dequeue. Then, it allocates a new log entry with a proper initialization and flushes the log content (lines 2-3). Next, it lets its entry in the *logs* array point to this log persistently (lines 4-5). The actual dequeue starts then. If the queue is empty, the thread updates its corresponding log status as true. The NULL pointer to the node indicates an empty returned value (lines 13-15). If the queue is empty and there is an enqueue operation in progress,

Figure 3.6: The dequeue operation of Log Queue

```
1  void deq (int threadID, int operationNumber) {
2    LogEntry*  log = new LogEntry(false, NULL, dequeue,
       operationNumber);
3    FLUSH(log);
4    logs[threadID] = log;
5    FLUSH(&logs[threadID]);
6    while (true) {
7      Node* first = this->head;
8      Node* last = this->tail;
9      Node* next = first->next;
10     if (first == this->head) {
11       if (first == last) {
12         if (next == NULL) {
13           logs[threadID]->status = true;
14           FLUSH(&(logs[threadID]->status));
15           return;
16         }
17         FLUSH(&last->next);
18         CAS(&tail, last, next);
19       } else {
20         if (CAS(&next->logRemove, NULL, log)) {
21           FLUSH(&first->next->logRemove);
22           next->logRemove->node = first->next;
23           FLUSH(&first->next->logRemove->node);
24           CAS(&head, first, next);
25           return;
26         } else {
27           if (head == first) {   //same context
28             FLUSH(&first->next->logRemove);
29             next->logRemove->node = first->next;
30             FLUSH(&next->logRemove->node);
31             CAS(&head, first, next);
32           }
33         }
34       }
35     }
36   }
37 }
```

then the dequeuer attempts to help complete the enqueue operation and dequeue again (lines 17-18). If the queue is not empty, a dequeue consists of three actions: setting the `logRemove` field of `head->next` to point to the log entry of the dequeuer (using an atomic CAS), letting this log entry point to the dequeued node, and advancing the `head`. In this case we do not bother setting the status flag to true, because the node pointer indicates that the operation executed. If the first action (line 20) succeeds,

the dequeuer flushes the logRemove field (line 21), updates the log entry, and flushes it as well. Only then does it update the head (line 24). Failure to update the head means that another thread has helped and completed the removal of the current thread. Failure to set the `logRemove` field of the node means that another thread has taken over this node, and we help that thread complete the dequeue before trying again. Here, we do not need to update the status because the association of the log with the node already indicates that the operation has been executed.

The main invariant here is that all dequeues, except maybe for the last one, are properly reflected in the log entries, and also have the dequeued node point to the appropriate log entry. The last dequeue is made durable only if the setting of the `logRemove` field persists after the crash. If the dequeue was not made durable, we will complete it after the crash.

3.5.3 The Recovery() Operation

Recall that each thread executes the recovery procedure after a crash and only then proceeds with its normal operations. The recovery procedure argument is the *logs* array from the NVMM. The procedure can be divided into six logical parts. Let A be the last node that has a non-NULL `logRemove` field, and let B be the last node that is reachable from the `head` and whose next pointer is NULL (Note that the nodes referred to as A and B change dynamically with changes in the queue).

1. If A's predecessor is reachable from the `head`, then update the `head` by a CAS operation to point to the predecessor of A. Otherwise, skip to the third step.

2. Flush A's `logRemove` field, update the log entry that is pointed by the `logRemove` field, to point to this node, flush the array entry and use a CAS operation to update the `head` to point to A (This becomes the new sentinel node).

3. If B's predecessor is reachable from the `head`, then update the `tail` by a CAS operation to point to the predecessor of B. Otherwise, update the `tail` to point B and skip to the fifth step. During the traversal, update the `status` field of the `logInsert` field of all the traversed nodes to be TRUE. This update is crucial for the fifth step, to avoid executing operations twice.

4. Flush the `next` pointer of the node that the `tail` points to, update the relevant status of the log to be TRUE. Then, update the `tail` by a CAS operation to point to B.

5. Traverse the *logs* array to finish all the started operations.

6. Create a new *logs* array and replace the old one by a CAS operation (if the old *logs* array has not been previously changed by another thread).

The main addition to the recovery function of the durable queue is the *logs* array traversal. We traverse the *logs* array and finish all the started operations.

For a dequeue operation, if the log entry is already referenced by the removed node, we flush the `node` field in the log entry and continue to the next log entry. If not, we sample the `head` and then recheck that the `node` field in the log entry is NULL. If so, we help the thread to finish its operation as follows. We check the `head` again, and if the `head` has been changed, we sample the `node` field in the current log entry again, and try again to finish the operation, unless it was already completed by another thread.

For an enqueue operation, we do something similar. If the log entry `status` flag is TRUE or the `logRemove` field is not NULL, we continue to the next log entry as this operation has completed. If not, we sample the `tail` and then recheck that the `status` field in the log entry is FALSE. If so, we help the thread to finish its operation as follows. We check the `tail` again, and if the `tail` has been changed, we sample the `status` field in the current log entry again, and try again to finish the operation, if this operation was not already completed. In this way, we first finish all pending operations from the previous phase, and only then, do we start to execute the operations from the current phase. We proceed to the current phase even if some threads are still running the recovery operation.

Note that if a certain thread finished its recovery phase and started running, threads that are in the middle of the recovery phase, will not execute any operation during that phase as there was already one thread that executed all those operations.

3.6 Algorithm Details of the Relaxed Queue

We now go into the details of the more efficient relaxed queue. This queue provides buffered durable linearization which is a weaker guarantee. The main idea is to maintain a snapshot of the queue, which has a consistent state by representing a prefix of linearized operations in the NVMM. Every `sync()` execution makes all the operations executed before the `sync()` persistent. This queue builds on two inner classes - `Node` and `LatestNVMData`. The Node class represents the nodes of the queue and is similar to the MS queue. The `LatestNVMData` class has three fields, `NVMTail`, `NVMHead` and a `NVMVersion`. It is used to keep record of the latest saved head and tail values. The `RelaxedQueue` class contains the standard *head* and *tail* and a `LatestNVMData` object named `NVMState` that maintains the latest persistent state of the queue. The invariant that we keep is that all nodes between `NVMState->NVMTail` and `NVMState->NVMHead` are durable, and so is the content of the object `NVMState` itself. An additional `version` field is used by the `sync()` method or other concurrent helping threads to distinguish one sync operation from another, so that older sync operations will not interfere with newer sync operations. In order to indicate that a snapshot is being recored by a sync operation, we use one additional class that inherits from `Node`. This class has the same fields as the `LatestNVMData` class and is called the `Temp` class.

Figure 3.7: Internal Relaxed Queue classes

```
1  class Node {
2    T  value;
3    Node*  next;
4
5    Node(T  val): value(val), next(NULL) {}
6  };
7
8  class LatestNVMData {
9    Node* NVMTail;
10   Node* NVMHead;
11   long NVMVersion;
12 };
13
14
15 class Temp: public Node {
16   int version;
17   Node* tail;
18   Node* head;
19
20   Temp(long v): version(c), tail(NULL), head(NULL) {}
21 };
22
23 class RelaxedQueue {
24   Node* head;
25   Node* tail;
26   LatestNVMData* NVMState;
27   atomic<int> version;
28
29   RelaxedQueue() {
30     Node* sentinel = new Node(T());
31     FLUSH(sentinel);
32     head = tail = sentinel;
33     FLUSH(&head);
34     FLUSH(&tail);
35     LatestNVMData* d = new LatestNVMData();
36     d->NVMTail = sentinel;
37     d->NVMHead = sentinel;
38     d->NVMVersion = -1;
39     FLUSH(d);
40     NVMState = d;
41     FLUSH(&NVMState);
42     version = ATOMIC_VAR_INIT(0);
43   }
44 };
```

Figure 3.8: The enqueue method of the Relaxed Queue

```
1  void enq(T value) {
2    Node* node = new Node(value);
3    while (true) {
4      Node* last = tail;
5      Node* next = last->next;
6      if (last == tail) {
7        if (next == NULL) {
8          if (CAS(&last->next, next, node)) {
9            CAS(&tail, last, node);
10           return;
11         }
12       } else {
13         if (next == tempAddress) {
14           CAS(&next->head, NULL, head);
15           CAS(&next->tail->next, next, NULL);
16           continue;
17         }
18         CAS(&tail, last, next);
19       }
20     }
21   }
22 }
```

The queue constructor initializes the underlying linked list with one sentinel node. It lets the head and the tail point to this sentinel node. It also initializes NVMState's `NVMTail` and `NVMHead` with pointers to the sentinel node, and `NVMVersion` with -1 value that indicates the initial state.

3.6.1 The Enqueue() Operation

The pseudo-code for the enqueue operation is provided in Figure 3.8. The enqueue method is similar to the MS queue. The only modification is in lines 13-17, where the thread checks whether it needs to help the sync() function. The sync() function records a snapshot of the head and the tail. To do that, it "freezes" enqueues and then records the value of the head and tail, knowing they reflect a snapshot. To freeze enqueues, a temporal node (that inherits from the original node) is set to the last node's `next` pointer. The condition in line 13 checks for this case, and helps the sync that is currently executing, by saving the current head to the current object (the tail is already set). It can then fix the last `next` pointer to NULL and restart the enqueue method execution.

Figure 3.9: The dequeue method of the Relaxed Queue

```
1  T deq() {
2    while (true) {
3      Node* first = this->head;
4      Node* last = this->tail;
5      Node* next = first->next;
6      if (first == this->head) {
7        if (first == last) {
8          if (next == NULL) {
9            throw EmptyException();
10         }
11         if (next == tempAddress) {
12           CAS(&next->head, NULL, head);
13           CAS(&next->tail->next, next, NULL);
14           throw EmptyException();
15         }
16         CAS(&this->tail, last, next);
17       } else {
18         T value = next->value;
19         if (CAS(&this->head, first, next)) {
20           return value;
21         }
22       }
23     }
24   }
25 }
```

3.6.2 The Dequeue() Operation

The pseudo-code for the dequeue operation is provided in Figure 3.9. The dequeue method is also similar to the MS queue with the exception that an alternative form of an empty queue is one where the sentinel's next pointer points to a temporal node. This indicates that the `sync()` method is currently running on an empty queue.

3.6.3 The Sync() Operation

The purpose of the `sync()` function is to make all the operations executed before the `sync()` persistent. To achieve that, we simply record a persistent snapshot of the queue state. Recovery can later return to this state and obtain a proper prefix (consistent cut) of the operations recovered. To record a persistent snapshot, we first save the values of `head` and `tail`, and then we make all node content between them persistent. The `sync()` function starts by allocating a new `Temp` object (with fields initialized to NULL) that will eventually replace the `NVMState` object if no other thread executes sync operations concurrently. To obtain an atomic view of `head` and `tail` simultaneously, `sync()` blocks `tail` from changing (for a very short while) and then it reads both `head`

Figure 3.10: The sync method of the Relaxed Queue

```
1  void sync() {
2    int currentVersion = 0;Temp* temp = new Temp(currentVersion);
3    while (true) {
4      currentVersion = atomic_fetch_add(&version, 1);
5      temp->version = currentVersion;
6      Node* last = tail, next = last->next;
7      if (last == tail) {
8        if (next == NULL) {
9          temp->tail = last;
10         if (CAS(&last->next, next, temp) {
11           CAS(&temp->head, NULL, head);
12           CAS(&last->next, temp, NULL);
13           break;
14         }
15       } else {
16         if (next == tempAddress) {
17           if (next->version > currVersion||next->head ==NULL){
18             CAS(&next->head, NULL, head);
19             CAS(&next->tail->next, next, NULL);
20             temp = next;
21             break;
22           }
23           CAS(&next->head, NULL, head);
24           CAS(&next->tail->next, next, NULL);
25           continue;
26         }
27         CAS(&tail, last, next);
28       }
29     }
30   }
31   Node* currNode = temp->head;
32   FLUSH(currNode);
33   while (true) {
34     if (currNode == temp->tail) break;
35     Node* next = currNode->next;
36     FLUSH(next);
37     currNode = next;
38   }
39   LatestNVMData* currNVMState = NVMState;
40   LatestNVMData* potential = new LatestNVMData();
41   potential->NVMTail = temp->tail;
42   potential->NVMHead = temp->head;
43   potential->NVMVersion = temp->version;
44   while (true) {
45     if (currNVMState->NVMVersion < temp->version) {
46       FLUSH(potential);
47       if (CAS(&NVMState, currNVMState, potential)) {
48         FLUSH(NVMState); break;
49       } else { currNVMState = NVMState; }
50     } else { break; }
51   }  return; }
```

and `tail`, and then unlocks the `tail`. To block `tail`, `sync()` installs a special pointer at the `next` field of the last node. This lets all other threads notice the block and help the `sync()` function before proceeding with further enqueue operations (lines 3-30). After sampling the `tail` and installing this special pointer, the `sync()` function (or any thread helping it) uses a CAS instruction to save a current `head` value (the `tail` value is already recorded in line 9) into the `Temp` object. Assuming a NULL value in that field, implies that the value of `head` in `Temp` can only change once and a stalled thread cannot later wake up and foil these values. Next, the special pointer in the last node can be switched back to NULL, enabling further enqueues to proceed (line 12). The `sync()` function proceeds by flushing the content of all nodes in the queue between the recorded `head` and `tail` (lines 33-38). Then it checks whether there was another snapshot that was concurrently recorded. A higher `NVMValue` value in the `NVMState` object represent a more progressed snapshot. Thus, if there was another snapshot that was concurrently recorded, with a higher value, then that snapshot can also be used by the currently executing thread, and we can simply exit. If another snapshot was recorded with a lower value, then it is outdated and cannot be used. In this case, we try to record the newer snapshot in `NVMState` and flush the `NVMState` field to the memory upon success. An optimization that we use is that if a thread notices a higher `sync()` operation that is blocking the `tail` or a lower `sync()` function that is also blocking the `tail` but whose `head` is still not updated, then it can cooperate with that snapshot and use it. Another optimization that we use for large queues is that instead of flushing the content of all nodes in the queue between the recorded `head` and `tail`, we only flush the content between the previous recorded `tail` and the current recorded `tail`.

3.6.4 The Recovery() Operation

The recovery operation, simply sets the head and the tail to their saved values in the `NVMState` object. It also sets the `next` field of the node pointed to from the new tail to NULL. This can be done quickly by a single thread before all threads start running the operations on the queue.

Buffered durability is obtained here because, whenever a crash occurs, we simply return to a previous state of the queue. This means that a prefix of the previous operations takes place and a consistent cut of the operations remains.

3.7 Memory Management

Existing technology can be used to handle object reclamation. In particular, we can use hazard pointers [73] for reclamation and the Makalu NVM allocator [14] for allocations and reclamation after a crash. In our measured implementation we included hazard pointers, as described below, but did not incorporate the Makalu allocator. The Makalu allocator is new and we have not yet had the chance to examine it and use it. We simply

used the standard `g++` GNU allocator instead.

Makalu allocator reduces overhead by lazily persisting metadata that is easily recoverable if there is a crash, and by employing a garbage collector after a failure. In case of a crash, Makalu performs a parallel mark-sweep to reclaim all data that was not accounted for properly at crash time. It also restores certain metadata, which eliminates the necessity of ensuring full consistency of metadata at every allocation. The Makalu provides an API for allocating and freeing NVMM objects, which can replace the `new` and `free` instructions in our implementation.

Memory management for lock-free data structures is a non-trivial task as threads may read a pointer to an object and then delay executing for a while. Until they wake up, the referenced object must not be reclaimed. Michael [73] proposed that threads announce objects that should not be claimed by publishing hazard pointers to them. In our implementation, we followed the hazard pointer scheme provided in [73] and used the implementation of [74]. Our extension had to ensure that hazard pointers are properly recorded also by threads that help other threads persist objects in the queue (in the durable and the log queues) and they were also used to enable reclamation of the log objects in these two queues. The most challenging work was with the relaxed queue as threads can concurrently call the `sync()` function and reclaim objects. We followed the principle that only a thread that managed to change the current snapshot was allowed to reclaim the objects from the `head` of the previous snapshot to the `head` of the current snapshot, and a thread that did not manage to update the snapshot once tried to take a new snapshot of the queue.

We did not attempt to persist the reclamation lists or hazard pointers. First, hazard pointers become irrelevant upon a crash, because threads do not proceed in the middle of an operation and so protecting objects that they locally reference is no longer required. New threads come up; they recover old operations and start new ones. As for the reclamation lists, they all contain objects that are no longer reachable from the data structure. Previously, they could only be reached from local pointers. After a crash, all objects not accessible from roots are reclaimed by an offline garbage collector that Makalu executes during the recovery.

Our implementation includes the hazard pointers and the measurements include the (non-trivial) overhead associated with maintaining them.

3.8 Measurements

We evaluated the performance of the proposed three queue implementations by comparing them one against the other and also against the original MS queue. We ran measurements on a 64-core machine, featuring 4 **AMD** Opteron(TM) 6376 2.3GHz processors, each with 16 cores. The machine has 128GB RAM, an L1 cache of 16KB per core, an L2 cache of 2MB for every two cores, and an L3 cache of 6MB per half a processor (8 cores). The operating system is Ubuntu 14.04 (kernel version 3.16.0).

As in previous work [14, 20, 78], we measured the performance of the execution with flushes on a real system because we assume that an NVMM will use a controller that will write data quickly into a local fast VM. We also assume that upon a crash, local batteries will allow saving the remaining local volatile data to the NVMM. Thus, the actual flush cost is expected to be similar to the one we see on current platforms.

Since the queue is not a scalable data structure, executions with many threads are not relevant and we only measured 1-8 threads. Each execution lasted 5 seconds. All functions were implemented in `C++` and compiled using the `g++` compiler version 6.2 with the -O3 optimization flag. Memory management was handled with hazard pointers, and is described in Section 3.7. In our implementation, we followed the hazard pointer scheme provided in [73] and used the implementation of [74]. Following [75, 62], we evaluated the performance of the queue algorithms with a workload that lets several threads run enqueue-dequeue pairs concurrently. The queue is either initiated with 5 enqueued elements (for a small queue), or $1,000,000$ enqueued elements (for a large queue).

We depict the difference in the throughput of the MS queue and our three new algorithms across different numbers of threads. Each test was repeated 10 times and the average throughput is reported. The x-axis denotes the number of threads, and the y-axis stands for millions of operations per second. A high number is better, meaning that the measured scheme has higher throughput. With hazard pointers (designed in [73], and implemented in [74]) the memory management overhead is large and the results of Figure 3.11 and Figure 3.12 are less indicative of the bare queue actual performance. This is why we also provide the measurements without memory management in Figure 3.13. As expected, queues that provide weaker durability guarantees perform better in most cases, with the exception being when the queue is very large and the garbage collection costs dominate performance. We believe that the reason for this is that large queues employ many hazard pointers and this cost is similar to all queue variants. In contrast, small queues can avoid some flushes. When the `sync()` function is infrequently called, the new head may pass the old tail between snapshots, reducing the required number of flushes, and eliminating most of the hazard pointer uses. Surprisingly, the relaxed queue performs better than the MS queue without garbage collection. We believe this is due to an implicit back-off effect that the slower queue creates.

We ran the relaxed queue and let each thread execute the `sync()` function every $K*N$ operations, where K varies between, 10, 100, 1000 and 10000 and N is the number of the threads. We omitted the $K = 10000$ results because they are similar to the $K = 1000$ results. As each of the N threads executes a sync every $K*N$ operations, we get that on average a sync is executed in the system after each thread executes K operations.

Finally, we study how the flushes and the field that we added to the durable queue affect the performance. To this end, we measured an execution of the original MS queue

and three intermediate versions between the original MS queue and the durable queue. The first intermediate version only adds flushes for enqueue nodes, as required by the durable algorithm. The second intermediate version maintains an extra field in the node in which the dequeuing thread writes its identity. This field is then properly flushed. The third intermediate version performs flushes on the queue nodes and also maintains an extra field in the node in which the dequeuing thread writes its identity. This extra field is then properly flushed according to the algorithm. Finally, the full durable queue is run, where, in addition to the above overhead, we also use an array for returned values and flush those when necessary. In order to measure these overheads with minimal interference, we measured those queue performances without memory reclamation, so only our additions would take effect. The results for the AMD platform are presented in Figure 3.14. We note that the most dominant operation is node flushing, which has a big impact on performance. Adding the extra field and the flushes for it is cheaper than flushing the enqueued nodes. The cost of the extra field is represented by throughput that is $6.6x$ and $1.4x$ lower when running, respectively, 1 and 8 threads. The cost of the flushes is represented by throughput that is $9.15x$ and $1.65x$ lower when running, respectively, 1 and 8 threads.

We also provide additional measurements on Intel similar to the ones reported for AMD. While the performance numbers are different, the trend is very similar. The platform is an 8-core **Intel** Xeon D-1540 2.6GHz with hyper-threading. This machine has 8GB RAM, an L1 cache of 32KB per core, an L2 cache of 256KB, and an L3 cache of 12MB. The operating system is Ubuntu 14.04 (kernel version 4.4.0).

The Intel version for Figure 3.13 is provided in Figure 3.17. The Intel version for Figure 3.11 is provided in Figure 3.15, and the Intel version for Figure 3.12 is provided in Figure 3.16. On the Intel platform with no memory management, the relaxed queue performs worse by $3.7x$ when running one thread and performing `sync()` every 10 operations, and by $2.6x$ when performing `sync()` every 1000 operations. When there are more than two threads running concurrently, it performs worse by $1.03x$, when performing `sync()` every 10 operations, and better by $1.15x$ when performing `sync()` every 1000 operations. It usually slightly outperforms the original queue. The durable queue has $12.5x$ lower throughput while running one thread, and $3.14x$ lower throughput while running two threads. However, it improves as the number of threads increases, until it reaches a factor of $2.2x$ while running 8 threads. The log queue has $2.3x$ lower throughput when running 8 threads, almost the same as the durable, $3.9x$ lower throughput when running 24 threads, and $16.2x$ lower throughput when running one thread.

Finally, the Intel version for Figure 3.14 is provided in Figure 3.18. It seems that the node's flush is the most effective when 1-4 threads are running, but then the extra field is more dominant when $5-8$ threads are running. The cost of the nodes flushes is represented by throughput that is $4.7x$, $1.6x$ and $1.2x$ lower when running, respectively, 1, 4 and 8 threads. However, the cost of the extra field is represented by throughput

that is $3.2x$, $1.6x$ and $2x$ lower when running, respectively, 1, 4 and 8 threads.

3.8.1 Optimizations

As indicated above, flushes typically need to be accompanied by a memory fence in order to guarantee that the write back is executed before continuing the execution. We omitted some fencees when the ordering of the flushes is not important or whenever there is a flush instruction followed by a CAS instruction with no shared write operation in-between them. In our implementation, some fence instructions can be omitted because in the Intel and AMD machines, atomic instructions (such as CAS and FAA) execute an implicit fence. In other words, any following CAS instruction serves as a fence, guaranteeing completion of previous flushes.

3.9 Related Work

To the best of our knowledge, the presented queues are the first lock-free data structure designed for adapted execution with NVMM. Several papers propose definitions for durability. In this research we work with the definition of [59] but our algorithms and guidelines suit other definitions as well. In [78] the authors propose alternative definitions, some of which require hardware modifications. They also design a queue, but it is not lock-free. They use a lock (with additional flushes) to synchronize queue access.

Several prior works proposed transactional updates to persistent memory that guarantee *failure atomicity* – a collection of persistent data updates all occur or none do across failure boundaries [20, 23, 43, 61, 68, 88]. While these approaches work, they trade off performance for consistency in the face of failures – transaction runtimes incur significant bookkeeping overheads to consistently manage transaction metadata. An interesting alternative strategy to transactional updates is to build libraries of high performance persistent data structures [37] that are heavily optimized using ad hoc techniques informed by the data structure architecture and semantics. This is the focus of the current work.

Several other papers proposed using stable storage to maintain the state of the object. In [3] the authors propose solving consensus using stable storage by recording the state of the processes every round. Another paper [**?**] optimizes the logging procedure and provides a logarithmic lower bound for robust shared memory emulations.

Recently, [25] studied the construction of an efficient log adequate for non-volatile memory in the same settings as this work. This protocol can be extended to build an efficient single-threaded hash map.

3.10 Conclusion

In this work we presented three designs for lock-free concurrent queues that can be used with non-volatile memory. These designs demonstrate avenues to deal with durable linearizability, buffered durable linearizability (with a `sync` operation), and detectable execution. As expected, full durable linearizability has a substantial performance cost. In contrast, buffered durable linearizability incurs a lower overhead as long as the `sync` operations are infrequently called.

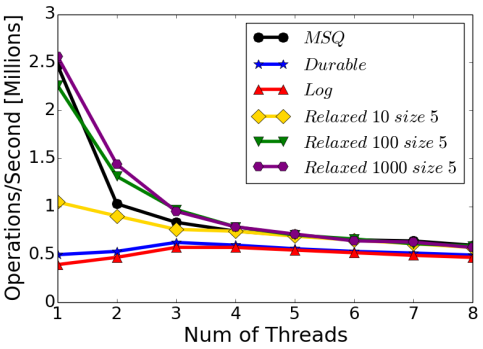


Figure 3.11: Throughput of the various queue implementations with memory management on the **AMD** platform.
Initial size of the queue is 5.

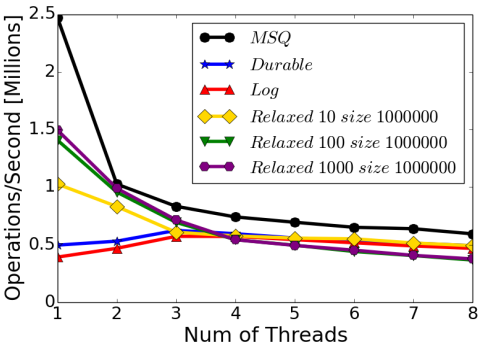


Figure 3.12: Throughput of the various queue implementations with memory management on the **AMD** platform.
Initial size of the queue is $1,000,000$.

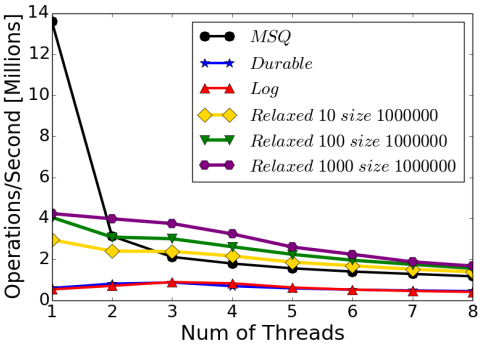


Figure 3.13: Throughput of the various queue implementations with no object reuse on the **AMD** platform.

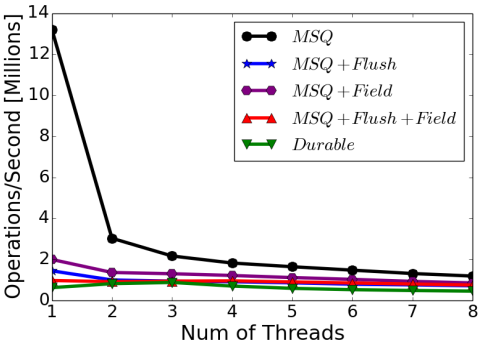


Figure 3.14: MSQ, only flushes added, flushes and additional fields, and the entire durable queue on the **AMD** platform.
Throughput reported in millions of operations per second.

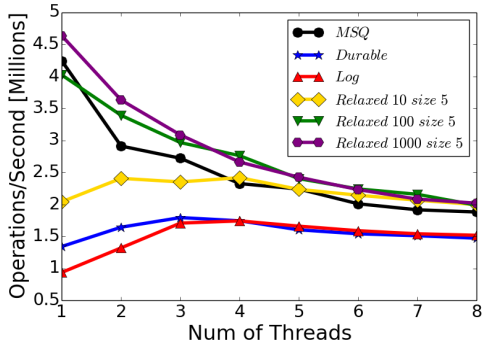


Figure 3.15: Throughput of the various queue implementations with memory management on the **Intel** platform. Initial size of the queue is 5.

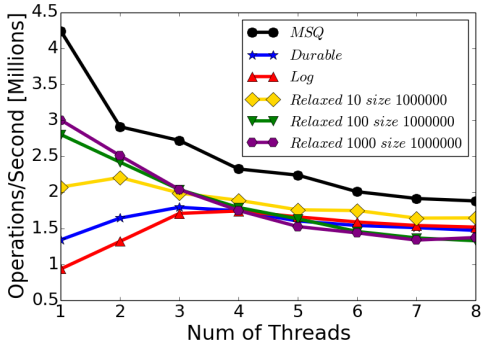


Figure 3.16: Throughput of the various queue implementations with memory management on the **Intel** platform. Initial size of the queue is $1,000,000$.

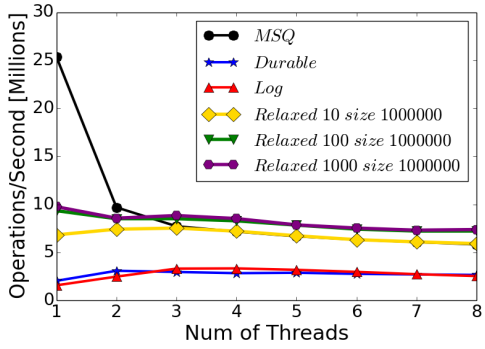


Figure 3.17: Throughput of the various queue implementations with no object reuse on the **Intel** platform.

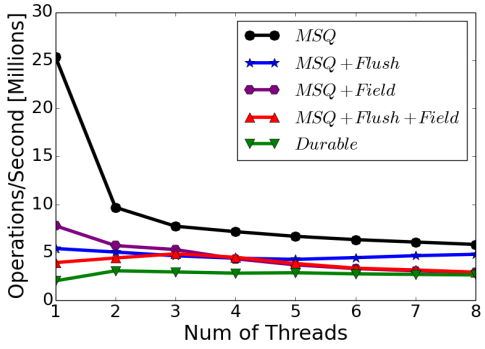


Figure 3.18: MSQ, only flushes added, flushes and additional fields, and the entire durable queue on the **Intel** platform.
Throughput reported in millions of operations per second.

Chapter 4

A General Construction: NVTraverse

4.1 Introduction

We continue with NVTraverse [40], which is a general transformation that takes a lock-free data structure from a general class called a *traversal data structure* (that we formally define) and automatically transforms it into an implementation of the data structure for the non-volatile memory setting that is provably durably linearizable and highly efficient. We take a substantial step toward removing the need for expert familiarity with an algorithm to make an efficient durably linearizable version of it. The transformation hinges on the observation that many data structure operations begin with a traversal phase that does not need to be persisted, and thus we only begin persisting when the traversal reaches its destination. We take a substantial step in removing the need for expert familiarity with an algorithm to make an efficient durably linearizable version of it. This form is formalized with a large class of lock-free linearizable algorithms we call *traversal data structures*. Traversal data structures are node-based tree data structures whose operations first traverse the data structure, and then perform modifications on nodes that descend from where their traversal stopped. The traversal is guaranteed to only make local decisions at every point in time, not relying on previous nodes to determine how to proceed from the current one. These algorithms also follow some natural rules when removing nodes from the data structure. Many existing pragmatic concurrent algorithms can easily be converted into traversal data structures, without losing their efficiency. Thus, it is not hard to transform a new data structure implementation into a traversal form.

We show that many existing pragmatic concurrent algorithms can easily be converted into traversal data structures, without losing their efficiency. Many data structure implementations such as Harris's linked-list, common BST, (a,b)-tree, and hash table algorithms [49, 36, 16, 17], can easily be converted. Furthermore, traversal data structures capture not just set data structures, but also queues, stacks, priority queues,

skiplists, augmented trees, and others. Most requirements of traversal data structures are naturally satisfied by many lock-free data structures. Thus, it is not hard to transform a new data structure implementation into a traversal data structure.

After defining traversal data structures, we show how to automatically inject flush and fence instructions into a traversal data structure to make it durably linearizable. The key benefit of our approach is that no flushes are needed during most of the traversal. Of all memory locations that are read, only a few at the end of the traversal must be flushed. Because of the careful way in which the traversal is defined, these can be automatically identified. After the traversal, all further locations that the operation accesses must be flushed. In most operations, however, the traversal encapsulates a large majority of the work. We formally prove that the flushes and fences that we specify are sufficient for durability for all traversal data structures. Therefore, this work presents the first practical, provably correct implementation of many durable data structures; the only previously known durable algorithm that was proven correct is the *durable* queue [41].

We experimentally evaluate the algorithms that result from our transformation, by transforming a list, two binary search trees, a skiplist, and a hash table into the traversal form and then injecting flushes automatically to obtain a durable data structure. We compare our implementations to those that result from the general transformation of Izraelevitz et al. [59], the original (non-persistent) version, the OneFile transactional memory [84], and the hand-tuned durable version presented by David et al. [32]. Our results show that NVTraverse data structures outperform Izraelevitz et al. [59]'s construction significantly on all workloads. Furthermore, NVTraverse data structures outperform David et al. [32]'s data structures on about half of the workloads – those with lower thread counts or larger data structure sizes. This provides some interesting insights on the trade-offs between flushes, fences, and writes when it comes to contention.

The method proposed in this work comprises two steps. The first step (which is manual) involves making sure that the target lock-free data structure is in the traversal format. The second step (which is automatic) involves adding flushes and fences to make the lock-free data structure durable. The second step is the major contribution of this work, because it saves programmers the effort of having to reason about persistence. On the other hand, the definition of traversal data structures is not as simple as we would have wanted it to be. While many data structures are already in traversal form, the programmer must verify that this is the case for their data structure before using our transformation. Sometimes, small modifications are required to make a data structure a traversal one.

The rest of this chapter is organized as follows. Section 4.2 reviews previous work and some notations. Section 4.3 provides a definition of a large class of concurrent data structures called *traversal data structures*. We provide the details of the automatic transformation of any traversal data structure to a durable one in Section 4.4. The

experimental evaluation is presented in Section 4.7. Section 4.8 discusses related work and Section 4.9 concludes.

4.2 Preliminaries

Throughout the chapter, we sometimes say that a node n in a tree data structure is *above (resp. below)* another node n' if n is an ancestor (resp. descendant) of n'.

We refer to write and successful CAS instructions collectively as *modifying* instructions, and to modifying instructions, and return statements collectively as *externally visible* instructions. In the experiments, we use the term *threads* instead of *processes*.

4.2.1 Running Example: Harris's Linked List

Throughout the chapter, we refer to the linked-list presented by Harris [49] when discussing properties of traversal data structures. We now briefly describe how this algorithm works.

Harris [49] presented a pragmatic linearizable lock-free implementation of a sorted linked-list. The linked-list is based on nodes with an immutable *key* field and a mutable *next* field, and implements three high-level operations: insert, delete, and find, which all take a key k as input. Each operation takes a key k as input; the insert operation inserts a new node with key k into the data structure if it does not already exist, the delete operation removed the node with key k from the list if it does exist, and the find operation returns a boolean indicating whether or not a node with key k is in the list. Each of these operations is implemented in two stages: first, the helper function *search* is called with key k, and after it returns, changes to the data structure are made on the nodes that the search function returned. The search function always returns two adjacent nodes, *left* and *right*, where *right* is the first element in the list whose key is greater than or equal to k, and *left* is the node immediately before it.

We now briefly describe how each of the operations is implemented, omitting some edge cases, but conveying the key algorithmic insights. To insert a node, the operation simply initializes a node with *key* k and *next* pointing to the right node returned from the search function, and then swings *left*'s *next* pointer to point to the newly initialized node (using a CAS with expected value pointing to *right*). If the CAS fails, the insert operation restarts. The find operation is even simpler; if *right*'s key is k, then it returns true, and otherwise it returns false.

The subtlety comes in in the delete operation. The *next* pointer of each node has one bit reserved as a special *mark* bit. If this bit is set, then this node is considered *marked*, meaning that there is a pending delete operation trying to delete this node. If a node is marked, we say that it is *logically deleted*. More specifically, a delete operation, after calling the search function, uses a CAS to mark the right node returned by the search, if the key of *right* is k. After successfully marking the right node, the delete

operation then *physically deletes* the right node by swinging *left*'s *next* pointer from *right* to *right.next*. This two-step delete is crucial for correctness, avoiding synchronization problems that may arise when two concurrent list operations contend.

The search function guarantees that neither of the two nodes that it returns are marked, and that they are adjacent. To be able to guarantee this, the search function must help physically delete marked nodes, since otherwise, a search function may wait arbitrarily long for a pending delete operation to complete, and the algorithm would lose its lock-freedom. Thus, the search function finds the *right* node, which is the first unmarked node in the list whose key is greater than or equal to k, and the *left* node, which is the last unmarked ancestor of *right*. Before returning, the search function physically deletes all nodes between the two nodes it intends to return.

4.3 Traversal Data Structures

In this section, we introduce the class of data structures we call *traversal data structures*, and the properties that all traversal data structures must satisfy. In Section 4.4, we show an easy and efficient way to make any traversal data structure durable. We begin with two simple yet important properties.

Property 1 (Correctness)**.** A traversal data structure is linearizable and lock-free.

Property 2 (Core Tree)**.** A traversal data structure is a node-based tree data structure. In addition to the tree, there may be other nodes and links that are auxiliary, and are only ever used as additional entry points into the tree.

The part of the data structure that needs to be persistent and survive a crash is called its *core*. The other parts can be stored in volatile memory and recomputed following a crash. Property 2 says that only the *core* part of a traversal data structure needs to be a tree.

For example, a skiplist can be a traversal data structure, since, while the entire structure is not a tree, only a linked list at the bottom level holds all the data in the skiplist, while the rest of the nodes and edges simply serve as a way to access the linked list faster. Similarly, data structures with several entry points, like a queue with a head and a tail, can be traversal data structures as well. Of course, all tree data structures fit this requirement.

More precisely, the core of a traversal data structure must be a *down-tree*, meaning that all edges are directed and point away from the root. For simplicity, we use the term *tree* in the rest of the chapter.

Property 2 is important since it simplifies reasoning about the data structure, and thus allows us to limit flush and fence instructions that need to be executed to make traversal data structures durable. Note that many pragmatic data structures, including

queues, stacks, linked lists, BSTs, B+ trees, skiplists, and hash tables have a core-tree structure. Optionally, a traversal data structure may also provide a function to reconstruct the structure around the core tree at any point in time. However, our persistent transformation maintains the correctness of the core tree regardless of whether such a function exists.

A traversal data structure is composed of three methods: *findEntry*, *traverse*, and *critical*. These are the only three methods through which a traversal data structure may access shared memory, and they are always called in this order. The operation findEntry takes in the input of the operation, and outputs an entry point into the core tree. That is, the findEntry method is used to determine which shortcuts to take. This can be the head of a linked-list, a tail of a queue, or a node of the lowest level of the skip list, from which we traverse other lowest-level nodes. Note that findEntry, and is allowed to simply return the root of the tree data structure, e.g., the head of a linked-list.

Once an entry point is identified, a traversal data structure operation starts a traverse from that point, at the end of which it moves to the critical part, in which it may make changes to the data structure, or determine the operation's return value. The critical method may also determine that the operation must *restart* with the same input values as before. However, the traverse method may not modify the shared state at all. The operation execution between the beginning of the findEntry method and the return or restart statement in the critical method is called an *operation attempt*. Each operation attempt may only have one call to the findEntry method, followed by one call to the traverse method, followed by one call to a critical method. The layout of an operation of a traversal data structure is shown in Figure 4.1. Operations may not depend on information local to the process running them; an operation only has access to data provided in its arguments, one of which is the root of the data structure. The operation may traverse shared memory, so it can read anything accessible in shared memory from the root. No other argument is a pointer to shared memory. This requirement is formalized in Property 3.

Property 3 (Operation Data)**.** Each operation attempt only has access to its input arguments, of which the root of the data structure is the only pointer to shared memory. Furthermore, it accesses the shared data only through the layout outlined in Figure 4.1.

By similarity to the original algorithm of Harris, the traversal version of Harris's linked-list is linearizable and lock-free (thereby satisfying Property 1), and the data structure is a tree (thereby satisfying Property 2). Furthermore, Harris's linked-list implementation gets the root of the list as the only entry point to the data structure and it only uses the input arguments in each operation attempt. Each operation of Harris's linked-list can be easily modified to only use the three methods shown in Figure 4.1. The findEntry method simply returns the root. The traverse method encompasses the *search* portion of the operation, but does not physically delete any

Figure 4.1: Operation in a traversal data structure

```
1 T  operation (Node  root, T' input) {
2   while (true) {
3     Node entry = findEntry (root, input);
4     List<Node> nodes = traverse (entry, input);
5     bool restart, T val = critical (nodes, input);
6     if (!restart)
7       return val;
8   }
9 }
```

marked nodes. Instead, the traverse method returns the *left* and *right* nodes identified, as well as any marked nodes between them. The rest of the operation is executed in the critical method. Therefore, Harris's linked-list can easily be converted to satisfy Property 3.

In the rest of this section, we discuss further requirements on traversal data structures, which fall under two categories: traversal and disconnection behavior (when deleting a node). We also show that Harris's linked-list can easily be made to satisfy these properties, and thus can be converted into a traversal data structure.

From now on, when we refer to a traversal data structure, we mean only its core tree, unless otherwise specified.

4.3.1 Traversal

Intuitively, we require the traverse method to "behave like a traversal". It may only read shared data, but never modify it (Item 1 of Property 4), and may only use the data it reads to make a local decision on how to proceed. The traverse method starts at a given node, and has a *stopping condition*. After it stops, it returns a suffix of the path that it read. In most cases the nodes that it returns are a very small subset of the ones it traversed. The most common use case of this is that the traversal stops once it finds a node with a certain key that it was looking for, and returns that node. However, we do not specify what this stopping condition is, or how many nodes are returned, to retain maximum flexibility.

Item 2 and Item 3 of Property 4 formalize the intuition that a traversal does not depend on everything it read, but only on the local node's information. The traverse method proceeds through nodes one at a time, deciding whether to stop at the current node, using only fields of that node, and, if not, which child pointer to follow. The child pointer decision is made only based on immutable values of this node; intuitively, if a node has an immutable key and a mutable value then keys can be compared, but the node's value cannot be used. The stopping condition can make use of both mutable and immutable fields of the current node.

If the traverse method does stop at the current node, it then determines which nodes to return. This decision may only depend on the nodes returned; no information from earlier in the traversal can be taken into account (Item 4 of **??** 4). Intuitively, we allow the traverse method to return multiple nodes since some lock-free data structures make changes on a *neighborhood* of the node that their operation ultimately modifies. Examples of such a data structure include Harris's linked-list [49], Brown's general tree construction [17], Ellen et al.'s BST [36], Herlihy et al.'s skip-list [52], and others. All of these data structures make changes on the parent or grandparent of their desired node, or find the most recent unmarked node under some marking mechanism.

Note that we allow arbitrary mutable values to be stored on each node. However, we add one more requirement. Intuitively, a non-pointer-swing change on a node may not make a traversal return a later node than it would have had it not seen this change. More precisely, suppose traversal T_1 reads a non-pointer value v on node n and decides to stop at n. Consider another traversal T_2 with the same input as T_1 that reads the same field after the value v was modified. We require that T_2's returned nodes be at or above n. Note that we consider a 'marking' of a node to be a non-pointer value modification, even though some algorithms place the mark physically on the pointer field. It is easiest to understand this requirement by thinking about deletion marks; suppose v is a mark bit, and node n is marked for deletion between T_1 and T_2. Then if T_1's search stopped at n (i.e., it was looking for the key stored at n and found it), it's possible T_2 may continue further, since n is now 'deleted'. However, when T_2 returns, it must return a node above n, since the operation that called it must be able to conclude that n's key has been deleted. This will be important for persisting changes that affected the return value of T_2's operation. This property can be thought of as a *stability* property of the traversal; it may stop earlier, but may not be arbitrarily perturbed by changes on its way. We formalize this in Item 5 of Property 4. We say that a *valueChange* is any node modification of a non-pointer value (i.e., not a disconnection or an insertion).

Property 4 (Traversal Behavior)**.** The traverse method must satisfy the following properties.

1. **No Modification:** It does not modify shared memory.
2. **Stopping Condition:** Only the current node is used to decide whether or not to continue traversing.
3. **Traversal Route:** Only *immutable* values of the current node are used to determine which pointer of the current node to follow next.
4. **Traversal Output:** The output may be any suffix of the path traversed. The decision of which nodes to return may only depend on data in those returned nodes.

5. **Traversal Stability:** Consider two traversals T_1 and T_2 such that both of them have the same input and read the same field f of the same node n. Let m be a *valueChange* of f that occurs after T_1's read but before T_2's read. If T_1 stopped at n, then T_2 returned n or a node above n.

We now briefly show how the traversal version of Harris's linked list algorithm can satisfy Property 4. Recall that the traversal of each operation in Harris's linked-list is inside the search function, which begins by finding the first node in the list that is unmarked and whose key is greater than or equal to the search's key input (this node is called the *right* node). It then finds the closest preceding unmarked node, called the *left* node. We define the search function up to the right node as the traverse method. The traverse method then returns all nodes from *left* to *right*. At every point along the traversal, it uses only fields of the current node to decide whether or not to stop. If not, it always reads the *next* field and follows that pointer; its decision of how to continue its traversal does not depend on any mutable value that it reads. The returned nodes depend only on values between *left* and *right*. Thus, Items 1, 2, 3, and 4 of Property 4 are satisfied.

To see that Item 5 is satisfied, note that the only valueChange in Harris's algorithm is the marking of a node for deletion. So, if a traversal T_1 stops at a node n (i.e., n is the *right* node of the search), and a traversal T_2 with the same input sees n marked, then T_2 would stop after n, but would return a node above n (T_2's *left* and *right* nodes must both be unmarked, and n must be in between them).

4.3.2 Critical Method: Node Disconnection

The only restrictions we place on the critical method's behavior are on how nodes are *disconnected* from the data structure. Disconnections may be executed to delete a node from the data structure, but some implementations may disconnect nodes to replace them with a more updated version, or to maintain some invariant about the structure of a tree.

Many lock-free data structure algorithms [49, 17, 36, 52, 77] first logically delete nodes by *marking* them for deletion before physically disconnecting them from the data structure. This technique prevents the logically deleted nodes from being further modified by any process, thus avoiding data loss upon their removal. We begin by defining marking.

Definition 4.3.1 (Mark)**.** A node is *marked* if the mark method has been called for it. Once a node is marked, no field in it can be modified.

We require that before any node is disconnected, it is marked (Item 1 of Property 5). Sometimes, the marking of a node/nodes, denoted as S consists of changing the fields in multiple nodes. For example, in the BST of Ellen *et al.* [36], the parent of S's and the grandparent of S's values are changed to indicate the marking of S. They hold a

descriptor indicating the disconnection that needs to be executed to remove S. Once the final field is written to specify the marking of S, the node is considered marked and cannot be modified.

Furthermore, marking is intended only for nodes to be removed from the data structure. To formalize that, Item 2 of Property 5 states that there is *always* a legal instruction that can be executed to atomically disconnect a given marked, connected subset of nodes S from a traversal data structure. An instruction is considered *legal* if it is performed in some extension of the current execution. We further require that at each configuration, there is *at most* one legal disconnect instruction for a contiguous set of marked nodes S. This in effect means that the marks themselves must have enough information encoded in them to uniquely identify the disconnection instruction that may be executed. Some data structures, like Harris's linked list, achieve this trivially, since there is only ever one way to disconnect a node. Other data structures use *operation descriptors* inside their marking protocol, which specify what deletion operation should be carried out [17, 36].

Let $P^*(S)$ be all the nodes whose values are changed as part of the marking of S, indicating the unique physical disconnection that needs to be executed. We make an additional requirement for algorithms whose marking of a node includes changing values (marking) multiple nodes, including nodes that are not intended for deletion (as discussed following Definition 4.3.1). Denote all mark indications on nodes that are not intended for deletion as *external marks*. Such algorithms usually remove the external marks after the marked nodes have been disconnected. We specifically require that a thread that removes external marks either attempted to execute the disconnection, or has read the updated pointer whose update executed the disconnection. We do not allow a removal of external marks "out of the blue" by threads that have not been involved in the disconnection or that have not seen the updated pointer. This property is formalized in Item 3 below, and it holds for all known lock-free algorithm (that use extra marking indications outside the marked node).

It is also important that marked nodes can be removed from the data structure in any order. This property is formalized in Item 4.

Property 5 (Disconnection Behavior)**.** In a traversal data structure, node disconnections satisfy the following properties:

1. **Mark Before Delete:** Before any node is disconnected from a traversal data structure, it must be marked.

2. **Unique Disconnection:** Consider a configuration C and let S be a connected subset of marked nodes in the core tree of a traversal data structure. Let $P(S)$ be the parent of the root of S. If $P(S)$ is unmarked, then there is exactly one legal instruction on $P(S)$ that atomically disconnects exactly the nodes in S.

3. **Delete Before Mark Removal:** Let m be the unique disconnection instruction on $P(S)$ that atomically disconnects exactly the nodes in S, and let l be the location of the pointer that is modified to disconnect S. Before a thread t reverts an external mark, it must execute m or fails to execute m because another thread has done it earlier, or it must read l after it was modified to disconnect S.

4. **Irrelevant Disconnection Order:** Let $N = \{n_1 \dots n_k\}$ be the set of nodes that were marked at configuration C. Let E_1 and E_2 be two executions that both start from the same configuration and only perform legal disconnecting operations. If all marked nodes are disconnected after E_1 and E_2, then the state of the nodes in the data structure after E_1 and after E_2 is the same.

We now argue that Harris's linked list satisfies Property 5. A node in Harris's linked list is considered marked if the lowest bit on its next pointer is set. Once this bit is set, the node becomes immutable. Item 1 of Property 5 is satisfied because a node can only be disconnected if it is marked. If S is a set of marked nodes with an unmarked parent P, S can be disconnected by a CAS that swings $P.next$ from pointing to the first node of S to pointing to the node after the last node of S. This is the only legal instruction on P that is able to disconnect exactly the nodes in S, so Item 2 of Property 5 is satisfied. If several nodes are marked, removing them in any order yields the same result: a list with all of the marked nodes removed, and all of the rest of the nodes still connected in the same order. Thus, all items of Property 5 are satisfied.

4.3.3 Algorithmic Supplements

We now present two additional requirements for traversal data structures. These requirements are imposed so that a traversal data structure can go through the transformation to being persistent. We do not expect these properties to naturally appear in a lock-free algorithm and we therefore call them 'supplements'. They should be added to a data structure for it to become a traversal data structure. Both supplements are easy to implement.

Supplement 1. There is a function $disconnect(root)$ which takes in the root of the traversal data structure and satisfies the following properties:

1. If no traversal data structure operation takes a step during an execution of $disconnect(root)$, then there will be no marked nodes at the end of the $disconnect(root)$.

2. $disconnect(root)$ can only perform disconnect instructions defined in Item 2 of Property 5 and no other modifying instructions.

3. $disconnect(root)$ can be run at any time during an execution of the traversal data structure (without affecting the linearizability of the traversal data structure).

The *disconnect*(*root*) operation can be implemented by traversing the data structure and using the the unique atomic disconnection instruction for the marked nodes. For Harris's linked list, we can supply a function that traverses the linked list from the root pointer and trims all the marked nodes.

The second supplement that we require for a traversal data structure is that it keeps an extra field in each node, which stores the *original parent* of this node in the data structure. Since the data structure is a tree, a node can only have a single parent when it joins the data structure. We require the address of the pointer field that was changed to link in the new node to be recorded in the extra field. Note that it is possible that a sub-tree is added as a whole by linking it to a single (parent) pointer in the data structure. In this case, that same parent pointer should be stored in all the nodes of the inserted sub-tree. The location of this pointer must be stored in the original parent field *before* the node is linked to the data structure to ensure that this field is always populated.

Supplement 2. A designated field in each node n, called *the original parent (OP)* of n, must store the location of the pointer that was used to connect n to the data structure.

In Section 4.4 we specify how this field is used. Adding a field to the data structure may be space consuming, so we also propose an optimization that can avoid storing this field.

In our running example, before inserting a node to Harris's linked list we put the address of the next field of the preceding node in the *original parent field* of the new node.

4.4 NVTraverse Data Structures

In this section we show how to apply flush and fence instructions to any traversal data structure to create an efficient and provably correct durably linearizable version of it. These flush and fence instructions can be applied *automatically.*

At a high-level, no persisting is done during the traverse method, whereas, in the critical method, every field accessed must be persisted before the next externally visible instruction is executed. Furthermore, we add another phase between the traverse and critical methods, in which we ensure that the nodes returned by the traverse method are persisted.

4.4.1 Recovery

The recovery phase executes the disconnection function guaranteed by Supplement 1 in Section 4.3.3. No additional action is required.

4.4.2 Before the Critical Method

We now specify the fields that must be persisted before the critical method begins.

Protocol 1. Let $n_1 \dots n_k$ be the nodes that were returned by the traverse method of some operation *op*, where n_1 is the topmost node returned. Before the beginning of the critical method of *op*, the following fields must be persisted.

- The original parent pointer of n_1.
- All fields that the traverse method read in $n_1 \dots n_k$.

We flush these fields in two functions, called *ensureReachable* and *makePersistent*, corresponding to the first and second items, respectively. We briefly describe how we implement each of these functions. Note that these functions are the same for all traversal data structures, and can simply be inserted as black boxes between the traverse and critical methods of a given traversal data structure.

ensureReachable Method

The ensureReachable function's goal is simply to flush one field: the original parent (OP) pointer of the topmost node returned by the traverse method. Note that the *original* parent of a node might not be the *current* parent of that node, since other nodes may have been inserted in between. By Supplement 2 (from Section 4.3.3) the OP field is available in the node. The function simply takes this topmost node and flushes the location indicated by its OP field.

An Optimization for ensureReachable

While the proposed original parent mechanism is simple, it can also be costly, since it requires an extra word on each node, and may also delay garbage collection. We therefore present an alternative solution. For the common case where the insert operation always connects a single node to the structure, ensureReachable may simply flush the *current* parent of its input node. For this, the traversal phase must return the *current* parent of the first node returned from the traversal.

This method can also be used if the insert operation links at most $k > 1$ nodes to the structure simultaneously, but becomes less efficient; the traversal must return the last k nodes on the traversal path before the first node that the traversal procedure returns. These nodes are then flushed by ensureReachable. In Section 4.5, we prove this approach correct. We summarize this in the following lemma.

Lemma 4.4.1. *In an NVTraverse data structure implementation in which the deepest tree ever atomically inserted is of depth k, the ensureReachable(n) method can be implemented as follows.*

- *If n has an OP field, flush the location in this field.*

- *Otherwise, flush a path of length k back from n.*

To prove this lemma, we start by proving the following:

Lemma 4.4.2. *Let n be a node and n_d be some descendant of n. Furthermore, let $c_{n,d}$ be the child pointer of n the points to the subtree in which n_d is. If n is deleted from the data structure at time t and n_d remains in the data structure at time t, then $c_{n,d}$ is flushed before time t.*

Proof We first show that there must have been some high-level operation that read an ancestor of n_d at or below n, and made a change at or above n. Let n be a node that is deleted from the data structure at time t by the high-level operation op_{del}. Consider child $c_{n,d}$ of n, and let T be the subtree rooted at $c_{n,d}$, such that T is not completely deleted at time t. Let n' be the topmost non-deleted node in T at time t. Note that for n' to remain undeleted in the data structure while n is deleted, the deletion *must* occur by swinging a pointer from some ancestor of n to some (new) node that points to an ancestor of n' (or n' itself). This is because of the tree property (Property 2); n' necessarily has only one parent, and for it to remain connected while its parent is removed, a node that is connected to the data structure must point to n' immediately after the deletion occurs. Therefore, op_{del} mush have known n''s or n''s ancestor's address. However, this info is stored at or above n', and in particular at or below n. According to Property 3, the only information an operation has is its parameters, which contain only one entry point to the data structure. Therefore, in order to know n''s address (or its ancestor's address), either op_{del} read it from below n itself, or some other operation wrote it above n. Consider op was the first operation that wrote n''s address somewhere above n. Then op must have read it below n, proving our claim.

Therefore, op must have read the child pointer of n that points to n''s subtree, because this is the only way it could have read n''s (or its ancestor) address and store it above n. Recall that by the way the Traversal data structures operations work, for op or op_{del} to read something below n and subsequently write above n, there are only two possibilities:

1. The read below n, including $c_{n,d}$ was during op's or op_{del}'s traverse method, but below the its return point. After reading $c_{n,d}$, those operations went on to make a change at or above n. Thus, op's or op_{del}'s traverse method must have returned a node at or above n.

2. The read below n, including $c_{n,d}$ was during the critical method.

Thus, in both these cases, $c_{n,d}$ must have been flushed; either in the makePersistent method, or during the critical method. ■

We now continue with proving Lemma 4.4.1.

Proof Assume by contradiction that the lemma does not hold. That is, there is some node n returned by some traverse method such that, after the execution of ensureReachable with the implementation specified by the lemma, n's original parent pointer has not been flushed. Let n's original parent *node* be $origNode$, and let $origPtr$ be the child field of $origNode$ on which the instruction that atomically inserted n into the data structure occurred.

If n had an originalParent field, then by the first option of the ensureReachable method implementation, n's originalParent field would have pointed to $origPtr$, and $origPtr$ would have been flushed. Contradiction.

Otherwise, if n does not have an originalParent field, the ensureReachable method would have flushed a path of length k up from n. Since we assumed that n's original parent pointer, i.e., $origPtr$, was not flushed by the end of the ensureReachable method, this must mean that $origNode$ was not in one of these k nodes. However, recall that by the definition of k, n could not have been inserted at a distance more than k away from its original parent node. Therefore, at least one of the following must have happened: (1) there was at least one insert operation at some node between n and $origNode$ since n was inserted, or (2) $origNode$ was deleted from the data structure before this ensureReachable method was executed. We handle these two cases separately.

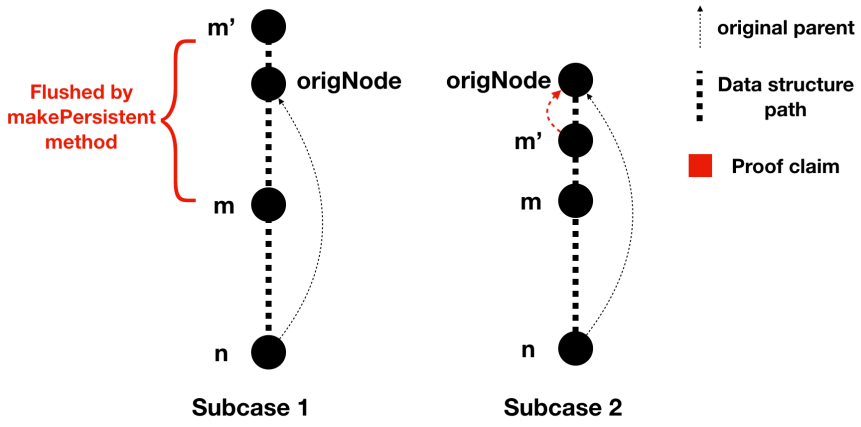
Case 1. There was an insert operation somewhere between n and $origNode$, but $origNode$ was not removed from the data structure. Let m be the node at which the first such insertion occurred. The insertion at m must have been a part of some high-level operation op, whose traverse method returned either m or a node above m. Let m' be the node returned by the traverse method of op. Consider two subcases, gudied by Figure 4.2:

Subcase 1.1. If m' is above $origNode$, then op must have traversed the only path between m' and m and flushed all nodes in between (either inside its traverse method or inside its critical method). Thus, since by assumption $origNode$ was not deleted from the data structure before this occurred and was on the path between m' and m (Property 2), $origNode$ would have been flushed by op before the insertion at m occurred, and therefore also before the end of the ensureReachable method in question. Contradiction.

Subcase 1.2. If m' is below $origNode$ (and above n), then note that since m is the first node on which the insertion between n and its original parent occurs, m' must have been inserted in the same batch as n, and thus m''s original parent must also be $origNode$, and its original parent pointer also $origPtr$. Note that ensureReachable must be run starting on m' before the insertion at m can occur. Note also that by the definition of k as the maximum batch depth, m' was inserted at a distance smaller than k from $origNode$, and at the time that ensureReachable is run on m', it is still at the same distance from $origNode$. This is because of the assumption that op is the first insertion between n and $origNode$. Thus, regardless of whether m' has an originalParent field or not, the ensureReachable method of op flushed $origPtr$. Contradiction.

Case 2. *origNode* was deleted from the data structure. Then by Lemma 4.4.2, since n remains in the data structure after the deletion of *origNode*, *origPtr* must have been flushed before *origNode* was deleted. ■

Figure 4.2: Diagram for Case 1 of Lemma 4.4.1



makePersistent Method

The purpose of makePersistent is to persist all fields read by the traversal on the nodes that it returned. This can be done by simply flushing these fields, and then executing a single fence instruction (which also ensures the completion of the flush of ensureReachable).

4.4.3 During the Critical Method

Simply put, all fields accessed by some process p during the critical method must be persisted before the next externally visible instruction is executed by p. However, this can be relaxed for fields that cannot be accessed by any process other than p. Intuitively, the idea is that p must ensure, before executing an instruction e that may affect other processes, that the values that p relied on to determine e's parameters must be persisted. To achieve this, we use the following rules.

Protocol 2. In any critical method, the following flush and fence instructions must be injected.

- Flush after every read of a shared variable.
- Flush after every write/CAS instruction.
- Fence before every write/CAS on a shared variable.
- Fence before every return statements.

Figure 4.3: Operation in an NVTraverse data structure

```
1  T  operation (Node root, T'  input) {
2    while (true) {
3      Node entry = findEntry (root, input);
4      List<Node> nodes = traverse (entry, input);
5      ensureReachable (nodes.first());
6      makePersistent (nodes);
7      bool restart, T val = critical (nodes, input);
8      if (!restart)
9        return val;
10   }
11 }
```

Note that this means that local work requires less flush and fence instructions than shared work. In particular, when initializing a node, a process executes flushes after initializing each field, but only needs to fence once before atomically inserting the new node into the data structure. Furthermore, there is no need to flush after reading an immutable field.

4.5 Correctness

We say that the algorithm resulting from applying Protocol 1, Protocol 2 and the specified recovery procedure to a traversal data structure form an *NVTraverse data structure*. The algorithm is presented in Figure 4.3. We continue with the proof of the following theorem:

Theorem 4.1. *Every NVTraverse data structure is durably linearizable.*

Intuitively, a data structure is durable linearizability if a crash cannot make the effect of any operation be lost. NVTraverse data structures achieve this by ensuring that every modification to shared memory is persisted before any process uses that modification's value. That is, every value read by p is persisted before p makes its next shared memory modification. The only exception is during the traversal, where a process may read values without persisting them. However, due to the restrictions on how the traverse method may behave, no modification that p does can depend on a value it read (but did not return) during its traversal. Other than persisting all values that may affect p's shared modifications, we also need to make sure that p's writes cannot disappear from the data structure at a later point. This can happen if p wrote on a part of the data structure that might not be *reachable* from the root upon a crash. To prevent this from happening, the ensureReachable function persists the pointer that connects the relevant subtree to the rest of the structure. Thus, the flush and fence instructions we prescribe are *necessary*; removing any of them could

violate the correctness of some NVTraverse data structure. However, hand-tuned data structure implementations could still save on flushes and fences by reasoning more carefully about dependencies in the data.

We now show formally that a NVTraverse data structure is durably linearizable [59]. That is, we show that the flushes described in Section 4.4 are sufficient to ensure that any high-level history H of a NVTraverse data structure with any number of system crashes is linearizable if we remove the crash and recovery events from it.

Consider a legal low-level execution H of a NVTraverse data structure. In the proof, we say that a successful modifying instruction m *was seen* by some instruction i if i was executed on the same location ℓ, and was after m in H. Furthermore, we say that an instruction i is *below (resp. above)* another instruction i' if i is executed on a node n such that the path from the root of the data structure to n passes through the node n' on which i' was executed (resp. n' is on the path from the root to n).

To help with our proof of durability, we first prove that the recovery function can be quite useful to us. More specifically, we show that the recovery function will always execute any pending disconnect instruction. For this purpose, we first introduce the notion of *dependence* between low-level instructions that modify data, and relate that notion to flush ordering.

Definition 4.5.1. Let H be a legal execution of a NVTraverse data structure, and m be a modifying instruction in H that was executed in the high-level operation op_m. We say that m *depends on* another modifying instruction m' in H if one of the following conditions hold.

1. There was a read instruction r in op_m such that r saw m', and the traverse method of op_m returned a node at or above r.

2. m and m' were executed by the same process, and m' is before m in H.

Here the idea is that if a modifying instruction m depends on another modifying instruction m', then we can show that m' must have been flushed before m was executed. Furthermore, we will show that the extended notion of dependency (formally defined below as a *dependency graph*) captures all instructions that might have affected the decision of m's parameters, that is, where and when it is executed, as well as what exactly is written. These two insights together intuitively imply that if we were to lose the effect of a modifying instruction that was not yet flushed, we would be able to recreate that instruction by observing the state of the data structure.

Definition 4.5.2. The *dependency graph* of an execution H of a traversal data structure is a directed graph in which modifying low-level instructions are vertices, and there is a directed edge from m to m' if m depends on m' in H. For a modifying operation m in H, the *dependency graph of* m is the subgraph reachable from m in the dependency graph of H.

Claim 4.5.3. *Let m be a modifying instruction in a history H of a traversal data structure. Then all instructions in m's dependency graph, other than m itself, were flushed in H before m was executed.*

Proof Consider a BFS tree T of m's dependency graph. Note that m is the root of T. The proof is by induction on the number of levels in T. Let op_m be the high-level operation that executed m.

BASE: If T is a singleton, then there are no modifying instructions in T other than m itself, so the claim is vacuously true.

STEP: Assume that for any modifying instruction m' for which the depth of the BFS tree of its dependency graph is at most $k-1$, all instructions in m''s dependency graph were flushed before m' was executed. Then consider a modifying instruction m whose dependency graph is of depth k.

Let T be a BFS tree of m's dependency graph. The depth of the dependency graphs of the children of m in the tree is at most $k-1$. Let c be a child of m in T, and let the subtree of T rooted at c be T_c. By the induction hypothesis, all nodes in T_c are flushed before c is executed, other than c itself. Therefore we only need to consider, for each child c of m, whether c is flushed before m was executed in H. So, first note that m is executed after c is executed. Furthermore, note that by the definition of T, m depends on c. By the definition of dependence, c was either (1) seen by a read instruction in op_m that was at or below the return point of the traverse method of op_m, or (2) executed by the same process as m, before m. In both these cases, c must have been flushed before executing m; in the first case, op_m would have flushed c either during the makePersistent method, or in the critical section, and in the second case, p must have flushed c before proceeding to its next instruction in the critical section. ■

To show that the recovery can execute all pending deletion instructions, our goal is to show that if a deletion instruction is pending, then the state of the data structure after a crash would identify this deletion instruction. Recall that by Property 5, for every given set of contiguous marked nodes S, and an unmarked $Parent(S)$, there is exactly one legal disconnection instruction that can be executed on $Parent(S)$. Furthermore, once S enters such a state, no node in S can change. We therefore begin by showing that before a deletion instruction can be executed, all modifications of nodes that mark S must be flushed.

Claim 4.5.4. *Let d be a deletion instruction and let S be the set of nodes that it deletes. Furthermore, let H be a legal history in which S enters a state in which it is marked, and let M be the set of modifications on nodes that mark these nodes. Then before d is executed in H, all instructions in M have been flushed.*

Proof Assume by contradiction that the claim does not hold. Then there is at least one modifying instruction, $m \in M$, without which S would not be marked, but which

was not flushed before the execution of d in H. Let n be the node that m operated on. Let p be the process that executes d in H. Then by Claim 4.5.3, m cannot be in the dependency graph of d. Moreover, m cannot be in the dependency graph of *any* modification instruction in H, since otherwise it would have been flushed. We now construct a history H' which is legal, and has the following two properties: (1) there is no instruction that executes m on n, and (2) d takes place.

Consider all processes $q_1 \dots q_i$ that attempt to execute m on n in H. Without loss of generality, assume that q_1 executes m in H, and that all other q_j's execute some failed CAS c_j. Then in H', we pause all these processes before they execute this modification attempt. Furthermore, we treat any read instruction r that read m in H as follows: if the operation in which r was executed did not execute any modification instructions or return statements after r, then we pause *op* before executing r. Otherwise, we reschedule any r such that it returns the previous value at that location. Clearly, H' satisfies properties (1) and (2) specified above, since we remove all attempts to execute m on n in H, and do not remove any other modification (in particular, we do not remove d). We now show that H' is legal.

Since by assumption, m was not flushed in H, no process in $q_1 \dots q_i$ could have executed any modification instruction after their attempt to modify n. This is because any such attempt would have been flushed before the next modification or return statement by that process, thereby flushing m. Thus, pausing these processes before their attempted modification of n cannot affect any other process in the execution, except possibly the processes that saw m. We therefore only have to argue that replacing a read that saw m with a read that saw the previous value at that location does not affect any other instruction in the execution.

Let r be a read instruction that saw m, and let *op* be the operation that executed r. Recall that r must have either (1) been during the traverse method of *op*, and the traverse returned somewhere *below* m, or (2) not been followed by any modification or return statement in *op*. This is because otherwise, m would have been flushed by *op*. Case (2) is easily treated since by the definition of H', *op* is paused before executing r, and since it did not modify anything else, it cannot affect any other process. So consider a read instruction r that saw m in the traverse method of *op*, and the traverse returned below m. By Item 2 and Item 4 of Property 4, m could not have made *op*'s traversal stop, since then the traversal would have returned at or above m. By Item 3 of Property 4, m could not have affected the direction in which the traversal proceeded, since that decision is made only using immutable fields. Finally, recall that since m was executed on n, m must not have been an insertion or a deletion or a node, and must only have modified the values of n. By Item 5 of Property 4, if the traverse returned below m, a traverse that had not seen m could not have stopped at n, meaning that m could not have been the reason for *op* to continue its traversal. Therefore, reading the previous value instead of m has no effect on the traversal of *op*, and therefore no effect on the rest of *op* either.

We are now ready to show that the recovery function can execute disconnect instruction if we lose them upon a crash.

Lemma 4.5.5. *After a crash event, the recovery function of a NVTraverse data structure executes all disconnect instructions that were pending at the time of the crash.*

Proof Let H be a legal history of a NVTraverse data structure algorithm A, in which m is a modifying instruction that disconnected the set of nodes S from the data structure. Let p be the process that executed m. By Item 2 of Property 5, there is a set of contiguous nodes S, such that m is legal on $Parent(S)$ only if S was marked. So, consider the set of modifying instructions, M, that marked S. By Claim 4.5.4, all instructions in M were flushed in H before m was executed.

Note that by Property 5, once S is marked, all nodes in S stay in their state. Furthermore, note that by Property 2, each node in a traversal data structure only has one incoming edge. Therefore, there is exactly one pointer that must be swung in order to atomically disconnect S; this is a pointer that resides in node $Parent(S)$, and thus m must be a modification on this pointer. By Claim 4.5.4, at the time that m occurs in H, all the marks has been flushed. Note also that since m was still pending at the time of the crash, all the nodes must still have been marked since no node in S could have changed (by Property 5). Moreover, if any external mark was reverted, by Item 3 of Property 5, it must have been done after executing m, failing to execute m because another thread has done it earlier, or reading the location on which m was executed to disconnect S. All these operations must have been made in the critical section, since updates are done only in the critical section. In addition, reading the location on which m was executed to disconnect S, which leads to reverting the external marks must also been done in the critical section due to Property 4. Therefore, in this case, it means that m was flushed too, contradicting the assumption that m was pending. Therefore, the recovery function finds S marked, and thus by Supplement 1, it executes m. ■

Lemma 4.5.6. *Let H_r be a low-level history of a NVTraverse data structure with exactly one crash event at the end of the history, followed by a recovery. There exists a low-level history H'_r (without crashes or flushes) such that:*

1. *H'_r is a legal execution of the same data structure*
2. *H_r is equivalent to H'_r*
3. *H'_r contains exactly all the flushed modifying instructions in H_r in the same order. H'_r does not contain any pending modifications.*

Proof We prove the lemma constructively; given history H, we show how to construct a history H' that satisfies the necessary properties. We do so iteratively by considering one pending modification at a time, starting with the last such modification in H and working backwards.

Let w_1 be the last pending modifying instruction in H, and let p_1 be the process that executed it, and ℓ be the location on which it was executed. First note that by Lemma 4.5.5, if w_1 disconnected a set of nodes, S, from the data structure, then the recovery function can redo w_1 at any order by Item 4 after a crash, thus w_1 is not pending. We therefore leave w_1 in H' in this case.

We now consider several cases, depending on what operations follow w_1. Intuitively, in each case we either show that we can remove w_1 from the history without harming its legality, or that w_1 must have already been flushed, contradicting the assumption that it is pending.

Case 1. Suppose w_1 was seen by no low-level instruction. Then we can remove w_1 from H.

Case 2. Suppose w_1 was seen by some modification m_2 by process p. If m_2 was flushed (i.e., it is not pending), then w_1 was flushed as well, which is a contradiction. If m_2 was not flushed then w_1 was not the last pending modification, which contradicts the choice of w_1.

Case 3. Suppose w_1 was seen by some read instruction. Let P be the set of processes that had read instructions that saw w_1 in H. For each such process p, let r be the read that saw w_1. Consider the following subcases.

1. There is no modifying instruction or return statement by p after r in the same high-level operation attempt in H. Then we can remove r and all subsequent instructions of the same operation attempt by p. If p has subsequent operations or repetitions of this operation, in H' we let a new process, p' execute these. This leads to a legal execution history since no other process could have been affected in H by p's reads, and since no information is passed between operation attempts, and thus the new process p' can behave exactly as p would have as p' can get the same arguments as p in its following attempt. This is equivalent to a history in which p stalls right before executing r.

2. Suppose there is some modification m or return statement R by process p after r, but r was either (1) inside a critical method, or (2) inside a traverse method, but below the node that this traverse method returned. In both cases, before p executes m or R, it must flush the location at which r executed, and therefore, p flushes w_1 as well. This contradicts the assumption that w_1 is pending.

3. Suppose there is a modification, m, or return statement, R, by process p after r in the same operation attempt as r, and r was in a traverse method that returned a node below r. Note that the modification/return statement must have happened inside the critical method, since by the definition of traversal data structure, all modifying instructions and return statements are in the critical method. Recall that w_1 could not have disconnected any nodes, since we've already covered this case. We thus consider two cases: (1) w_1 inserted some set of new nodes S, but

did not disconnect any nodes, or (2) it neither inserted nor disconnected any nodes.

Case 1. w_1 inserted a set S of new nodes into the data structure. Then consider two subcases, depending on the location of the node n returned by the traverse method of p in which r was executed:

Subcase 1. $n \in S$. That is, p's traverse method in which it read r returned one of the new nodes that w_1 inserted. Then recall that between every pair of traverse and critical methods in the same operation, we run a ensureReachable function that ensures that the original parent of the node returned by the traverse is flushed. Note that in this case, n's orignal parent pointer location is ℓ (where w_1 occurred). Therefore, before the critical method of p's operation begins, ℓ must have been flushed, and thus w_1 is no longer pending. Contradiction.

Subcase 2. $n \notin S$. That is, the node returned by p's traverse method is not part of the set of new nodes added by w_1.

Let n_1 be the node that w_1 operates on and n_2 be the node pointed to by the previous value of w_1. We first claim that the traversal in H must have passed through n_2. We know that the traversal passes through n_1 and it does not stop on any node in S. Therefore it must eventually reach a node that is not in S. Let n_3 be the first such node. If n_3 was not in the data structure immediately before w_1 then it must have been inserted by a pointer swing on some node in the set S. However due to the flushes we add, all the nodes in S have to be reachable by traversing persistent memory before the pointer swing can occur. This would contradict the fact that w_1 is pending. Therefore n_3 must have been in the data structure immediately before w_1. Now suppose for contradiction that $n_3 \neq n_2$. Then that means there are two paths from the root to n_3, which violates Property 2. This is because there was a path to n_3 before w_1, and w_1 adds a new path to n_3. Therefore $n_3 = n_2$ which means the traversal passes through n_2. So we can pause p in H' immediately before its traversal visits n_2 and we can unpause p when p is about to visit n_2 in H. Since p ends up at the same node in both histories and the decision of whether or not to stop and which node to visit next (Property 4) depends only on information inside the current node, after visiting n_2, p would perform the exact same steps in both H and H'. This means p's interpreted history remains the same in H and H' and since we only removed some shared memory reads from H, the order of writes in H are maintained.

Case 2. w_1 neither inserted nor deleted any node. Recall that the traverse method's direction decision may not depend on mutable values, and thus, the traverse of p may not depend on the value read by r. Furthermore, by Item 5 of Property 4, since we know that the traverse method returns a node below w_1 in H, the traverse method in H' must stop at a node below w_1, meaning that reading

the previous value instead of w_1 has no affect on the traversal in H'. Thus, we can safely remove w_1 from the history, and replace r by a read instruction that returns the previous value of ℓ. ■

Therefore, for every p that executed a read operation r that saw w_1, we can safely remove w_1, and possibly have to change some of p's subsequent reads, but not any of its subsequent modifications.

To complete the proof, we need to show that the H' that we've constructed satisfies the 3 properties required by the lemma. Firstly, note that, as agrued above, all changes and removals done to H in the construction of H' leave a legal history. In a nutshell, this is due to the fact that no low-level instructions from the critical method are changed at all between H and H' other than pending modifications, which are never followed by any other step by the same process. Thus, decisions like whether or not to restart an operation are unaffected. Secondly, note that we never remove or change a return statement, and therefore, we do not remove or change any responses of high-level operations. Thus, the interpreted histories of H and H' on completed operations are the same. Consider the interpreted history of H' in which, after every high-level response which is not already followed by a high level invocation, we place the invocation of the high-level operation that is invoked in H at this location. We arrive at the same interpreted history for both H and H'. Finally, note also that we never change or remove any completed modifying instruction in the above construction of H'.

To complete the proof of Theorem 4.1, let H be a low-level history of a NVTraverse data structure, with one crash event followed by a recovery at the very end of the history. Recall that, after a crash, the state of volatile memory is lost, and the state of persistent memory remains. That is, each memory location's value becomes the value of the most recent flushed write on this location. Note that by Lemma 4.5.6, there exists a legal low-level history H' with no crashes such that the state of memory after H' is the same as the state of memory after H's crash and recovery. Furthermore, H and H' are *equivalent*; their interpreted histories are the same. Thus, future operations cannot 'see' the difference between the two histories. We formalize this intuition to complete the proof.

Proof of Theorem 4.1 Let $H = H_1 \cdot C \cdot R \cdot H_2 \cdot C \cdot R \ldots \cdot H_n$ be a high-level history of a NVTraverse data structure, where each H_i contains no crash events, and m and R correspond to a crash event and an execution of NVTraverse data structure's recovery function, respectively. Furthermore, for any pair of subhistories H_i and H_j, the set of processes in H_i and H_j is disjoint. To show that a NVTraverse data structure D is durable linearizable, we must show that we can remove the crash and recovery events and obtain a linearizable history of D. We show this by induction on the number of crash events $n-1$ in H.

BASE: If there are no crash events in H, then H is a linearizable history of D by its definition.

STEP: Assume that if H has k crash events in it, then we can remove the crash and recovery events from H and obtain a linearizable history of D. Now let the number of crash events in H be $k+1$. Consider the prefix G of H that ends immediately before the k'th crash and recovery event of H. By the induction hypothesis, we remove all crash and recovery events from G and obtain a linearizable history of D that contains no crash events. By lemma 4.5.6, there exists a low-level history G' such that G' is legal and equivalent to G, and the state of volatile memory after G' is the same as the state of persistent memory after the crash and recovery immediately following G. Since G and G' are equivalent, any legal continuation of history G is also a legal continuation of history G', as long as the set of processes in the continuation is disjoint from that of G and of G'. Thus, we can remove all crash and recovery events in H and get a linearizable history of D. ■

4.6 Example

In Figures 4.4, 4.5, 4.6 and 4.7, we present pseudocode showing Harris's linked-list (HLL) as an NVTraverse data structure. Note that the traverse method ends in the middle of the search function, since the search also executes some physical deletions, which are part of the critical method of a traversal data structure. The traverse method returns the set of nodes for the ensureReachable (using the ensureReachable optimization) and makePersistent functions to flush.

In Figure 4.4, in lines 7-18, we show how every operation is executed. Every operation starts with finding an entry to the core tree structure. In a linked-list, the entry point is the root of the list. Therefore, findEntry returns the root. After that, the traverse function from Figure 4.7 is called. This function returns exactly three nodes. The *right* node, which is the first unmarked node in the list whose key is greater than or equal to k, and *left* node, which is the last unmarked ancestor of *right*. In addition, it returns the *current* parent of *left*, as described in the optimization in Section 4.4.2. In line 12, the current parent node is flushed in order to make sure that the *left* node is reachable from the head, followed by line 13, where makePersistent flushes the *left* and *right* nodes. After that, the critical part is executed; depending on the operation, we go to either *insertCritical* in line 20 in Figure 4.4, *deleteCritical* in line 1 in Figure 4.5, or *findCritical* in line 1 in Figure 4.6.

The critical function of an insert operation, in lines 20-40 in Figure 4.4 starts by deleting marked nodes. This deletion occurs only if the *left* and *right* nodes that were returned from traverse are not adjacent. If *left* and *right* were not adjacent and the deletion from lines 34-54 in Figure 4.7 fails, the insert operation restarts. If the key already exists, the operation returns false (lines 26-27). As the key is an immutable field, we do not flush after reading the key. Note that *deleteMarkedNodes* executes a

fence before returning. Therefore, there is no need to re-execute that fence in lines 23 and 27. Afterwards, a new node is allocated, followed by a flush after write. In line 31 there is a fence before the CAS in line 32 to insert the newly allocated node. This CAS is followed by a flush after CAS and a fence before the return. If the insertion has failed due to concurrent activity, the operation will be re-executed (line 38). The critical functions of the delete and find operations follow the same rules.

The traverse function is presented in Figure 4.7 in lines 1-33. The inner while loop, from line 9 to line 23, traverses the list from the root and tries to find the first node which is unmarked with a key equal to or greater than k. The marked nodes before k are saved in the *nodes* variable. After the right node is inserted to *nodes* in line 25, the *nodes* variable contains the *left* node which was unmarked at the moment it was inserted (in line 14), followed by all the marked nodes until the *right* node (line 25). If the right node is marked by the time line 26 is executed, the traversal restarts. If not, we proceed to line 29 where we insert *left's* parent to the *parent* variable and return both the *parent* and *nodes* variables (to allow us to persist them later on). Note that by the given properties, no modification is ever done in the traverse.

The last function we present here is called *deleteMarkedNodes* (lines 34-54 of Figure 4.7). This function gets the nodes from the traverse as an input and checks whether there are more than two nodes (more than the left and right ones). If it finds more than two nodes, then there is a need to trim all the marked ones by executing a CAS in line 42. The key observation here that the CAS will be successful only if the current *left.next* pointer is still the pointer that was read during the traverse. If this is the case, the marked nodes will be trimmed successfully and the changed field will be flushed afterwards. In line 45 we make sure again that the *right* node is not marked. If the node is marked, or the trimming was unsuccessful (line 54), then the function will return false and the traversal will need to be re-executed. Otherwise it returns true. Before every return, we make sure that a fence is executed. Moreover, in line 46 there is a flush which is done due to the read of the shared variable in line 45.

Some further optimizations can be done, but we omit them from the pseudocode for readability.

4.7 Experimental Evaluation

We implement five traversal data structures: an ordered Harris linked-list [49], two binary search trees (BST) based on the algorithm of Ellen et al. [36] and Natarajan and Mittal [77], a hash table implemented by David et al. [32] based on Harris's linked-list, and a skiplist based on the algorithm of Michael [73]. We compare the performance of the original, non-durable version of the algorithms to four ways of making it durable: our NVTraverse data structure (*Traverse*), Izraelevitz et al. [59]'s construction (*Izraelevitz*), the implementation of David et al. [32] (*Log Free*) and Ramalhete et al.'s implementation for durable transactions [84] (*Onefile*).

Figure 4.4: HLL Persistent Insert

```
1  class Node<T, V>  {
2    T key;            // immutable field
3    V value;
4    Node* next;
5  }
6
7  bool operation (T key) {
8    bool restart, val = true, false;
9    while (restart) {
10     Node* entry = findEntry(root, input);
11     List<Node*> parent, nodes = traverse(root, key);
12     flush (&parent.next); // ensureReachable()
13     makePersistent (nodes);
14     restart, val = opCritical (nodes, key);
15     if (!restart)
16       return val;
17   }
18 }
19
20 bool, bool insertCritical(List<Node*> nodes,T key)
21   bool succDelete = deleteMarkedNodes (nodes);
22   if (succDelete == false) {
23     return true, false;  // retry
24   }
25   Node* left, right = nodes.front(), nodes.back();
26   if (right.key == key) { // no flush - immutable
27     return false , false;   // key exists
28   }
29   Node* newNode = new Node(key, right);
30   flush (newNode);
31   fence // before CAS
32   bool res = CAS(&(left.next), right, newNode);
33   flush (&left.next);
34   fence; // before return
35   if (res) {
36     return false, true; //  node inserted
37   } else {
38     return true, false; // retry
39   }
40 }
```

Figure 4.5: HLL Persistent Delete

```
1  bool, bool deleteCritical (List<Node*> nodes,T key)
2    bool succDelete = deleteMarkedNodes (nodes);
3    if (succDelete == false) {
4      return true, false; // retry
5    }
6    Node* left, right = nodes.front(), nodes.back();
7    if (right.key != key) {
8      return false, false;  // no key
9    }
10   Node* rNext = right.next;
11   flush (&right.next);
12   if (!isMarked(rNext)) {
13     fence; // before CAS
14     bool res = CAS(&(right.next), rNext,mark(rNext));
15     flush (&right.next) ;
16     fence; // before CAS/return
17     if (res) {
18       CAS(&(left.next), right, rNext));
19       flush(&left.next) & fence;
20       return false,true;
21     }
22   }
23   return true, false; // retry
24 }
```

Figure 4.6: HLL Persistent Find

```
1  bool, bool findCritical(List<Node*> nodes, T key)
2    Node* right = nodes.back();
3    fence; // before return
4    if (right.key != key) { // no flush - immutable
5      return false, false;  // no key
6    }
7    return false, true; // key exists
8  }
```

Figure 4.7: HLL Persistent Search

```
List<Node*>,List<Node*> traverse(Node* head, T k)
  List<Node*> parent, nodes;
  Node* leftParent, left, right;
  while (true) {
    leftParent, left, right = head, head, null;
    nodes.clear();
    Node* pred, curr = head, head;
    Node* succ = curr.next();
    while (isMarked(succ) || (curr.key < k)) {
      if (!isMarked (succ)) {
        nodes.clear();
        leftParent = pred;
        left = curr;
        nodes.append (left); // found left node
      } else {
        nodes.append (curr);
      }
      pred = curr;
      curr = succ;
      if (!curr)
        break;
      succ = curr.next();
    }
    right = curr; // found right node
    nodes.append (right);
    if (right && isMarked (right.next)) {
      continue;
    } else {
      parent.append (leftParent);
      return parent, nodes;
    }
  }
}
bool deleteMarkedNodes(List <Node*> nodes) {
  if (nodes.size() == 2) {
    fence; // before return
    return false;
  }
  Node* left, right = nodes.front(),nodes.back();
  Node* leftNext = nodes[1];
  fence; //before CAS
  bool res =  CAS(&(left.next), leftNext, right);
  flush (&left.next);
  if (res) {
    if (right && isMarked(right.next)) {
      flush(&right.next) & fence; // before return
      return false;
    }
    fence; // before return
    return true;
  }
  fence; // before return
  return false;
}
```

4.7.1 Setup

We run our experiments on two machines; one with two Xeon Gold 6252 processors (24 cores, 3.7GHz max frequency, 33MB L3 cache, with 2-way hyperthreading), and the other with 64-cores, featuring 4 AMD Opteron(TM) 6376 2.3GHz processors, each with 16 cores.

The first machine has $375GB$ of DRAM and 3TB of NVRAM (Intel Optane DC memory), organized as $12\times$ 256GB DIMMS (6 per processor). The processors are based on the new Cascade Lake SP microarchitecture, which supports the `clwb` instruction for flushing cache lines. We use the `sfence` instruction for fences. We use *libvmmalloc* from the PMDK library to place all dynamically allocated objects in NVRAM, which is configured to app-direct mode. All other objects are stored in RAM. The operating system is Fedora 27 (Server Edition), and the code was written in `C++` and compiled using `g++` (GCC) version 7.3.1. The second machine has 128GB RAM, an L1 cache of 16KB per core, an L2 cache of 2MB for every two cores, and an L3 cache of 6MB per half a processor (8 cores). The operating system is Ubuntu 14.04 (kernel version 3.16.0). `clwb` is not supported, so we used the synchronized `clflush` instruction instead. The code was written in `C++` 11 and compiled using `g++` version 8.3.0 with -O3.

We found that David et al. [32]'s code could not run on this NVRAM architecture for executions using more than 3 threads, so we only show comparisons to David et al's log-free data structures on the AMD machine.

On the NVRAM machine, we avoid crossing NUMA-node boundaries, since unexpected effects have been observed when allocating across NUMA nodes on the NVRAM . Hyperthreading is used for experiments with more than 24 threads on the NVRAM machine. No hyperthreading is used on the DRAM machine. All experiments were run for 5 seconds and an average of 10 runs is reported.

On all the data structures, we use a uniform random key from a range $0, .., r-1$. We start by prefilling the data structure with $r/2$ keys. Keys and values are both 8 bytes. Unless indicated otherwise, all experiments use an insert-delete-lookup percentage of $10-10-80$. For all data structures, we measured different read distributions, covering workloads A, B and C of the standard YCSB [29].

For volatile data structures, memory management was handled with ssmem [31], that has an epoch-based garbage collection and an object-based memory allocator. Allocators are thread-local, causing threads to communicate very little. For the durable versions, we used a durable variant of the same memory management scheme [95].

4.7.2 Results on NVRAM

We begin our evaluation by examining the performance of various data structure implementations on the NVRAM Intel machine. We first examine the NVTraverse version of Harris's linked-list [49].

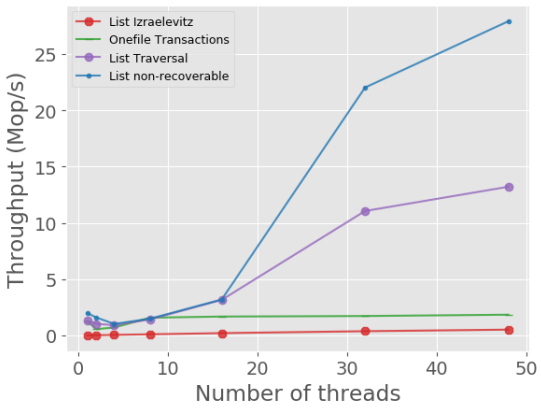
List Scalability

We test the scalability as the number of threads increases, showing the results in Figure 4.8 (a). We initialize the list to have 512 keys, and insert and delete keys within a range of 1024. We note that while the non-durable version of the list outperforms the NVTraverse data structure by 2.1×, the latter outperforms Izraelevitz et al. [59]'s construction by 25.4× and OneFile by 7.3× on 48 threads. While OneFile performs better than Izraelevitz et al. [59]'s construction, they scale similarly. The dramatic differences between NVTraverse data structures and the other approaches hold true throughout all of the experiments that we've tried, highlighting the significant advantage of our approach.

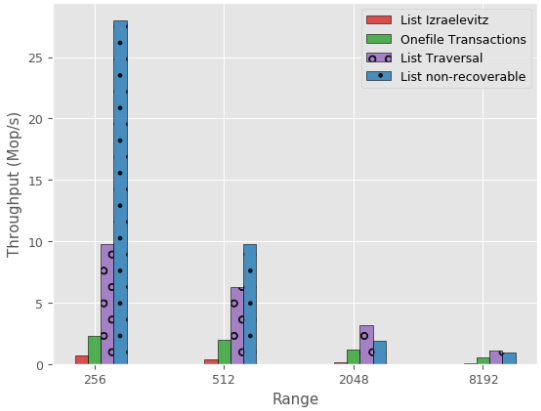
The non-durable version and the NVTraverse data structure have a similar throughput up to 16 threads. However, the non-durable version scales better than the NVTraverse data structure. Note that as the thread count increases, there are more flushes in the NVTraverse data structure, since each thread flushes a constant number of nodes. Each flush invalidates that cache line, meaning that as the number of threads increases, and cache misses become more likely.

List Size

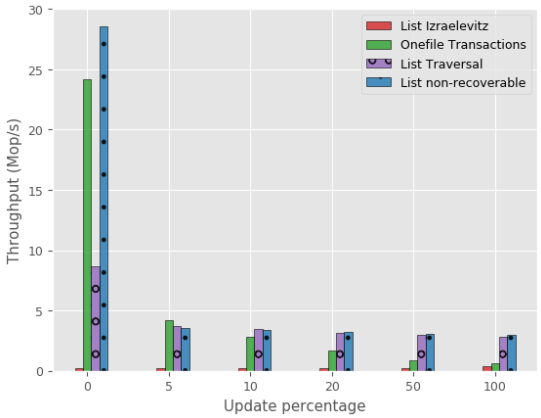
Next we test how lists of different sizes affect performance. We expect that the size of the data structure may have a significant effect because a larger data structure means operations spend a larger fraction of their time traversing. The results are shown in Figure 4.8 (b). The first thing to note is that the original, non-durable version of the list greatly outperforms the NVTraverse data structure version for smaller lists. The non-durable version is better by 2.9× for a short list of 128 nodes and by 1.5× for the size of 256 nodes. However, the difference becomes less pronounced, and even inverts, as the list grows. This phenomenon can be explained by considering the traversal phase of operations on the list, and their role in determining the performance of a given implementation. Recall that the NVTraverse data structure construction only executes a constant number of flushes and fences per traversal, and the non-durable version never executes flushes or fences. As the size of the data structures increases, the cost of the traversal outweighs the cost of persistence. For the original list, the traversal is the primary source of delay, so the effect of increasing the traversal length in this implementation is starker than the effect of the same phenomenon in the durable version, which also spends significant time persisting. For the durable competitors, we observe the same trend as we saw in the list scalability test. The NVTraverse data structure construction outperforms Izraelevitz et al. [59]'s construction by 13.5×-39.6× and OneFile by 4.25×-2.2× on a range of 256-8192 respectively.



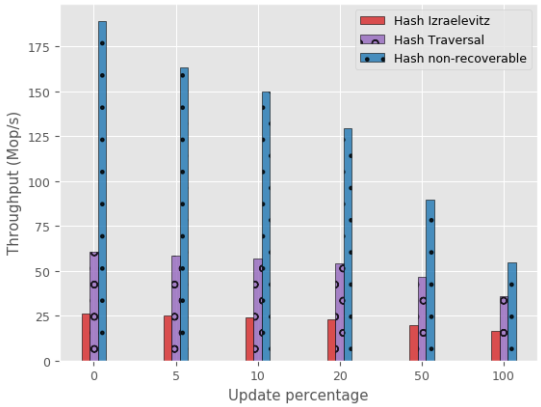
(a) Linked-List, varying threads, 80% lookups, 500 nodes



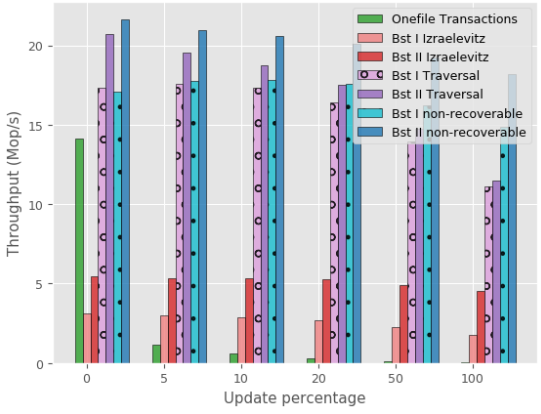
(b) Linked-List, varying size, 16 threads, 80% lookups.



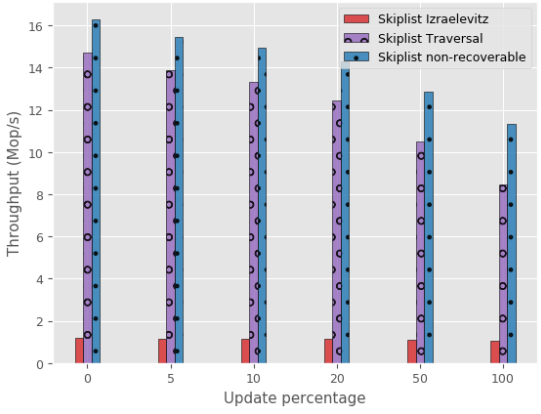
(c) Linked-List, varying update percentage, 16 threads, 500 nodes



(d) Hash-Table, varying update percentage, 16 threads, 1M nodes



(e) BST, varying update percentage, 16 threads, 1M nodes



(f) Skip-List, varying update percentage, 16 threads, 1M nodes

Figure 4.8: NVRAM throughput results.

List Update Percentage

We now consider the effect of varying update ratios (Figure 4.8 (c)). These tests were run on a relatively small list (500 nodes), so the non-durable version outperforms the

NVTraverse data structure. Interestingly, the non-durable list's throughput sharply drops between 0% and 5% updates whereas the NVTraverse data structure version stays relatively stable across all update percentages. Since the list is less than 12Kb, it is small enough to fit in L1 cache, so there are virtually no cache misses on read-only workloads on the non-durable list. The NVTraverse data structure version still experiences cache misses because lookup operations perform `clwb`, invalidating the cache line. In our experience, it seems that in the current architecture, `clwb` and `clflushopt` yield the same throughput. In future architectures, we believe `clwb` will no longer invalidate cache lines and will perform better. OneFile does extremely well in read-only workloads (for which it is optimized).

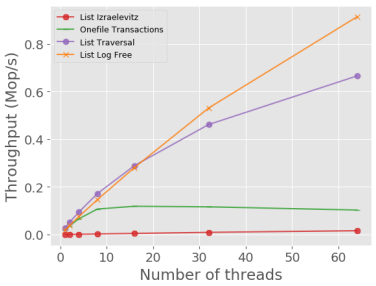
BST, Hash Table and Skiplist

We study how the Hash Table, BSTs and Skiplist behave under different YCSB-like workloads. The results are shown in Figures 4.8 (d), (e), and (f) respectively. We only show OneFile's performance in the BST, since the patterns seen on OneFile are similar in all cases, and similar to the list. We implemented two versions of the BST; one based on Natarajan and Mittal [77]'s tree, and the other on Ellen et al. [36]'s tree. We saw that the amount of flushes and fences Ellen et al. [36]'s tree executes is *less* than in Natarajan and Mittal [77]. However, Ellen et al. [36] performs worse than Natarajan and Mittal [77] in their volatile version, and the gap remains in the durable version.

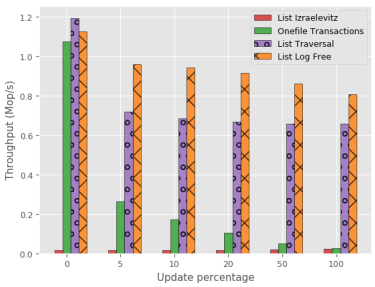
We see that in the hash table, the non-recoverable version degrades twice as fast as the NVTraverse data structure as the number of updates grows. This is because allocating and writing nodes is more expensive than just reading. However, in the NVTraverse data structure, these costs do not form a bottleneck, because of the additional flush and fence instructions. Interestingly, the skiplist and BSTs do not exhibit this behavior; in fact, the NVTraverse data structure version degrades faster than the non-recoverable version as the update percentage increases. This can happen due to the fact that as the number of updates increases, the likelihood of failed CASes increases, which is more meaningful than in the hash table. For the NVTraverse data structure, this means executing extra flush and fence instructions, which slows it down more in comparison to the non-recoverable version.

4.7.3 Results on DRAM

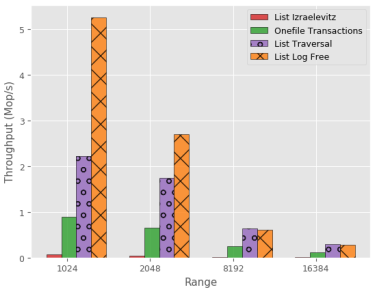
We ran experiments on a machine with classic DRAM to compare with the algorithms of David et al. [32]. We ran David et al. [32]'s algorithms in the *link-and-persist* mode. Link-and-persist, suggested by David et al. [32] and Wang et al. [90], is an optimization that allows avoiding flushing clean cache lines by tagging flushed words, but is not completely general, so we did not apply it to the NVTraverse data structure constructions. David et al. [32] actually present two optimizations in their paper, but the second one they present, called the link-cache, does not provide durable linearizability [59]. At



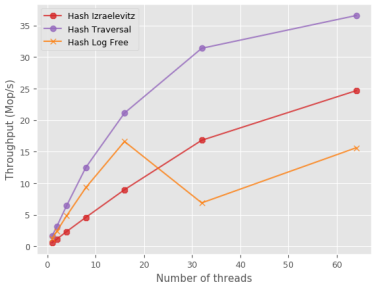
(g) Linked-List, varying threads, 80% lookups, 8000 nodes



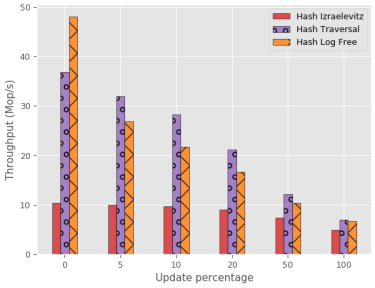
(h) Linked-List, varying update percentage, 64 threads, 8000 nodes



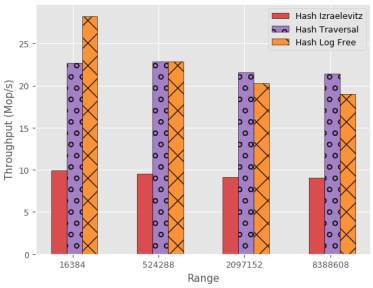
(i) Linked-List, varying size, 16 threads, 80% lookups



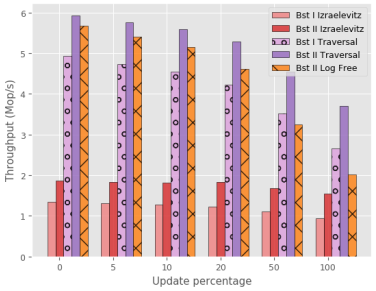
(j) Hash-Table, varying threads, 80% lookups, 8M nodes



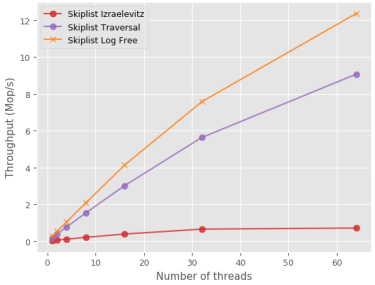
(k) Hash-Table, varying update percentage, 16 threads, 8M nodes



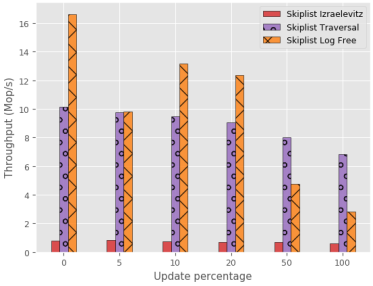
(l) Hash-Table, varying size, 16 threads, 80% lookups



(m) BST, varying update percentage, 16 threads, 8M nodes



(n) Skip-List, varying threads, 80% lookups, 8M nodes



(n) Skip-List, varying update percentage, 64 threads, 8M nodes

Figure 4.9: DRAM throughput results.

least one optimization must be selected.

List

We ran with an initial size of 8192 nodes in the list with a range of 16,384 keys, varying the thread count. The results are shown in Figure 4.9 (g). We notice that the linked-list algorithm of David et al. [32] outperforms ours by 15%-50%, from 32 to 64 thread counts; this is due to the fact that the link-and-persist technique reduces the number of flushes. With more threads, it is more likely that two threads get the same key, meaning that only one of them will have to flush. On the other hand, the NVTraverse data structure outperforms David et al. [32] by 40%-16%, for thread counts of 1-8. We believe this happens because of the same optimization. In link-and-persist, there is an extra CAS for each flush executed (to tag the word), and on a lower thread count, this

optimization is less beneficial. So, our automatic construction does have a cost, but this cost is much smaller than Izraelevitz et al. [59]'s (by up to 56×).

We now test various update percentages, with 64 threads and same list size as above (Figure 4.9 (h)). As previously noted, the linked-list algorithm of David et al. [32] outperforms the NVTraverse data structure by at most 1.37x at 20% updates, as the flush-and-persist technique avoids some flushes. When there are read-only operations, our list is faster by 1.7x, again, as David et al. [32] executes some CASes to avoid extra flushes. We see the same trend in Figure 4.9 (i), that shows 16 threads with varying key ranges. The bigger the list, the smaller the advantage of link-and-persist. OneFile [84] performs worse than our construction, as expected, by 1.1x-25x for 0%-100% update percentages respectively (Figure 4.9 (h)).

Hash table

We observe the opposite trend in the hash table. In Figure 4.9 (j), (k) and (l) we can see the scalability, various update percentages and ranges of keys respectively. The first two are filled with 8M nodes. For a fair comparison, due to the anomaly that we observe in Figure 4.9 (j) on 32 threads, the update percentages are shown on 16 threads. With 0% updates, the algorithm of David et al. [32] outperforms the NVTraverse data structure. This is due to the hash function; David et al. [32] use a bit-mask, assuming the table size is a power of 2. This is faster than the more general `modulo` function that we use. In all the other comparisons, the NVTraverse data structure outperforms David et al. [32] by up to 30% on 16 threads and 230% on 64 threads. In a hash table with 8M nodes, the contention on every bucket is low, so the price of the link-and-persist outweighs the benefit. This is clearer in Figure 4.9 (l), which shows various ranges of keys with 16 threads and 20% updates.

Binary Search Tree

Figure 4.9 (m) shows the results of various update percentages of the BST with 8M initial size and 16 running threads. We compare our two NVTraverse BST implementations with the implementation of David et al. [32], as well as the two BST versions for Izraelevitz et al. [59]'s algorithm. David et al. [32] implements the durable version of Natarajan and Mittal [77] as well. As the contention is low, same as in the hash table, the CASes for marking the flushed nodes downgrades the performance of the same BST implementation by 4%-83% for 0%-100% of updates.

Skiplist

The scalability and varying update percentages of the skiplist are depicted in Figure 4.9 (n) and (o). Figure 4.9 (n) shows the scalability for an initial size of 8M nodes and 20% updates. As it executes one less flush in every search operation, the implementation of David et al. [32] performs better than the NVTraverse data structure in a read

dominated workload. For 64 threads and 20% updates, David et al. [32] is 1.3x better, and it reaches the maximum difference of 1.63x in 0% of updates. However, as seen in Figure 4.9 (o), as the workload becomes more write dominant, the performance degrades; it benefits less from the flush that was saved in the search operation. The NVTraverse data structure gets better throughput by 1.68x and 2.4x on 50% and 100% updates.

4.7.4 Other Architectures

We showed the evaluation of our transformation on two different architectures. We believe that our persistent transformation is hardware-agnostic and relevant to other frameworks that satisfy other memory models. However, the instructions that are used for executing flushes and fences should be adjusted accordingly. For instance, in an ARM architecture, a flush instruction may translated to the *DC CVAP* instruction and a fence instruction may be executed by calling to a full *DSB* instruction [6, 83].

4.8 Related Work

NVRAM has garnered a lot of attention in the last decade, as its byte-addressability and low latency offer an exciting alternative to traditional persistent storage. Several papers addressed implementing data structures for file systems on non-volatile main memory [22, 63, 65, 92, 87, 94].

Recipe [64] provides a principled approach to making some index data structures persistent. While on the surface, their contribution is similar to ours, their original approach does not always yield correct persistent algorithms, even for data structures that fit their prescribed conditions. The ArXiv version of their paper has updated Conditions 1 and 2 to account for this. The new formal conditions require flush and fence instructions for every read and write, similarly to the requirement of Izraelevitz et al. [59]. While they note that in some situations, one can leave out some of these flushes and fences, it is left for the user to do so. Recipe thus exemplifies the difficulty of providing a correct, general and efficient solution, and the importance of having one. In our work, we provide a general and automatic way to reduce the required flush and fence instructions for traversal data structures that yields efficient persistent data structures, and we provide a proof that this transformation is correct.

Friedman et al. [41] presented a hand-tuned implementation of a durable lock-free queue, based on the queue of Michael and Scott [75], and presented informal *guidelines* for converting linearizable data structures into durable ones. Based on these guidelines, David et al. [32] implement several durable data structures. David et al. [32] achieve this by carefully understanding each data structure to find its dependencies, and only intuitively argue about correctness. Our definition of traversal data structures formalizes some dependencies in a large class of algorithms and removes the need for expert understanding of persistence and concurrency.

Other general classes of lock-free algorithms have been defined. Brown et al. [17] defined a general technique for lock-free trees, and Timnat and Petrank [86] defined *normalized* data structures. These classes were defined with different goals in mind and do not aid in finding dependencies that are critical for efficient persistence. Another line of work focuses on formally defining persistency semantics for different architectures [79, 81].

4.9 Conclusions

Recent NVRAM offers the opportunity to make programs resilient to power failures. However this requires persistent memory to be kept in a consistent state. In general this can be very expensive since it can require flushing and fencing between every read or write. This renders caching almost useless. In this chapter we considered a broad class of linearizable concurrent algorithms that spend much of their time traversing a data structure before doing a relatively small update. The goal is to avoid any flushes and fences during the traversal (read-only portion). For a balanced tree, for example, this can mean traversing $O(\log n)$ nodes without flushing, followed by $O(1)$ flushes and fences. We describe conditions under which this is safe. Although the conditions require some formalism, we believe that in practice they are quite natural and true for many if not most concurrent algorithms. We study several algorithms under this framework and experimentally compare their performance to state-of-the-art competitors. We run the experiments on the recently available Intel Optane DC NVRAM. The experiments show a significant performance improvement using our approach, even beating hand-tuned algorithms on many workloads.

Chapter 5

A General Construction: Mirror

5.1 Introduction

NVTraverse [40], presented and discussed in previous chapter, improve performance dramatically, albeit with increased complexity and with the transformation restricted to a subclass of the lock-free data structures. Mirror [42], in contrast, uses a very different transformation and demonstrates significant improvements on current platforms. Mirror is a simple automatic transformation that is capable of converting any linearizable lock-free data structure into a persistent and lock-free durably linearizable [59] data structure.

The first idea underlying the Mirror transformation is to have two copies of the data. At first sight, this may seem costly because each write to the data structure needs to be executed twice, but there are two points that make these two replicas worthwhile. First, only the first write (to the first replica of the data) needs to be persisted (with a flush and a fence), making the second (non-persisted) write a great deal lighter. Second, using the second replica for reading from the data structure eliminates the need to ever persist read data, which is a great advantage for a general transformation.

The second idea is to place the second (volatile) replica in the volatile DRAM. While the first copy of the data is used for persistence and must be placed in the non-volatile memory, the second copy is only used to speed loads from the data structure and is never persisted. Placing the second copy in the volatile DRAM increases the access speed of loads substantially; in current platforms this is by a factor of 3×. This idea improves the performance of the durable data structures obtained by the Mirror construction even further and we get a significant throughput improvement over NVTraverse [40].

Given that many operations on lock-free data structures spend a considerable amount of time traversing nodes, effectively executing loads, the usage of DRAM for the volatile replica significantly increases the throughput in read-dominated workloads, and can yield a non-negligible advantage even on write-intensive workloads due to traversals that precedes the updates. For example, the persistent linked-list data structure created by Mirror outperforms the persistent linked-list created by NVTraverse by a factor

higher than 4× (for a list of size 128, with 8 concurrent threads, and 20% update operations). This factor increases more than 10× for a workload with read-only operations. Furthermore, Mirror outperforms the persistent lock-based data structure Cmap by pmemkv [85] by up to 3.95× for a hash table of size $8M$, with 8 concurrent threads. In fact, the evaluation shows that the data structures created by Mirror sometimes beat even hand-made specific data structures that were designed to execute solely on the non-volatile main memory. Moreover, because of the partial use of volatile DRAM, the persistent data structures created by Mirror can often execute faster than original (non-persistent) data structures that execute on the slower non-volatile memory.

In Section 5.4 we explain the details of the construction and explain why it works correctly. Loosely speaking, all read data is guaranteed to be persisted (in the persistent replica) before it is read (in the volatile replica) and, therefore, there is no need to worry about persisting read data. An invariant of the durable data structure obtained from the Mirror transformation is that the second volatile copy is at most one value behind the first copy. A version number is added to each shared data structure field to keep the modifications well ordered, in an adjacent word, using a double-word-compare-and-swap (DWCAS) on every modification.

To make the transformation easy for the programmer, we built an implementation of the primitives that handle the two copies, i.e., the modified load, store, and CAS operations that should be called by the durable data structure. These are implemented as overloading of these operations using a modified `std::atomic` type. We also built an adequate allocator that can work with the two copies. Given these implementations, applying Mirror to a data structure is simple. It consists of type annotation to replace the usage of `std::atomic` with a `patomic` and replacing the calls to the system allocator with the Mirror's allocator. In addition, the programmer needs to specify the roots of the data structure from which all nodes of the data structure are reachable, and provide a routine that, given the roots, traverses all nodes reachable from the roots. In case of a crash, this traversal is required by the recovery procedure. This can be easily implemented for all existing data structures that we are aware of. Mirror imposes no algorithmic change to the lock-free data structure code in order to make it persistent.

The rest of this chapter is organized as follows. Section 5.2 provides an overview of the Mirror library and the detailed implementation and its correctness appears in Section 5.3. The experimental evaluation for different data structures is presented in Section 5.5. Section 5.6 discusses related work and Section 5.7 concludes.

5.2 The Mirror Library

The *Mirror library* is a general interface that provides durability in an automatic manner. The main concept consists of having two copies of every variable, which allows separation between operations, such as read and write. Not only does this separation enable a significant performance improvement, but the *actual* memory type on

which these operations execute also affects the throughput, e.g., DRAM vs NVMM, as presented in Section 5.5. The interface is simple to use, and eliminates the need to understand all the complexities that emerge when using non-volatile main memory. The library uses all three components of current existing hardware: volatile caches, non-volatile main memory and a volatile DRAM.

5.2.1 Replica Location

Mirror keeps two replicas of each persistent variable. One replica is used for persistence and the other one for reading. This separation allows reading a consistent value only from the fast volatile memory. To ensure persistence, however, writes still have to be done on both the persistent and volatile memories. A careful implementation is needed to guarantee correctness, as shown in Section 5.3.

The first replica must be located in the NVMM, which provides durability. The second replica is located in the volatile main memory, the DRAM, where reads are more efficient. If a volatile main memory is not available, for example, when there is insufficient space for a large database, the second replica from which the reads are executed might be located on the persistent memory as well, which still yields advantages in read-heavy workloads, as presented in Section 5.5. For simplicity, we call the first replica the persistent replica rep_p, and the second replica the volatile replica rep_v.

5.2.2 Interface

The *Mirror* library is simply an extension of the std::atomic library [18] with added support for persistence on non-volatile main memory, by overloading the existing operations. Its API provides the exact API of *std::atomic* from the `C++` standard, and two additional operations for the allocator usage.

compare_exchange_strong (T& expected, T newVal): Compares the current field's value with the expected value and, upon success, replaces the current value with the new one. Otherwise, loads the current object's value into the expected parameter. This is all done atomically on both replicas. When successful, both memory locations will have the new value. If the operation fails, neither of the memory locations will have the new value.

fetch_add (T add): Replaces the current value with the arithmetic's addition result of the current value and the parameter, atomically. This operations always succeeds, and guarantees that both memory locations will have the same value.

load (): Returns the current value atomically.

store (T desired): Stores a new value atomically. The same value is stored on both replicas.

There are two more operations made available to the programmer:

init(): Initializes the persistent and volatile memory regions. This operation needs to be called once at the beginning of the execution, and immediately after each system-

failure. This operation mmaps a persistent and volatile memory region and copies the relevant data from the persistent to the volatile memory.

alloc(T value): A wrapper for allocating an object. This operation needs to be called every time an object is allocated to guarantee it will be adequately allocated on the volatile and persistent regions.

Last, the user needs to provide a tracing operation which is able to trace all data given the persistent roots, similar to previous works [95, 59, 40].

To use this interface, a variable T needs to be converted to *patomic<T>*. An example is shown in Figure 5.1. It shows how to convert an object with all its fields so that it can use this infrastructure. The example converts a node of Harris's linked list [49] into *patomic<>*. It calls the *init()* operation at the beginning of the execution and the allocator's wrapper every time an object is dynamically allocated (instead of the allocator itself), so that the variable is allocated on both replicas.

In Section 5.3, we explain how using this library guarantees atomicity and persistence.

Figure 5.1: Example of using Mirror's Library

```
1 class Node {
2   patomic<unsigned int> key;
3   patomic<T> value;
4   patomic<Node*> next;
5 }
```

5.3 Mirror Underlying Mechanism

If an object needs to be persistent, i.e., survive a crash, every field T within this object needs to be converted to *patomic<T>*. It must use Mirror's allocator wrapper and provide a tracing operation. Moreover, if a data structure is lock-free and linearizable [53], using *patomic* version with all its fields will automatically convert it to be durable linearizable [59]. In what follows we explain the implementation of the Mirror library. This implementation does not require any compiler or operating system modification and it may be added to a new platform by any programmer to obtain the benefits of the Mirror transformation.

5.3.1 Implementation

To maintain consistency, every variable T that is converted to *patomic<T>* has two fields: an *std::atomic* value and an *std::atomic* sequence number, which is correlated to that value. The template class is presented in Figure 5.2.

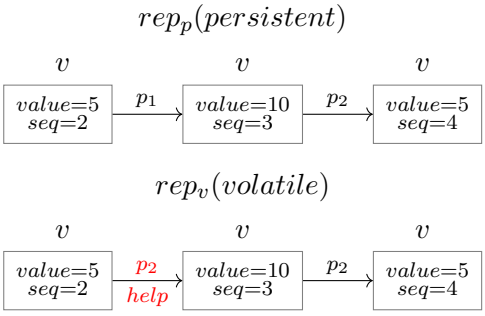
Figure 5.2: Patomic class

```
1 template <typename T>
2 class patomic {
3   atomic<T> value;
4   atomic<uint64_t> sequence;
5 }
```

Sequence Number

The sequence number is crucial for correctness, linearizability in the presence of two object copies. Every variable has an associated sequence number which is increased monotonically with every value change. The sequence number is important because we need to maintain two copies and avoid *ABA* problems. A sequence number, however, is needed only if the field changes so immutable fields, such as immutable keys, do not require it. The scenario shown in Figure 5.3 demonstrates why a value change requires an additional sequence number: Let variable v start with holding the tuple $\{5, 2\}$, which represents a value 5 and a sequence number 2, both on its volatile and persistent replicas. Let process p_1 write the tuple $\{10, 3\}$, where the sequence number is increased by one in the persistent replica and pause before writing it to the volatile replica. Now, let another process p_2 write a new value, $\{5, 4\}$. Before doing it, it needs to make sure that both replicas have the same value. If this is not the case, p_2 *helps* p_1, meaning that it finishes p_1's write to the volatile replica, and only then changes the persistent value and the corresponding volatile value to be $\{5, 4\}$. If the sequence number had not existed, and the first process had continued its execution, it might be able to change the volatile replica to be 10 again, even though the sequence of the values $5 \rightarrow 10 \rightarrow 5 \rightarrow 10$ has not ever occurred. Attaching an increasing sequence number to it, however, guarantees this scenario can never happen. Now, p_1 will not be able to change the value $v = \{5, 4\}$ to be equal to $\{10, 3\}$ and the volatile replica will stay $\{5, 4\}$.

Figure 5.3: A scenario that could cause problems if there was no sequence number



CAS Instruction

As mentioned above, all reads are made on one replica, called rep_v and writes are made on both replicas to maintain consistency. The write is first made on the persistent replica, called rep_p, and then on rep_v. Reads are always wait-free and made on rep_v solely. The pseudo-code for the CAS operation is presented in Figure 5.4. Let v be a variable. v is located on both rep_v and rep_p. We start by checking that the variable v in rep_p, v_{rep_p}, is equal to the value of v in rep_v, v_{rep_v}, both in terms of the sequence number and the value itself. We read v's value and sequence number from the persistent and the volatile replicas in lines 5-15. Reading the value and its sequence number is not atomic, and, therefore, there is a need to assure that they are related. Accordingly, there is a read of the sequence number, followed by a read of the value, followed by another read of the sequence number. Only when the values are equal (after line 28), it is possible to write the new value to rep_p. The new value will contain the new value itself and a sequence number that is increased by 1 compared to the last written sequence number. The write is done with a *double-word-compare-and-swap (DWCAS)* instruction that swaps both the value and the sequence number atomically (line 39). Upon success, there is an attempt to write the same new value on rep_v (line 43). If this succeeds, then the write is finished after writing to the volatile replica. If there is a failure in writing to volatile memory, it means that there was a concurrent attempt to write the same value (or a more updated value) to rep_v. Either way, we can finish the operation. Note that it is not possible that the CAS fails due to an older value of v in the volatile replica, because we previously made sure that the volatile version had the expected value before updating the value on both replicas.

On the other hand, if there is a failure in writing to persistent memory, then we help the thread that succeeded by attempting to write v's value on rep_p to rep_v (line 47). Then we can return false. To maintain consistency, the helping is done only by changing the volatile replica from the value the helping thread read from the volatile replica before its failure, with the value that made its DWCAS operation in the persistent replica in line 39 to fail. The value that made it fail will be located in the before variable (the DWCAS updates this value if it fails). In addition, there is a special case where the concurrent thread might write the expected value to the persistent replica, and the DWCAS in line 39 will fail due to a different sequence number. In a regular CAS, the operation needs to succeed and therefore, we restart the operation in line 46.

The last remaining case is that the value read from rep_v is different from the value read from rep_p. In this case, there is an ongoing concurrent operation executing and we attempt to help it (line 18), by copying the value and sequence number from the persistent replica to the volatile replica. As we first write to the persistent replica and then to the volatile replica, the sequence number in the volatile memory and its correlated value, v_{rep_v}, can be at most off by one, in comparison to the persistent memory.

All other writing operations, e.g., *store*, *fetch_add*, are implemented by calling the *CAS* operation, with slight changes. As they may never fail, they keep calling the *CAS* operation with the relevant parameters until they succeed.

The method as described here does not support data structures that use double-word fields with a wide CAS, as we need to add a version to each field and modify the value and version atomically. In all algorithms with double-word fields that we are aware of, however, these fields contain a unique value for each modification. In fact, most of these algorithms use on of the words for versioning. In such cases, the Mirror construction works well without adding an additional version word and can be applied as is. An ABA problem cannot occur in this case. The Mirror algorithm can be extended to also handle all other algorithms than use double-word CAS operations, but this extension is more involved and not needed in practice.

5.3.2 Persistent Roots

Loads and stores are used to access the non-volatile memory, which is mapped directly into the process address space. To access the NVMM, it is possible to *mmap* a direct access file, making all the data accessible through *persistent roots* [23, 88, 20, 14]. Persistent roots are simply known addresses from the mmaped file, which is located on the NVMM. Making all data accessible from these persistent roots and guaranteeing a consistent state, will assure complete recovery upon a crash. We assume that the NVMM mapping is always done to the same base address. In this way, all the pointers within that memory will remain in a consistent state.

5.3.3 Memory Reclamation

Every object may have two kinds of data: critical data and auxiliary data. Auxiliary data refers to data that may be recovered from the critical one, but critical data must persist for recovery. Therefore, there is a trade-off between run-time overhead and recovery time. On the one hand, we can save all data, both critical and auxiliary, and spend minimal time on recovery, but it incurs a costly overhead on the run-time itself. On the other hand, we may reduce the run-time overhead, and let recovery reconstruct all the auxiliary missing data. Exploiting the fact that there are two replicas of the data may allow us to ensure that the auxiliary data are read/written from the volatile replica, while the critical data are always written to the persistent replica. A natural fit for this will be all the metadata of the allocator. As we maintain two replicas, all allocator metadata may reside on the volatile replica due to the fact that the persistent replica behaves exactly the same, with just an offset. Moreover, having persistent roots allow us to reconstruct everything we need to continue executing upon a crash, without the need to read the metadata. Our technique guarantees that all reachable data from persistent roots is persistent. Therefore, upon a crash, it is easy to distinguish which data can be reclaimed. Moreover, if the reachable data is duplicated (by having two

replicas), there is no need to manage the memory of both replicas. Thus, we manage the memory in the volatile replica, which yields more efficient reads and writes, and upon a crash, we need to traverse the persistent roots, and copy all the reachable data to the volatile memory. This, however, is possible only if a tracing operation is provided, as expected in previous works as well [95, 59, 40, 32]. For the volatile memory reclamation scheme, we use *ssmem*, an epoch based garbage collector (GC), and an object-based memory allocator [31].

Another possibility is to use a more costly technique, which persists only the allocator's core-data on the NVMM. Upon a crash, it re-constructs all the auxiliary data, and executes an offline GC. This offline GC simply traverses the persistent roots and reclaims all the data that is not reachable [19, 14].

Address Translation

To be able to manipulate both replicas, rep_v and rep_p, we assume that both base addresses are always mapped to the same virtual address space. As pointers are managed by the volatile memory, it is easy to translate the volatile address to the persistent address by simply adding the difference between the persistent base address and the volatile base address to the pointer itself. In other words, the delta between the two replicas is used to translate the addresses of the volatile and persistent memories. This technique is simple and the translation is efficient. If mapping to the same virtual address space is not possible after a crash, then another way of implementing the address translation might be to persist both the base address of the persistent memory and the offset between the volatile and persistent addresses from the persistent root so addresses will be calculated by using these offsets.

Init and Allocation

When a program begins to run, we mmap a direct access file to maintain the replica on the NVMM. First, we allocate the roots of the data structure itself in the persistent roots' region. Afterwards, for every allocated location we perform the allocation on the volatile space and then copy the variable with its sequence number to its matching persistent memory location according to the address translator. From that point, the memory is updated in both replicas, but metadata are managed only on the volatile replica.

Our underlying allocator's wrapper operation is responsible for constructing an object which is first constructed on the DRAM. We use the object-based memory allocator provided by David et al. [31], but any allocator can be used. After that, the object is constructed on the NVMM as well, without the metadata that are related to the allocator.

Recovery

After a crash, a recovery operation should be invoked before re-executing the program. As we reauire the data structure to supply a tracing operation, which just traces all the reachable data from a set of roots, by knowing the location of the persistent roots, it is easy to trace all the reachable objects on persistent space and re-allocate them. As memory allocated on the NVMM without any metadata, there is a need to re-allocate the objects on both memories. First we mmap a new file on the NVMM. Once objects are traced, we allocate every node on the volatile and the persistent memories with correlated addresses, and then the previous mmaped file is deleted.

5.4 Correctness

We now describe briefly the correctness of our construction, showing that any linearizable [53] and lock-free data structure that uses our construction is persistent and satisfies durable linearizability [59], in particular.

A data structure is considered durably linearizable if all the operations that completed, survive upon a crash, plus some overlapping ones. The operations that were concurrent with the crash must survive if their effect has impacted other operations. By reading only from the volatile region, we make sure that any value read was already persisted, meaning that if that process executed a durable change, it already read the persisted values that influenced that change. This occurs because the Mirror's write persists the value right before writing it to rep_v either by the writer itself or by a helping one.

Theorem 5.1. *A linearizable and lock-free data structure that uses the Mirror construction provides a durably linearizable data structure.*

To prove Theorem 5.1, we first need to claim that our implementation for the load and store instructions yields the expected behavior. As mentioned above, a load reads a copy from the volatile memory, rep_v, and a store first writes the value to the persistent memory, rep_p and only then to rep_v. As all the store operations are implemented with the help of our Mirror' CAS implementation, presented in Figure 5.4, we describe only the relation between loads and CASes.

We start by describing the linearization points which are the actual moments where these operations take effect. Usually, these instructions are atomic, but in our implementation they are more complex. The load returns the value that is located in the volatile memory, atomically, according to Figure 5.5, even though the variable actually consists of an *atomic<T>* value and an *atomic<int64_t>* sequence number. Therefore, the linearization point of this operation is the actual load of the value itself. According to Figure 5.4, the linearization point of a successful CAS operation is the moment when the new value and the corresponding sequence number is written by a *DWCAS*

Figure 5.4: Patomic Compare_exchange_strong Implementation

```
1  template<typename T>
2  bool patomic<T>::compare_exchange_strong (T& expected, T newVal
       ) {
3    patomic<T>* rep_p_addr = REP_V_2_REP_P(this);
4    while (true) {
5      rep_p_seq = rep_p_addr->seq; // Read rep_p
6      rep_p_val = rep_p_addr->val;
7      rep_p_seq_again = rep_p_addr->seq;
8
9      rep_v_seq = this->seq;      // Read rep_v
10     rep_v_val = this->val;
11     rep_v_seq_again = this->seq;
12
13     // Restart if seq and val inconsistent
14     if (rep_p_seq_again != rep_p_seq || rep_v_seq_again !=
       rep_v_seq)
15       continue;
16
17     // Help to complete another ongoing write
18     if (rep_p_seq == rep_v_seq+1) {
19       FLUSH(rep_p_addr);
20       FENCE();
21       before = {rep_v_val, rep_v_seq};
22       after = {rep_p_val, rep_p_seq};
23       DWCAS(this, before, after);
24       continue;
25     }
26
27     // Make sure we have the same versions
28     if (rep_p_seq != rep_v_seq) continue;
29
30     // If value on rep_p is not expected, fail
31     if (rep_p_val != expected) {
32       expected = rep_p_val;
33       return false;
34     }
35
36     // Update rep_p
37     before = {rep_p_val, rep_p_seq};
38     after = {newVal, rep_p_seq+1};
39     bool res = DWCAS(rep_p_addr, before, after);
40     FLUSH(rep_p_addr);
41     FENCE();
42     if (res) {
43       DWCAS(this, before, after);
44     } else {
45       if (before.val == expected)
46         continue;
47       DWCAS(this, {rep_v_val, rep_v_seq}, before);
48     }
49     return res;
50   }
51 }
```

Figure 5.5: Patomic Load Implementation

```
1 template<typename T>
2 T patomic<T>::load () {
3   return this->value.load();
4 }
```

instruction to the volatile memory, rep_v, after a successful write of the same value and sequence number to the persistent memory in line 39. The linearization happens between lines 39–43. The linearization point of an unsuccessful operation can occur in line 31 or in line 47.

Lemma 5.4.1. *The implementation of the load and CAS operations is a linearizable implementation of an atomic variable.*

Before we prove this lemma, we prove some helping ones.

Lemma 5.4.2. *The persistent value can be changed at time t only if the sequence numbers on persistent and volatile memories match right before t.*

Proof In line 28, a writing process checks whether the sequence numbers on the volatile and persistent memory are the same. Only after it has done so, will it try to change the persistent memory. As the change is done by a DWCAS and first done to persistent memory, it will succeed only if the current value on the persistent memory equals the expected value, which was equal on both the persistent and volatile memories. ■

Lemma 5.4.3. *The sequence number in volatile memory is always lower by one or equal to the sequence number in persistent memory.*

Proof According to Lemma 5.4.2, only if both the sequences on the persistent and volatile memories match, will there first be an attempt to change the persistent memory. If this attempt is successful, then after being equal, the sequences are at a distance of one, as the DWCAS changes the persistent memory to contain the current sequence raised by one due to lines 38–39. At this point, there is an attempt either by the same thread in line 43 or by others in line 23 or line 47 to make the sequence number on the volatile memory match the persistent memory. Moreover, to change the volatile sequence, the expected sequence is always lower by one than the current sequence in the persistent memory. The first that succeeds, match the sequence in volatile and persistent memories again. In addition, once the volatile sequence has changed, it might never get the same sequence number again, as sequence numbers on persistent memory are always monotonic, and as a consequence, on volatile memory as well. ■

Lemma 5.4.4. *If the sequence numbers on volatile memory and persistent memory match, then the values must also match.*

Proof DWCAS always updates the sequence number atomically with a related value, and there is only one value attached to a sequence number that is successfully written to the persistent memory. If the DWCAS fails, the next attempt will get a larger sequence number. Correspondingly, as values are first written to the persistent memory and only then to volatile memory, the process that succeeded in writing to the persistent memory will attempt to write the same value and sequence number to the volatile memory. According to lines 18–25, other processes can write to the volatile memory as well, but only the value that currently exists in the persistent memory, and will succeed only if the current sequence number of the volatile memory is lower by one than the current sequence number of the persistent memory. Therefore, there is always a match between the value and sequence number on the volatile and persistent memories. ■

Let us proceed to the proof of Lemma 5.4.1.

Sketch Proof Let v_a be an atomic variable with the value v, which currently exists in the persistent memory, implemented by the Mirror's load and CAS operations. Let us assume, w.l.o.g. that the sequence number that is related to the value v is s at time t. Let p_1 be the first process that attempts to change v to v' after t, and let p_2 be a process that loads the current value. We show that a load always reads the last successful CAS. According to Lemma 5.4.3, the sequence number of the volatile memory is distant from the sequence number of the persistent memory by at most one. Moreover, according to Lemma 5.4.4, the value corresponds to the sequence number. Thus, we have two possible scenarios:

1. Both replicas of the value and the sequence number are the same in the volatile and persistent memories at time t. If p_2 reads before p_1's attempt, i.e., when the values of the persistent and volatile memories have not changed, by the semantics of the std::atomic library, p_2 will return the value v as written in the volatile memory, as expected. If, however, p_2 reads after t, this means that p_1 has already attempted to change v_a. p_1 first tries to change the persistent memory, and succeeds (since it is the first one after t, no other process manages to change the persistent replica before this and the values remain the same). Afterwards, this process, and all the other concurrent processes that try to CAS as well help (or fail) to change the value in the volatile memory so that they match. In other words, there was a point in time where both values on persistent and volatile memories matched because if they did not, no other process could have changed the persistent memory again due to Lemma 5.4.2. This happens before line 43. If p_2 reads before that linearization point, it still reads the value v. Otherwise, it will read the value v', which is after the linearization point, which is the expected value. In both cases, p_2 reads the correct value. Any other process that tries to change v as well will fail if p_1 is the first one that attempts to change v_a. A special case exists where the expected value and the new value are the same. In this case, a process might still fail in line 39 as

the sequence number might have changed but a regular CAS should not fail. If the former case occurs, this process will try again (line 46). If another process fails due to p_1's success, it helps to update the volatile memory to match both replicas before its return.

2. The persistent replica is off by one from the volatile replica. In this case, p_1 will recognize this state in line 18 and help to update the volatile value so that it is equal to the persistent value. If p_1 is the first process that manages to update v to v', this means it reached line 39, i.e., it passed line 28 where both replicas are the same, and then we go back to the first case. If p_2 reads before the values match, i.e., before v reaches the volatile memory, it means that the operation that wrote value v was not linearized yet, and p_2 would return the current value of the volatile memory, which is the previous one, as expected. If it reads after v was written to rep_v, then the operation that wrote v was linearized and p_2 would return that value. If, however, p_1 fails to change the value in the persistent memory, one of the following might have happened: (1) the values did not match, meaning that another process has managed to change rep_p in contradiction to the fact that p_1 was the first one that attempted to change v_a or (2) the expected value was different than v. In this case, we expect the operation to fail, exactly as occurs in line 31. ■

We now outline the proof for Theorem 5.1.

Sketch Proof To prove a data structure is durably linearizable we need to show that if we remove all crash events, the history remains linearizable. Let us consider a history H with one crash event c at the end (if there is more than one crash, the theorem could be proved by induction). We construct an equivalent history H' without a crash that contains all the persisted writes. By showing that history H' is linearizable, we prove that the original history H is durably linearizable. As mentioned above, all linearization points were already persisted before their occurrence, thus, all linearized operations survive a crash. Consider all running processes p_1, p_2, ..., p_i in H whose last instruction before the crash was a successful CAS on the persistent memory. As there might be only one successful CAS on a single location, there are different such locations. Our recovery operation simply copies the content of the persistent memory to the volatile memory, and linearizes those i successful CASes, as every location has only one successful CAS instruction. All other processes are paused and do not continue executing. The paused processes have not written anything to persistent memory (otherwise they would have been considered as among the mentioned i processes). Therefore, these processes have not changed the data structure, and have no effect. We get that H' is equivalent to H and linearizable. Since our data structure is lock-free, new processes will be able to continue executing from that state. ■

5.5 Evaluation

5.5.1 Experimental Setup

We ran our measurements on an an Intel machine with two Xeon Gold 6234 processors, each with 8 cores, 3.3GHz max frequency and 2-way hyper-threading, which were disabled during the experiments to increase stability. The machine has 366GB of DRAM and 1.5TB of NVMM (Intel OptaneTM DC memory), organized as 12× 128GB DIMMS (6 per processor). Each core has an L1 cache of 32KB and an L2 cache of 1MB. The L3 cache is 25MB per processor (8 cores). The operating system is Ubuntu 18.04.1, and code was written in `C++` compiled using `g++` (GCC) version 9.3.0. with -O3 optimization. We used an *App-Direct Mode Interleaved* in our configuration. For persisting objects, we called the *clwb* and *sfence* instructions for flush and fence, respectively. We use the *clwb(address)* instruction followed by an *sfence* to allow different write-backs to occur in parallel. To measure the influence of different flush instructions on our construction, we tried to use also *clflush* and *clflushopt* instead of *clwb* and got the same results up to a statistical error. We believe it happens due to the way our algorithm works, as there is DWCAS right after every flush instruction, which acts as a fence on Intel platform. In addition, current NVRAM platforms invalidate cache lines after they are flushed (even by clwb), which implies the same cache misses. Therefore, once clwb does not invalidate cache-lines, it may improve our performance, and others techniques even further. To implement Mirror on ARM, the analogue instructions are *DC CVAP* and a full system *DSB* instruction for flush and fence execution.

To fix the memory reclamation and make it similar in all compared algorithms, we used the *libvmmalloc* implementation of the PMDK library [55] and the durable version of the *ssmem* memory manager for all compared algorithms. ssmem is the same allocation method used in related work [95, 40]. We work with key-value pairs, both of size 8B. Nodes are cache-aligned to 128B. The reported results are averages of 10 repetitions, each ran for 5 seconds. We used a uniform random key distribution from the range of $[0, r-1]$ for varying r's. Every data structure was initialized with $r/2$ keys before the run, and measured with varying percentages of reads that cover the standard YCSB benchmark [29]: A (50% reads), B (95% reads), and C (100% reads). In other experiments, we also ran the frequently used workload of 10% inserts, 10% deletes and 80% read operations.

We evaluated the performance of our construction on four different linearizable and lock-free data structures: A Linked-List [49], A Hash-Table, (based on Harris et al.'s [49] with a linked-list in every bucket), a lock-free BST by Aravind et al. [77] and a lock-free Skip-List [39]. We compared our general construction with two other general constructions: (1) Izraelevitz et al.'s [59] construction that adds a flush and a fence for every shared read/write operation, and (2) the NVTraverse [40] construction that can be applied to traversal data structures (defined in [40]) and removes the need

for persisting traversals. Data structures generated by NVTraverse apply flushes and fences only to reads and writrs of fields in the nodes around the "destination" of the operation, where operations actually take effect. We also added the state-of-the-art ad hoc construction for sets: *SOFT* and *LinkFree* by Zuriel et al.'s [95]. This allowed testing the performance of data structures automatically output by Mirror to highly optimized hand made data structures whose design requires high expertise. In addition, we also tested our construction against an Intel's engine, Cmap in pmemkv [85], which is a persistent lock-based key-value datastore.

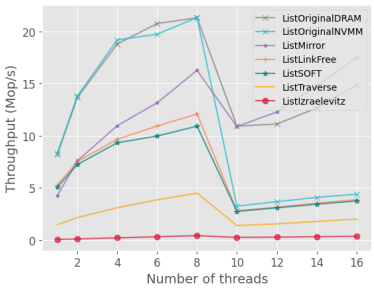
5.5.2 One Replica on DRAM

We first checked the performance where the volatile replica of the data structure output by the Mirror transformation is placed in the volatile memory (DRAM), and the other (persistent) replica is placed in the non-volatile memory. This configuration enabled both advantages of the Mirror transformation: no persisting of reads, and fast DRAM read executions.

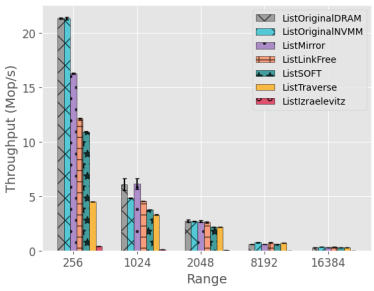
List Scalability

The results for the Linked-List's scalability are shown in Figure 5.6 (a). We ran $1-16$ threads, where above 8 threads, we cross the NUMA-node boundaries, which leads to some threads reading from further and slower memory. The list is initialized with 128 keys over a key range of $0-255$, and the workload executes randomized 80% look-ups, 10% inserts and 10% deletes. Since the list is small, it resides on the cache, and therefore sometimes the original non-persistent list *ListOriginalDRAM* from [49], has the same throughput as its version *ListOriginalNVMM* that executes on the NVMM, up until crossing the node boundaries. (The non-persistent version executes no flushes or fences). Crossing the NUMA-node boundaries makes data go through the memory and then the use of DRAM implies faster execution.

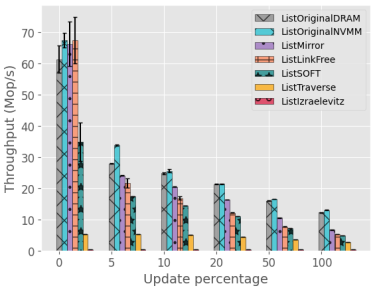
Looking at the three general techniques for durability, we see that the persistent list output by the Mirror transformation outperforms the list output by *NVTraverse* [40] by $2.88x-8.7x$ on $1-16$ threads respectively. The list output by NVTraverse outperforms the Izraelevitz et al. [59] list by $29x$ for one thread, $7.7x$ for 8 threads and $5.6x$ for 16 threads. Throughout the experiments, this advantage of the data structures' output by Mirror over their competitors is evident. It remains to compare to the hand made version of the linked list of Zuriel et al. [95], we notice that our list and Zuriel's list are comparable up to 4 threads, but when contention becomes higher, meaning that more writes occur because there are more running threads, Mirror's list outperforms Zuriel's list by up to 35% on 8 running threads. More threads means more writes, more write-backs, and more cache misses that are served faster on DRAM for the Mirror data structure. The NUMA effects are more chaotic, making the Mirror list extremely successful, but we leave the study of NUMA behavior on non-volatile memory to future



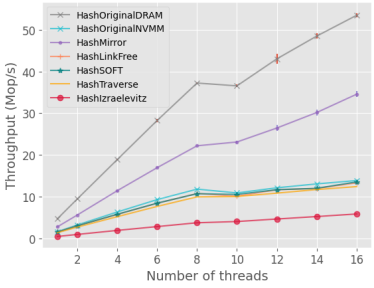
(a) Linked-List, varying threads, 80% lookups, 128 nodes



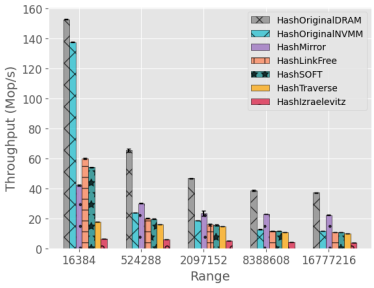
(b) Linked-List, varying size, 8 threads, 80% lookups



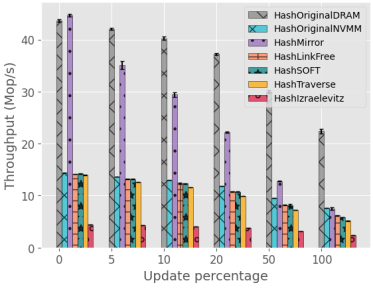
(c) Linked-List, varying update percentage, 8 threads, 128 nodes



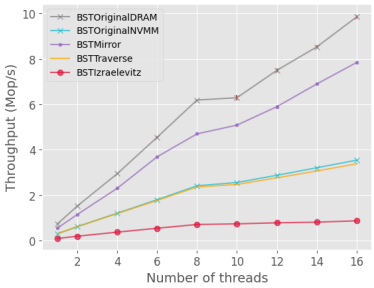
(d) Hash-Table, varying threads, 80% lookups, 8M nodes



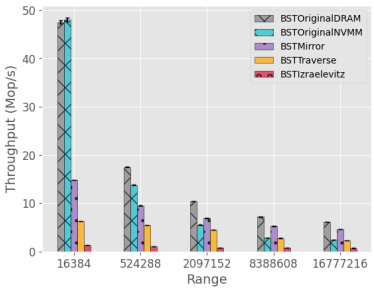
(e) Hash-Table, varying size, 8 threads, 80% lookups



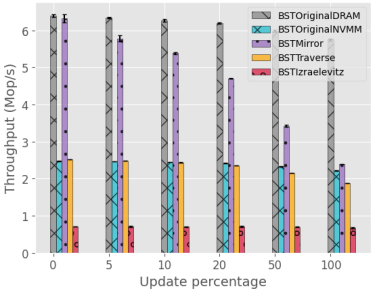
(f) Hash-Table, varying update percentage, 8 threads, 8M nodes



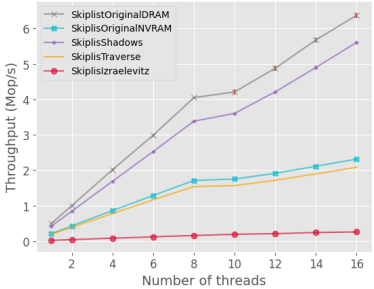
(g) BST, varying threads, 80% lookups, 8M nodes



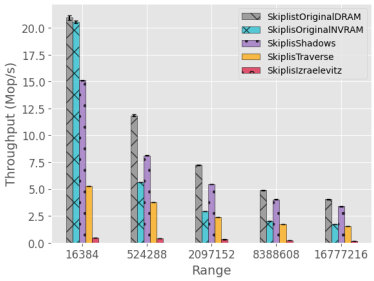
(h) BST, varying size, 8 threads, 80% lookups



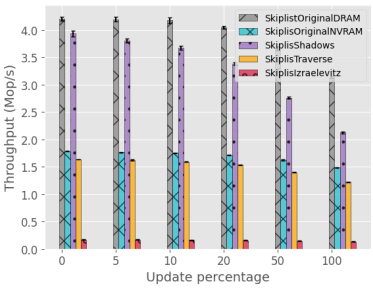
(i) BST, varying update percentage, 8 threads, 8M nodes



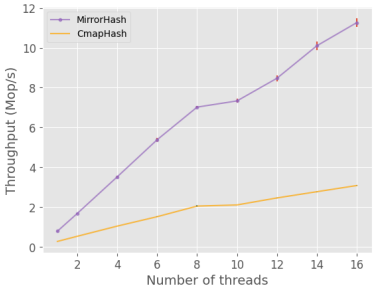
(j) Skip-List, varying threads, 80% lookups, 8M nodes



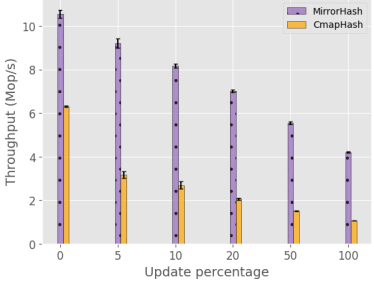
(k) Skip-List, varying size, 8 threads, 80% lookups



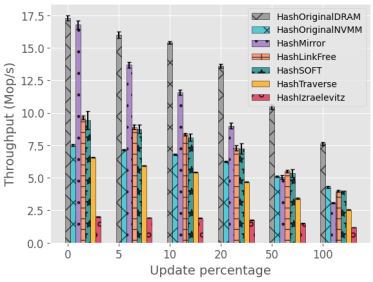
(l) Skip-List, varying update percentage, 8 threads, 8M nodes



(m) Hash-Table, varying threads, 80% lookups, 8M nodes



(n) Hash-Table, varying update percentage, 8 threads, 8M nodes



(o) Hash-Table, varying update percentage, 8 threads, 32M nodes

Figure 5.6: NVMM throughput results with one replica of Mirror placed on DRAM.

work.

List Varying Size

Next, we measured the throughput for varying list sizes, with 8 running threads, and a workload of 80% reads, 10% inserts, and 10% deletes. The results appear in Figure 5.6 (b). Notice that as lists become larger with dominating traversal times, the differences between the various implementations become less noticeable. When traversals dominate performance, the different number of flushes/fences become less crucial. Long lists are not used in practice as it does not make sense to traverse a thousand nodes in order to locate the desired key.

We mostly see the same trends that we saw earlier. Up until the range of 8192, we notice that the Mirror construction outperforms NVTraverse by $3.6x - 1.25x$ on ranges of $256 - 2048$ keys. For larger ranges of keys, we get comparable performance.

List Updates

Figure 5.6 (c) studies the performance of workloads with various percent of updates with 8 running threads, and a list of size 128. On the read-only workload with 0% updates, loads are obtained from the cache so we expect all algorithms that do not execute any flush/fence instructions to perform excellently. This includes both original non-durable versions (either executing on DRAM or on NVMM), the Mirror list, and Zuriel's hand-made Link-Free algorithm. The 95% confidence intervals for all these algorithms overlap. Zuriel's SOFT behaves differently because it consumes more space with its split nodes. As the update percentage grows, the performances of all implementations decrease, as there is more contention of writes on the data structure, but the trend among the compared algorithms remains the same.

Hash-Table, BST and Skip-List Scalability

Moving to much larger data structures that do not fit into the cache, we now examine the hash-table, the binary search tree (BST) and the skip-list scalability, presented in Figures 5.6 (d), (g) and (j). We ran $80\% - 10\% - 10\%$ look-ups-inserts-deletes with a structure size of $8M$ nodes. Here, memory accesses (reads and writes) from the main memory, whether the DRAM or the NVMM, dominate the performance. Accessing the DRAM is much cheaper, and on a single thread execution we see Mirror's hash-table outperforming the hash-table generated by NVTraverse, and Zuriel's hash-tables by a factor of 1.8x. These three competitors behave similarly to each others. With 8 threads the difference grows to 2x and with 16 threads to 2.5x. For the BST, we see Mirror's BST outperforming NVTraverse's BST by a factor of 1.84x-2.33x on $1 - 16$ threads. For the skip-list, the difference even grows to a factor of 2.1x-2.65x on $1 - 16$ threads. The fact that reads are never persisted, in addition to the fact that reads access DRAM,

benefits the Mirror BST and skip-list considerably, especially with data structures that do not fit in the cache.

Hash-Table, BST and Skip-List Varying Sizes

We now consider the effect of running varying sizes of hash-tables BTSs and skip-lists with 8 threads, and $80\% - 10\% - 10\%$ look-ups-inserts-deletes. The results are depicted in Figures 5.6 (e), (h) and (k). Excluding the smallest $8K$ size, the data structures do not fit in the cache and need to be read from the memory. Since Mirror's data structures use two replicas (and double the memory consumption) they suffer more cache misses. This is why they do not perform as well as the original (non-persistent) versions of the data structures and Zuriel's hand-made (persistent) data structure on 8K structures' size. Nevertheless, the data structures generated by the Mirror transformation always outperform the data structures output by NVTraverse and Izraelevitz's et al. When the size of the data structure grows to 256K keys, no implementation is small enough to fit in the Last Level Cache (LLC) and we see the same trends as before: reading from the NVMM and persisting the reads downgrades the performance. The Mirror's hash-table outperforms Zuriel et al.'s link-free hash-table by a factor of 1.5x–3x, and it outperforms NVTraverse by a factor of 2.8x–3.4x, for the larger (realistic) structure sizes of 255K–8M nodes.

Hash-Table, BST and Skip-List Updates

We also evaluated the hash-tables, BST and skip-lists to examine the impact of various update workloads. We ran 8 threads with $8M$ nodes. The related graphs are presented in Figures 5.6 (f), (i) and (l). We also tested a larger data structure, a hash-table with 8 threads and $32M$ nodes ,to see if it makes any difference. The results are depicted in Figure 5.6 (o). On a read-only workload, the data structures generated by the Mirror transformation perform similarly to the non-persistent original data structure executing on the DRAM, simply because they both access the DRAM only. But as the percentage of writes increase, the throughput of the Mirror data structures degrades, as writes are executed on the non-volatile memory as well. Nevertheless, Mirror's structures beats all other persistent data structure significantly, in both sizes. The only place where SOFT and Link-Free results are better than Mirror's results are in $32M$ nodes and above 50% updates.

Lock-Based Key-Value Datastore

To compare our construction with a lock-based data structure, we used the concurrent key-value store from Intel's pmemkv library [85], which is optimized for persistent memory. We tested Cmap, which was the only concurrent non-experimental engine that was provided at the time we tested. Since it was based on a hash-table, we ran

it against Mirror's hash-table. We used the pmemkv-bench [56] which is a testing framework based on db-bench from LevelDB [45] and RocksDB [38].

We ran the tests with $8M$ keys of a size $8B$. The value size was $8B$ as well and the benchmark executes randomized reads and writes. The scalability of both of those key-value stores are shown in Figure 5.6 (m) with 80% reads and 20% writes. We notice that the Mirror's construction outperforms Cmap significantly as Mirror is lock-free and takes advantage of its DRAM copy. Mirror's hash-table outperforms Cmap by 2.85x-3.65x on $1-16$ threads.

The same trend is shown for various update workloads with 8 running threads, depicted in Figure 5.6 (n). Mirror performs better by 1.67x-3.95x on $0\%-100\%$ writes respectively.

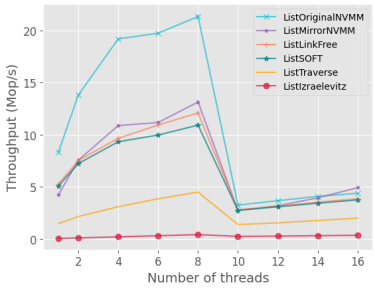
We conclude this part of the evaluation, where the volatile replica is placed on the DRAM, that even though the Mirror construction executes two writes, it still outperforms other durable constructions due to its usage of DRAM. Next, we check how Mirror's data structures behave when both replicas are placed on non-volatile memory. It may be important for possible future architectures in which DRAM will not be incorporated.

5.5.3 Mirror with Both Replicas on NVMM

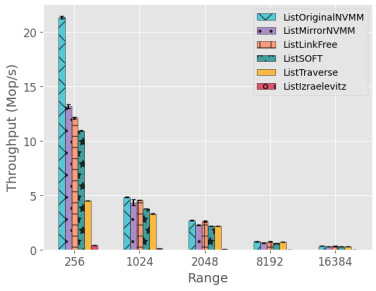
Since volatile memory may not be available in some future platforms, it is interesting to check how Mirror's data structures performs when we allocate both of the replicas on the persistent memory. We still expect to see benefits from never persisting a read value, but we also expect reduced performance due to slower read accesses to the volatile replica (which is now on non-volatile memory), and also writing twice is more costly. In the evaluation that follows, we see that when the two replicas reside on non-volatile memory, Mirror is competitive with NVTraverse, sometimes better and sometimes worse, yet Mirror is somewhat easier to implement.

The Linked List

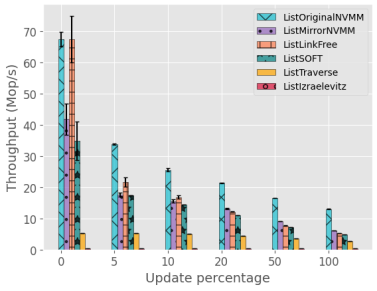
Figures 5.7 (a) - (c) show the performance of the linked list with the same workloads as in Figure 5.6. As expected, writing to NVMM has a much higher cost than writing to the DRAM, as we write twice to the two replicas. Mirror's data structures still perform better than state-of-the-art general constructions (NVTraverse and Izraelevitz). Nevertheless, Zuriel's hand-made lists, the SOFT and Link-Free, perform better on longer lists and on workloads with more than 20% writes. Link-Free and SOFT use an optimization that eliminates repeated redundant persisting operations. This optimization helps most with a low percentage of updates and low contention. But managing this optimization with more than 20% updates is costly and not as useful. This is why Mirror's data structures compete well with the Zuriel's manually optimized data structures in workloads with higher update percentage. As before, when the list grows,



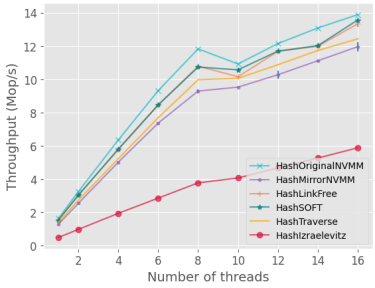
(a) Linked-List, varying threads, 80% lookups, 128 nodes



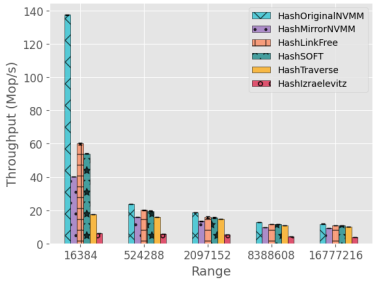
(b) Linked-List, varying size, 8 threads, 80% lookups



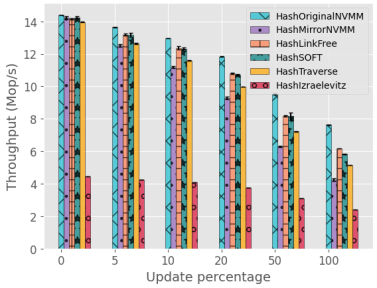
(c) Linked-List, varying update percentage, 8 threads, 128 nodes



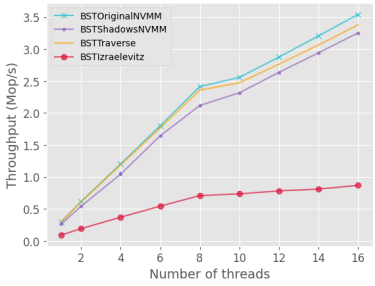
(d) Hash-Table, varying threads, 80% lookups, 8M nodes



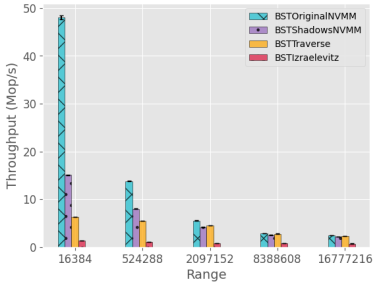
(e) Hash-Table, varying size, 8 threads, 80% lookups



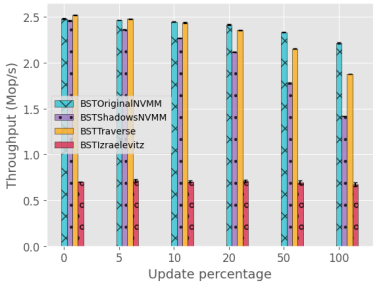
(f) Hash-Table, varying update percentage, 8 threads, 8M nodes



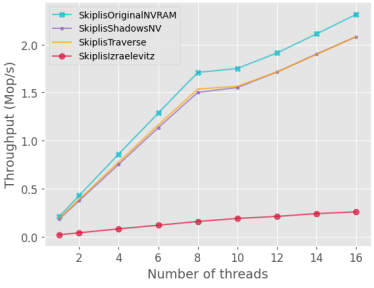
(g) BST, varying threads, 80% lookups, 8M nodes



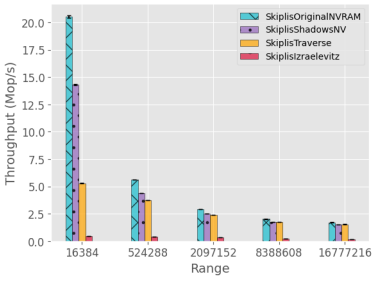
(h) BST, varying size, 8 threads, 80% lookups



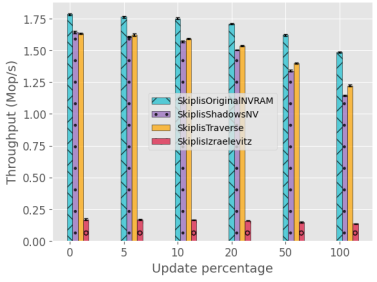
(i) BST, varying update percentage, 8 threads, 8M nodes



(j)



(k)



(l)

(j) Skip-List, varying threads, 80% lookups, 8M nodes

(k) Skip-List, varying size, 8 threads, 80% lookups

(l) Skip-List, varying update percentage, 8 threads, 8M nodes

Figure 5.7: NVMM throughput results. Both copies of Mirrors on NVMM.

the traversals dominate the performance, and the differences between implementations reduce.

Hash-Table, BST and Skip-List

The Hash-Table, the BST and the Skip-List results are shown in Figures 5.7 (d) - (l). Interestingly, we see that the extra cost of writing to two different memory locations has a significant impact in this case, and Mirror does not come out best. In workloads of at most 10% updates, Mirror's data structures are comparable to those of NVTraverse, because the extra writing cost is not as significant as the benefit of eliminating the need to persist the reads. In contrast, for workloads that require more than 20% writes, NVTraverse becomes better. Moreover, the optimization for the hand-made implementations, and the fact that they do not persist pointers, avoids extra flushes and fences, which gives such implementations an advantage over the general techniques. In terms of data structure size, for smaller data structures we see an advantage for Mirror, but NVTraverse becomes faster as the size grows.

5.6 Related Work

Izraelevitz et al. [59] presented a general technique that provides durable linearizability and can be applied to any lock-free data structure. This generality, however, comes at the cost of efficiency. To address this low performance, Zuriel et al. [95] presented *SOFT* and Link-Free, a technique applicable to specific set-like data structures, which eliminates the need to persist pointers.

NVTraverse [40] reduces the cost of most of the loads for *traversal data structures*. We introduced a form of a data-structure called *Traversal data structure* which starts with a read-only phase, called *traversal* followed by a *critical* phase. In this construction, most of the read values from the traversal phase are not persisted, reducing the persistency cost. NVTraverse provides an automatic way to insert flushes and fences but it requires the lock-free data structure to be in a special traversal form. Transforming a data structure into this format may require some efforts and proving that it suffices all the required conditions requires some expertise. Moreover, in some cases conversion to the traversal form may reduce efficiency, e.g. trimming virtually deleted elements during a traversal in a linked-list. Nevertheless, we are not aware of any data structure that cannot be converted to the traversal form. As opposed to Mirror, NVTraverse does not require underlying services such as a specific allocator and it has a smaller footprint.

5.7 Conclusions

In this chapter we presented a simple and effective automatic transformation from lock-free linearizable data structures to persistent lock-free data structures that satisfy durable linearizability. Our transformation can make use of a hybrid system where non-volatile main memory co-exists with conventional DRAM and generate data structures that are extremely efficient in this environment. Evaluation of a linked-list, a hash table, and a binary search tree demonstrates that our method always beats state-of-the-art transformations for lock-free data structures, sometimes by a factor of up to 10x, depending on the percentage of read operation, the size of the structure, and the number of executing threads. Applying the transformation on a given lock-free data structure is easy, involving type annotation to replace the usage of `std::atomic` with `patomic`, replacing the calls to the system allocator with the Mirror allocator, and providing a traversal function for the nodes in the data structure.

Chapter 6

A Library for Efficient Persistence: Flit

6.1 Introduction

Data races in persistent programs pose even more challenges than usual. Since writing and persisting values cannot be done atomically, a value can be visible to other threads before being persisted. Thus, to avoid memory inconsistencies, a process may have to flush locations it reads, even if processes flush locations when they write as well. In most cases, however, a writing process can finish persisting its new value before any other process reads it. In that case, it seems wasteful to have the reader flush this value as well. Existing work in the literature avoids these wasteful flushes by using a bit in each memory word to indicate whether or not it has already been flushed [32, 90, 48]. This optimization has been shown to have tremendous benefits in practice, but borrowing a bit from each word is not always possible. Furthermore, this optimization requires modifying memory using compare-and-swap, and therefore cannot be applied to data structures designed with other primitives, such as fetch-and-add or swap.

We propose a new technique for avoiding unnecessary flushes which is fully general in the sense that it can be applied to any code safely. The idea is to use counters (separate from the memory word) to keep track of ongoing stores for each variable. When a store begins, it *tags* the memory location it operates on by incrementing the corresponding counter. Loads check the counter when accessing a given memory location, and only execute a flush instruction on it if it is tagged. In this way, flush instructions are only executed when needed. This technique allows for flexibility in the placement of these counters. The counters can be, for example, placed next to each variable or in a separate hash table. We experiment with different options in Section 6.6.

We package this technique into an easy-to-use `C++` library called *FliT* [91], or *Flush if Tagged*, which helps programmers easily design efficient persistent code for NVRAM, abstracting away details of flush and fence instructions, and applying the optimization under the hood. At a high level, the FliT library persists the effect of each instruction

without requiring the programmer to handle low-level flushing and barriers.

The FliT library greatly improves the performance of persistent code, since it enables the program to safely skip flush instructions when they are not needed. Furthermore, FliT is easy to use, and its syntax requires minimal changes when applying it to existing code. Indeed, to use FliT, the programmer simply needs to modify the declaration of variables to be persisted, and annotate when an operation terminates – this already makes any linearizable data structure durably linearizable [59]. For example, a `C++` implementation of Harris's linked list [49] can be made durably linearizable using our library by changing just seven lines of code.

Another advantage of the FliT library is its flexibility; while, by default, the FliT library instruments each load and store instruction to access the tag counters, this does not have to be the case. Many previous works have focused on understanding which values must be persisted, and which can be left volatile [41, 40, 32, 21, 23, 26]. These efforts have led to many optimized persistent data structure implementations. The FliT library can complement these existing works by allowing the programmer to specify whether a specific instruction's arguments should be left volatile. In that case, the instruction can be annotated as such, and the flushing mechanism is bypassed. Thus, while the FliT library can be used to persist all memory values in a naive manner to yield a fairly performant solution, it can be combined with existing optimizations to yield even better results. Skipping the flush instructions for already flushed values is one benefit when using this library, and we also suggest implementing it using the hardware, which may increase the performance in durable programs.

To more formally argue about the library's correctness, we define an *abstract interface*, called the *P-V Interface*, which the FliT library implements. Intuitively, the interface considers two types of instructions; those whose effects must be persisted (called *p-instructions*), and those whose persistence has been optimized away (called *v-instructions*). The P-V Interface describes the interaction between these two types of instructions and the resulting effect on the memory. We show that the P-V Interface captures persistence behavior in many algorithms in the literature. Intuitively, the P-V Interface abstracts flush and fence instructions down to their underlying meaning, and we use it to show that the FliT library behaves as expected. We believe that the P-V Interface offers a good balance between ease of programming and the efficiency of potential implementations. Since it is relatively low-level, it can be implemented efficiently, as is exemplified by FliT. Furthermore, designing durably linearizable data structures [59] is easy using the P-V Interface; if every instruction is made a p-instruction, a linearizable data structure becomes durable. On the other hand, carefully reasoned optimizations can also be applied by making some instructions v-instructions where possible. Thus, we believe that the P-V Interface may be of independent interest.

We evaluate the FliT library by using it to implement four different durable data structures; a linked-list [49], a BST [77], a skiplist [32], and a hash table [32]. Further-

more, for each data structure, we evaluate three different ways of making it durable; one that makes all instructions p-instructions, and two more optimized settings that appear in the literature; we consider the NVtraverse methodology [40], which allows us to have v-loads while traversing the data structure, and a manually optimized durable version of the same data structure [32]. We also evaluate different settings for the placement of the counters in the implementation of FliT, and compare these to the existing bit-tagging technique [32, 48, 90]. We observe that, the FliT library provides up to 200× speedup over a durable linearizable version implemented with plain flush instructions. Furthermore, even for highly optimized implementations, the FliT library still provides up to 4.32× speedup and never slows down any implementation.

The rest of this chapter is organized as follows. Section 6.2 reviews some critical previous work. Section 6.3 provides a description of the abstract P-V interface that Flit implements. We provide the details the FliT library, which implements the P-V interface defined in Section 6.3, in Section 6.4. Section 6.5 presents the implementation of the FliT library, and prove that it satisfies the P-V interface. The experimental evaluation is presented in Section 6.6. Section 6.7 discusses related work and Section 6.8 concludes.

6.2 Preliminaries

We defer most of the discussion of related work to Section 6.7. However, some flushing optimizations have appeared in the literature that are reminiscent of the FliT library's implementation, so we briefly discuss them now. David et al. [32] introduce a technique they call *link-and-persist* to avoid executing flush instructions when the variable being flushed is clean. Their technique works by using a single bit in each memory word as a flag indicating whether or not it has been flushed since the last time it was updated. When a new value is written, it is written with the flag up. The writing process then executes a flush and a fence to persist the new value, and then executes another store to flip the flag down. A reader executes a flush on any location it read that had the flag up, and skips flushing every time the flag is down. This technique has appeared in the literature under different names [48, 90, 95], always optimizing redundant flushs, and yielding faster algorithms. This technique is similar to the implementation of our FliT library. However, the FliT library is more general and flexible. For one, it does not require taking a bit in every memory word. While pointers leave unused bits in each word, some algorithms make use of these bits for other parts of their logic. The link-and-persist technique is not applicable to such algorithms. Furthermore, for link-and-persist to work, all stores must be executed using a CAS instruction (as opposed to, for example, *fetch-and-add* or *swap*), to prevent accidentally removing a flag for a value that has not yet been flushed. The FliT library does not suffer from these restrictions. Finally, as will be shown in the rest of the chapter, the FliT library also provides flexibility in allocating the space used for metadata tracking persistent state,

which can sometimes be a useful way to optimize implementations.

6.3 Persistent-Volatile Instruction Interface

Before presenting the FliT library, we define the abstract interface that it implements. This interface, called the P-V Interface, is important for discussing the correctness of the FliT library implementation; we later prove that our implementation satisfies the abstract interface. Furthermore, this interface allows users of the FliT library to reason about their code in a precise manner.

The P-V Interface aims to capture the behavior of a program with both volatile and persistent memory. Firstly, handling persistence should not affect the behavior of the volatile memory. In particular, this means that we should expect to see the same sequential semantics on volatile memory as we do in a classic system. That is, any load on volatile memory should return the value written by the most recent store. For persistence, we expect the interface to capture the behavior of code that uses flush and fence instructions. We also note that dependencies between instructions can play a role in when we expect a value to be persisted; if a value has been written but never read, it may be ok for it to be lost upon a system crash, since its effects have not yet been observed. We formalize these intuitions below.

We begin defining the interface by introducing terminology to separate two types of instructions: we say an instruction is a *p-instruction* if it has to be persisted (defined below), and a *v-instruction* if it does not. More specifically, we refer to persisted loads and stores as *p-loads* and *p-stores* respectively, and to their volatile counterparts as *v-loads* and *v-stores*. If we do not specify whether an instruction is persisted or volatile, then it could be either. Intuitively, a p-instruction must be followed by a flush and a fence; in this interface, we aim to capture *when* these flushs and fences must occur.

To nail this down precisely, we further distinguish between *shared instructions*, which operate on shared memory, and *private instructions*, which operate on a private memory location. A private instruction may allow more flexibility in when it is persisted, since other processes cannot observe its effects. Moreover, private values are not needed after a crash, since we assume new processes are spawned.

We refer to the memory location an instruction operates on as its *location*. Furthermore, we associate a *value* with each instruction; a *load*'s value is what it returned (read from its location), and a *store*'s value is the value newly written on its location. When we say an instruction is *persisted*, we mean its value is on persistent memory.

To create durable code, we must reason about *dependencies* among different instructions. In particular, for a new store to be safe in a persistent setting, a process i must ensure that all its dependencies have been persisted *before* executing the store. That is, the values that process i used to determine the value and location of the new store must not be lost at a later time. Furthermore, to maintain a store-order guarantee for persistent memory, previous store instructions by the same process i must also

be persisted before i's new store. Finally, to prevent losing the effects of a completed operation, we must persist all of i's dependencies and store values before i completes an operation.

The P-V Interface, defined in Definition 6.3.1, formalizes the meaning of dependencies in terms of p-stores and p-loads; a process i depends on its own p-stores (Condition 2), and on previous p-stores on locations on which i executes a p-load (Condition 3). The interface then requires that these dependencies be persisted before i executes a store that is visible to other processes (shared), or before it completes an operation (Condition 4). To capture which p-stores become dependencies, we consider the *linearization* of instructions. Intuitively, an instruction linearizes at the time it accesses volatile memory (Condition 1). Note that Conditions 1, 2, and 3 apply to both private and shared instructions, and that v-instructions don't add dependencies.

Definition 6.3.1. [The P-V Interface.] Each instruction has a linearization point within its interval, such that:

1. **Keeping Volatile Memory Behavior.** A load r on location ℓ returns the value of the most recent store on ℓ that linearized before r.

2. **Store Dependencies.** Let s be a linearized p-store executed by a process i. i depends on s.

3. **Load Dependencies.** Let r be a p-load by process i on location ℓ. i depends on every p-store on ℓ that was linearized before r.

4. **Persisting Dependencies.** Let t be either the linearization point of a shared store by process i, or the time at which i completes an operation. The value of every store i depended on before time t is persisted by time t.

6.3.1 Applicability of the P-V Interface

In this subsection, we show that for many algorithms designed for NVRAM, it is easy to replace their memory accesses and all flush and fence instructions with p-instructions (for dependencies) and v-instructions (for instructions optimized out as non-dependencies).

Simple durability

We begin by considering how to guarantee durability using the P-V Interface for any given linearizable algorithm. Izraelevitz et al. [59] show that, for any linearizable data structure, if every load-acquire and store-release is accompanied by a flush and a fence, and stores are followed by a flush, then the data structure becomes durable. We show that declaring these instructions as p-instructions achieves the same guarantee. Furthermore, using our implementation of the P-V Interface yields a much faster solution.

Theorem 6.1. *Given a linearizable data structure, if we make all its loads and stores p-instructions, then the resulting data structure is durably linearizable.*

To prove this theorem, we rely on a useful definition for arguing about different concurrent executions of a data structure.

Definition 6.3.2. Two histories are considered *equivalent* if (1) they are legal executions of the same data structure and contain the same set of operations, (2) the return values and the order of invocations and responses of all operations are the same, and (3) the state of memory after the two histories are the same.

We begin the proof with the following lemma, which will help us prove the theorem constructively.

Lemma 6.3.3. *Consider a linearizable data structure D, and let D_p be an implementation of D in which all loads and stores are p-instructions. Given a history H of D_p with a single crash at the end, we can construct an equivalent history with no crashes.*

Proof Consider the set of p-stores in H that are visible but not persisted. Let s be the last such store, i be the process that executed s, and m be the memory location on which s was executed. For convenience, we say that a load ℓ *sees* a store s if ℓ returns the value that s wrote, assuming (without loss of generality) that all values written on memory are unique. We construct an equivalent history H' in which s, and all the loads that see s, never happen.

First, note that after i executes s, i cannot perform any more stores or complete an operation because otherwise, s would have been persisted by Definition 6.3.1. Therefore i only performs load instructions after s and removing these loads leads to an equivalent history.

Suppose some process p' performs a p-load that sees s. After this load, p' cannot perform a store or complete an operation, by the same argument as above. Therefore, removing this load and all later instructions by p' results in an equivalent history. There cannot be any stores to m after s because s was chosen to be the last store that was visible but not persisted.

Now we have an equivalent history in which i performs no steps after s, and no process sees or overwrites the value written by s. Since s will be lost after the crash, we can remove s without violating the legality of the history and without affecting the final memory state.

Applying this argument for each visible but not persisted store yields an equivalent history in which all visible stores are persisted. Therefore we can remove the final crash event without changing the final memory state. The resulting history is equivalent to H and has no crash events, as desired. ■

We are now ready to prove the main theorem.

Proof of Theorem 6.1 Applying Lemma 6.3.3 inductively, we can show that a history of the transformed data structure with any number of crash events is equivalent to some history with no crashes. We know that histories with no crashes are linearizable since the original data structure is linearizable, so the transformed data structure is durably linearizable. ■

NVTraverse

While the above construction is very simple, and can be easily applied to any linearizable algorithm to make it persistent, there may be opportunities to optimize such a construction if some instructions be could identified as non-dependencies (marked as v-instructions). This can give more flexibility to the underlying implementation to omit flush and fence instructions where possible. Indeed, there are several constructions of durable data structures in the literature that do not persist every memory instruction. For example, Friedman *et al.* [40] present a general construction to make certain lock-free data structures persistent more efficiently than the construction of Izraelevitz et al. mentioned above. In particular, they consider data structures in *traversal form*, in which each operation has a read-only traversal phase followed by a short critical phase. Many lock-free data structures, including linked-lists, BSTs, and skip lists, can fit this form. Friedman et al. show that such data structures do not need to execute any flush instructions during the traversal phase. That is, any load in the traversal phase can be thought of as a v-load, and any instruction (load or store) in the critical phase can be thought of as a p-instruction. There is a short *transition* between the traversal and critical phases in NVtraverse, in which some locations that were read during the traversals are flushed. This can be achieved by executing p-loads on those locations.

Other Algorithms

Many other NVRAM algorithms appear in the literature, with various techniques to optimize the interaction with persistent memory. As a general rule of thumb, any instruction that isn't immediately followed by a flush in such algorithms can be seen as a v-instruction, and any other instruction can be seen as a p-instruction. The 'dependency' terminology is used intuitively in several works [41, 32]; generally, non-dependencies in those works can be seen as v-instructions.

6.4 The FliT Library and Interface

In this section, we introduce the FliT library, which implements the P-V Interface defined in Section 6.3. At its core, FliT provides an interface with which to declare each instruction as either a p- or v-instruction (using the `pflag` parameter).

The FliT library is implemented in `C++`. To use the library, a programmer must declare variables as `persist<>`. The `persist` template can take any type. Declaring

Figure 6.1: Basic interface of FliT.

```
1  class persist<T> {
2    public member functions:
3      T load(bool pflag);
4      void write(T value, bool pflag);
5      bool CAS(T oldval, T newval, bool pflag);
6      T exchange(T newVal);
7      // FAA is only supported if T is an int type
8      int FAA(int amount, bool pflag);
9    public static functions:
10     void operation_completion();
11 } ;
```

a variable in this way essentially allows the FliT library to track its persistence state. Whenever this variable is accessed for loads or stores, the instruction is overloaded with the library's implementation of it, which we call a flit-instruction. Each flit-instruction takes the standard arguments for its underlying instruction, in addition to a flag specifying whether it is a v- or a p-instruction. Finally, a special `operation_completion` function is made available, which must be called at the end of each data structure operation. Figure 6.1 shows the basic interface.

The FliT library further improves the syntax of this interface to allow for minimal code changes to apply it. In particular, when declaring a variable in the `persist` template, a default `pflag` value can be specified, making the `pflag` argument optional when executing instructions on this variable. Furthermore, we overload the `->` and `=` operators to execute FliT loads and stores instead of the default one. These operators can only be used with the default `pflag` value though, since it does not allow for an additional argument.

Figure 6.2 shows an example of the implementation of a concurrent binary tree, achieving durability by making all instructions p-instructions. The change over the original code is highlighted in red. All fields within a node are declared with the `persist<>` template, and given the persisted option as a default for the `pflag`. This means that without any code changes, all accesses to these node fields will be persisted flit-instructions. In the example, all code elided inside the '...' remains identical to the original implementation.

The example above only shows the use of a single setting; all instructions are called as the default p-instructions. The FliT library is in fact more flexible, and is still easy to use even for more complicated code. We note that while not shown in the example, it is also possible to leave a variable declaration as-is, without using the `persist` template, if that variable never requires persistence. This use case arises in some algorithms. For example, Friedman et al. [41] present a durable queue implementation that completely avoids flushing the head and tail pointers of the queue. In this case, these variables can

Figure 6.2: FliT library used for a concurrent BST.

```
1  struct Node {
2    persist<int, flush_option::persisted> key;
3    persist<T, flush_option::persisted> value;
4    persist<std::atomic<Node*>, flush_option::persisted> right;
5    persist<std::atomic<Node*>, flush_option::persisted> left;
6  };
7
8  persist<Node*> root;
9
10 void lookup(int key) { // automatic BST lookup
11   Node* node = root->left;
12   while (node->left != nullptr) {
13     if (key < node->key)
14       node = node->left;
15     else
16       node = node->right;
17   }
18   bool result = (node->key == key);
19   persist::operation_completion();
20   return result;
21 }
22
23 bool insert(K key, V val) { // automatic BST insert
24   ...
25   persist::operation_completion();
26   return result;
27 }
```

be declared normally, without the FliT library.

6.5 The Algorithm

We now describe the implementation of the FliT library, and prove that it satisfies the P-V Interface specified in Section 6.3. At a high-level, each p-store flit-instruction executes a fence before its store, and a flush on this location after the store. This means that already, conditions 1, 2 and 4 are satisfied (ignoring dependencies from Condition 3). If we had a guarantee that every p-load to any location ℓ will always happen after persisting the most recent p-store on ℓ, then Condition 3 would be satisfied as well, without having to change the implementation of load instructions at all. However, this is not the case; since we cannot store and persist atomically, it is possible for another process to read a value written into ℓ by a shared p-store before the writing process persists.

One way to handle dependencies from Condition 3 is to have each p-load execute a flush after reading its value. However, this would introduce many unnecessary flushs, since most flushs do not execute concurrently with a pending p-store on the same location. Our goal in this work is to avoid as much excessive flushing as possible.

The basic idea behind the implementation of FliT is to associate each `persist` variable with a counter, which we call the *flit-counter*. Intuitively, this counter keeps track of the number of pending p-store flit-instructions. When a p-store flit-instruction begins, it increments its associated flit-counter. It then executes its modification, followed by a flush on this location, and then decrements the counter. This counter is checked by all p-loads on this location, and if its value is non-zero, the p-load executes a flush after reading the value. A location whose flit-counter is non-zero is said to be *tagged*, and p-loads only flush locations that are tagged (i.e. Flush if Tagged (FliT)); this is where the FliT library gets its name.

Store flit-instructions also execute a fence before beginning their execution, and another one before decrementing the counter (in the case of a p-store). These fences ensure that all modifications are persisted at the correct times according to Definition 6.3.1. In particular, the fence before a store ensures Condition 4 holds, by making sure all values flushed by this process (which includes all of its dependencies) have been persisted. The fence before decrementing the counter is required for Condition 3; if this fence is not executed, a p-load may observe the flit-counter at value 0 and avoid flushing the location, even though the written value has not yet been persisted.

The FliT library implementation distinguished between shared and private accesses to the memory. The details above in fact describe the implementation for shared accesses. If a given flit-instruction is private, then its implementation is more efficient; we can ignore the flit-counter associated with the accessed location, and avoid the fence before a private p-store. Intuitively, if a flit-instruction cannot be concurrent with any other, then the accessed location is guaranteed not to be tagged (i.e. its

counter has value 0), and return to this state (in the case of a store) before the next flit-instruction accesses it. Therefore, there is no need to check it, or to leave any traces for other processes. Furthermore, note that Condition 4 of the P-V Interface only requires persisting before *shared* stores, so we can skip the fence before private stores. Unless specified otherwise, we always discuss shared flit-instructions in the text, since their implementation is more involved than that of private ones. The pseudocode of the implementation of instructions on `persist` variables is presented in Figure 6.3. Recall that p- and v-instructions are distinguished by the `pflag` argument. We combine all types of store flit-instructions (CAS, FAA, write, etc) into one in the pseudocode, since their behavior is the same.

Note that we do not specify how each memory location is associated with a flit-counter. In Section 6.5.1, we discuss possible ways to assign counters to memory locations, but we note that this is flexible. In particular, having many concurrent stores to the same memory location, or sharing a flit-counter among several locations, cannot result in unsafe behavior (though it may result in extra flushs executed).

We now argue that our implementation (Figure 6.3) satisfies Definition 6.3.1. We begin with a simple lemma.

Lemma 6.5.1. *The value of any flit-counter is always non-negative.*

Proof Only p-stores change the value of an flit-counter. Furthermore, each p-store increments the flit-counter associated with its location exactly once, and decrements the same counter exactly once. The increment is always executed before the decrement. Therefore, the balance on the flit-counter after a p-store terminates is always 0, and the balance during a p-store is either 0 or 1. ■

Theorem 6.2. *Algorithm 6.3 satisfies Definition 6.3.1.*

Proof We let the linearization point of each flit-instruction on a location X be the time at which it accesses the volatile memory at X. To argue that dependencies are persisted by the time Condition 4 dictates, we argue that a flush is executed on all dependencies defined in Conditions 2 and 3, and that a fence is executed after these flushs and before the time at which they need to be persisted. Given this approach, we handle each condition separately.

Condition 1. Note that all load (resp. store) flit-instructions execute a single load (resp. store) instruction, and the arguments/return values of the flit-instruction are the same as the atomic instruction that it executes.

Condition 2. Each p-store executes a flush instruction (Line 18 or 28), followed by a fence instruction (Line 19 or 29), both of which are after its atomic store instruction (Line 17 or 27). Therefore, the value stored by this flit-instruction is flushed and fenced before the flit-instruction terminates, and in particular has been flushed before the executing process executes another p-store or completes an operation.

Figure 6.3: The Flush-Marking Algorithm

```
1  T shared-load(T* X, bool pflag) {
2    T val = X.load();
3    if (pflag && flit-counter(X) > 0) {
4      PWB(X);
5    }
6    return val;
7  }
8
9  T private-load(T* X, bool pflag) {
10   return X.load();
11 }
12
13 void shared-store(T* X, T* args, bool pflag) {
14   PFENCE();
15   if (pflag) {
16     flit-counter(X).fetch&add(1);
17     X.store(args);
18     PWB(X);
19     PFENCE();
20     flit-counter(X).fetch&sub(1);
21   } else
22     X.store(args);
23 }
24
25 void private-store(T* X, T* args, bool pflag) {
26   if (pflag) {
27     X.store(args);
28     PWB();
29     PFENCE();
30   } else
31     X.store(args);
32 }
33
34 void completeOp() {
35   PFENCE();
36 }
```

CONDITION 3. Consider a p-load, r on location X, executed by process i. We consider two cases: either r executes a flush, or it does not. If r executes a flush on X, then, since a p-store linearizes when it is applied to the volatile memory, all dependencies on this location get flushed, and we are done. If r does not execute a flush, then we again split into two cases. If r is a shared-load, by the algorithm, r must have read a non-positive value in X's flit-counter. By Lemma 6.5.1, this means r read the value 0 on the flit-counter. Consider any shared p-store flit-instruction, s, on X. s's store linearizes after its increment of the flit-counter, and s flushs X after its store linearizes, and executes a fence, before decrementing the flit-counter. Therefore, if s's store has linearized and the flit-counter's value is 0, then s's value must already be persisted, so the condition holds. Furthermore, any private p-store, s', on X, must have completed its execution, executing a flush and a fence, before r could access it. Finally, if r is private, then no store could be pending while r executes. In particular, this means that all values stored by a p-store on this location are already persisted, by the argument above.

CONDITION 4. Note that every shared store flit-instruction executes a fence before any other instruction. Therefore, a process i executing a shared store flit-instruction s ensures that all values on which it executed a flush before beginning s are persisted before s linearizes. By the arguments above, this includes all dependencies of i, unless they have already been persisted earlier. Similarly, i also executes a fence at the end of each operation, when `completeOp()` is called. ■

6.5.1 Placement of the Counter

In Figure 6.3, we intentionally abstracted away how flit-counters are assigned to memory locations, using the unspecified `flit-counter()` function. Note that the flit-counter is completely decoupled from the memory locations it represents, so it can be placed anywhere, and can be shared by any number of locations. Furthermore, the flit-counters can be very small; the maximum value in a flit-counter is at most the number of concurrent processes in the system, since each process can increment at most one flit-counter at most once before decrementing it. Therefore, on most machines, including the one we test on, 8 bits suffice to store a flit-counter without the possibility of overflow.

In this section, we discuss a couple of practical implementations for the `flit-counter()` function, which we later implement and test. However, we remind the reader that other practical implementations are possible, and that the FliT library allows the flexibility of modifying the counter placement to suit the needs of the user.

Adjacent Counter

One straightforward way to implement the flit-counters is to place each counter adjacent to the memory word that uses it. That is, we can make each memory word in an algorithm be a double-word, and use the second word for the flit-counter. The

advantage of this is that the counter for each word X is on the same cache line as X, and therefore accessing it has minimal cost. However, this approach can be inconvenient and wasteful, since this fundamentally changes the memory layout of a given data structure's objects. Indeed, an object that fit in a single cache line might overflow it if all its fields double in size.

Hashed Counter

Another flit-counter placement strategy is to use a hash table; each memory location X hashes into the table, which has a counter in each of its entries. This method allows different memory locations to use the same flit-counter. The number of collisions depends on the ratio of the size of the hash table and the number of threads in the system, since each thread can access at most one hash-table entry per flit-instruction. The advantage of this approach is two-fold. First, it saves memory. In many data structures, especially if they are not highly-contended, most memory locations will have no pending p-stores most of the time. This means that sharing counters results in a negligible amount of extra flushing. Secondly, it does not require changing the layout of memory in the data structure itself, since the flit-counters are not placed in the same cache lines as the data structure elements. However, this can also be a downside in some situations; since the flit-counter is in a separate cache line, accessing it could incur an additional cache miss.

Note also that the hashing method allows us to compact the memory usage of flit-counters even further, by squeezing several counters into each word. Recall that 8 bits suffice for each flit-counter, so we can fit 8 counters in a single memory word. However, compacting the flit-counters in this way can increase false-sharing; many different memory locations could be mapped to counters on the same cache line.

6.6 Evaluation

The FliT library's implementation optimizes flush instructions on shared locations. To highlight its effects and focus on them in the evaluation, we evaluate the library applied to lock-free data structures, in which most memory accesses are shared. We apply the FliT library to 4 lock-free data structures; a linked-list [49], a binary search tree (BST) [77], a skiplist [39], and a hash table which uses Harris's linked list to implement each bucket [49]. For each data structure, we implement three different ways of making it durable; the first is the **automatic** transformation discussed in Section 6.3.1, in which all instructions are made p-instructions, the second is using the **NVtraverse** framework [40], and the third is a hand-tuned (**manual**) construction based on algorithms presented by David et al [32]. We also study the effect of various policies for placing the flit-counters in memory with respect to the memory locations they are associated with. In particular, we implement the adjacent counter variant

(flit-adjacent) and a hash table (flit-HT), for which we test five different sizes. We evaluate the tradeoffs of the different approaches. Finally, we also implement the link-and-persist technique on the data structures that can support it. We compare these implementations of the P-V Interface with the **plain** version, which places flush and fence instructions where necessary, but does not utilize any tagging method to avoid flushs in the loads.

6.6.1 Setup

We run experiments on a machine with two Xeon Gold 6252 processors (24 cores, 3.7GHz max frequency, 33MB L3 cache, with 2-way hyperthreading). The machine has 375GB of DRAM and 3TB of NVRAM (Intel Optane DC memory), organized as 12× 256GB DIMMS (6 per processor).

On Intel/AMD architectures [54, 4], the three available flush instructions are *clflush*, *clflushopt*, and *clwb*, where *clwb* is not blocking and supposed to not invalidate the cache. Thus, *clwb* is the most efficient one, and is the one we use in our implementation. The processors are based on the Cascade Lake SP microarchitecture, which supports the `clwb` instruction for flushing cache lines (flush). However, its implementation of *clwb* still invalidates cache lines. Performance might be improved in future platforms where clwb does not invalidate cache lines. For ordering, we use the *sfence* instruction. The equivalent instructions on ARM are *DC CVAP* and a full system *DSB* instruction for flush and fence execution [6]. We use *libvmmalloc* from the PMDK library to place all dynamically allocated objects in NVRAM, which is configured in an App-Direct mode to let the NVRAM reside alongside the DRAM and allow byte addressable access. All other objects are stored in RAM. The operating system is Fedora 27 (Server Edition), and the code was written in `C++` and compiled using `g++` (GCC) version 7.3.1. We use `std::atomic`s with relaxed memory orders where appropriate. In our implementation of the algorithm presented in Figure 6.3, some of the fence instructions can be omitted because on our Intel machine, atomic instructions (such as CAS and FAA) perform an implicit fence.

We avoid crossing NUMA-node boundaries, since unexpected effects have been observed when allocating across NUMA nodes on the NVRAM . Hyperthreading is used for experiments with more than 24 threads. Unless stated otherwise, all data structures are tested with three different workloads; 0% updates, 5% updates, and 50% updates. Updates are split 50/50 between inserts and deletes, and chosen randomly. All experiments were run for 5 seconds and an average of 5 runs is reported. A grey dotted line (shown in some plots) represents the original (non-persisted) form of the tested data structure.

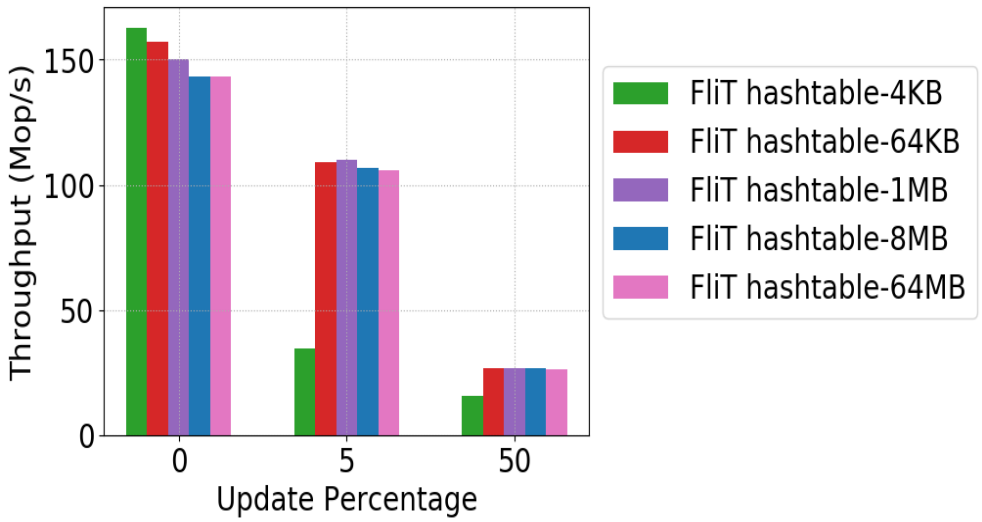


Figure 6.4: Tuning Hash-Table size for the FliT library. Throughput shown is for the automatic BST with 10K keys.

6.6.2 FliT Hash Table Size

We begin our evaluation by testing the effect of the size of the flit-HT on performance. There is a trade-off between memory footprint and the number of collisions on the counters; keeping the table small allows it to fit in cache, making accesses to it potentially cheaper. However, if it is too small, hash collisions could cause cache coherence misses. Figure 6.4 shows the result of different flit-HT sizes on the BST, with three different update ratios. We show the automatic BST implementation. Other data structures showed similar patterns, and are omitted for brevity.

We first note that for 0% updates, we see that the larger the hash table, the lower the throughput. This is as expected; as the flit-HT grows, less of it fits in cache, and therefore accesses to it more frequently incur cache misses. Furthermore, at 0% updates, the flit-counters are never updated, so coherence misses are not a concern. Starting at 5% updates, we see a stark performance drop for the 4KB hash table. Two types of hash collisions can occur in this framework: (1) two locations hash to the same counter, resulting in potentially redundant flushs executed, if the flit-counter balance is inflated due to an ongoing p-store on a different location. More severe, however, is the second type of hash collisions: (2) cache line collisions; the 4K flit-counters in the hash table are packed into only 64 cache lines. This means that if any two p-instructions, at least one of which is a p-store, occur on locations that hash to the same cache line (quite likely), they suffer a coherence cache miss. In such a small hash table, this effect is very prominent. This is much less noticeable in the larger hash tables.

For the rest of the plots, we show only one hash table size; the 1MB flit-HT. We note that this size fits in the L3 cache, but is large enough to avoid most hash collisions.

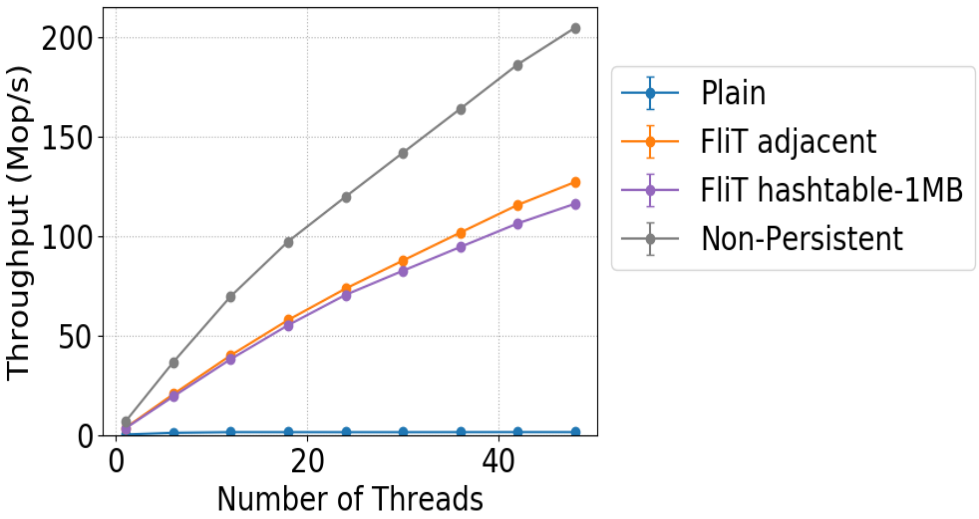


Figure 6.5: Scalability graphs for the automatic BST with 10K keys and 5% updates.

6.6.3 Varying Number of Threads

We now consider the scalability of data structures that use the FliT library, as the number of threads grows. The results can be seen in Figure 6.5. Again, the automatic 10K BST with 5% updates is shown. Note that in this plot, aside from the flit-HT, we show a few different settings for comparison. In particular, the gray line shows a non-persistent version of the data structure, in which no flush or fence instructions are issued. This forms a baseline that cannot be significantly outperformed by any persistent implementation. Furthermore, the blue line shows a BST version implemented with plain flush and fence usage, without applying the FliT library at all. This version performs many more flushs, and its performance and scalability suffer. We show both the flit-HT and and the flit-adjacent versions of the FliT library. Both of them scale similarly, and quite well.

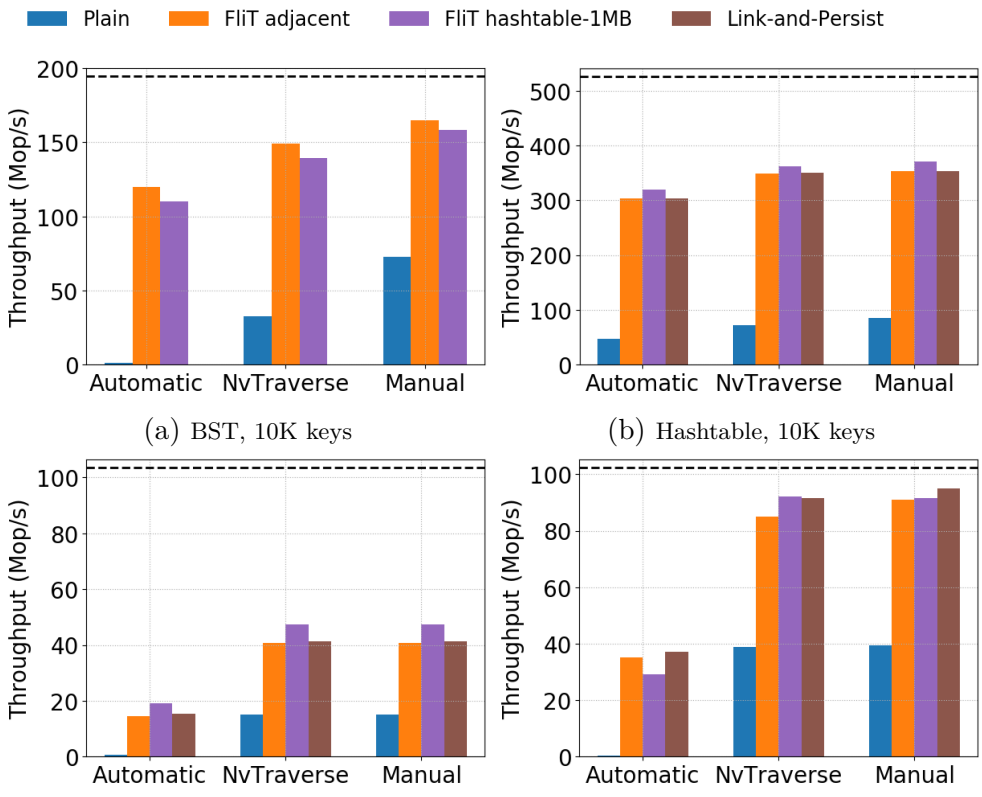
6.6.4 Comparing Durability Methods

Figure 6.6 shows the four implemented data structures, each with their three different methods of durability: automatic, NVtraverse, and manual. When using the FliT library, these methods differ in how many v-instructions they execute; the automatic version only executes p-instructions, the NVtraverse executes many v-loads while traversing the data structure, and the manual version carefully reasons about these individual data structures to make a larger fraction of the instructions be volatile. All plots show 5% updates, and the smaller size of the tested data structure (10K nodes for the scalable data structures, and 128 nodes for the linear linked-list). For each setting, we show a plain implementation, flit-adjacent, flit-HT, and link-and-persist where applicable.

Generally speaking, the link-and-persist method follows the same patterns as the FliT implementations. We note that the more optimized the underlying durability

implementation is, the less it benefits from FliT. However, for all settings, the performance boost from FliT is still substantial; while in the automatic version, FliT boosts throughput by a factor of at least 6.68× (in the hash table), and at most 99.5× (in the skiplist), we still observe an improvement of at least 2.17× when using FliT in all data structures under all durability methods. However, it is also important to note that across the board, the optimized durability methods with FliT outperform the automatic durability method with FliT. Thus, while benefiting less from the FliT library, optimizations that allow using more v-instructions are still useful, and should still be implemented using the FliT library.

Interestingly, while optimized solutions do perform better, the automatic version implemented with the FliT library performs surprisingly well; it significantly outperforms the NVtraverse and manual versions without the FliT library for the BST and hash table, and approximately matches their performance in the linked-list and skiplist.



(a) BST, 10K keys

(b) Hashtable, 10K keys

(c) Linked List, 128 keys

(d) Skiplist, 10K keys

Figure 6.6: Throughput of 44 threads with 5% updates.
Dotted bar represents throughput of the non-persistent version of each data structure.

6.6.5 Effect of updates

In Figure 6.7, we show each data structure with two different sizes, and in each subplot, we vary the update ratio of the workload. These plots are normalized to the throughput

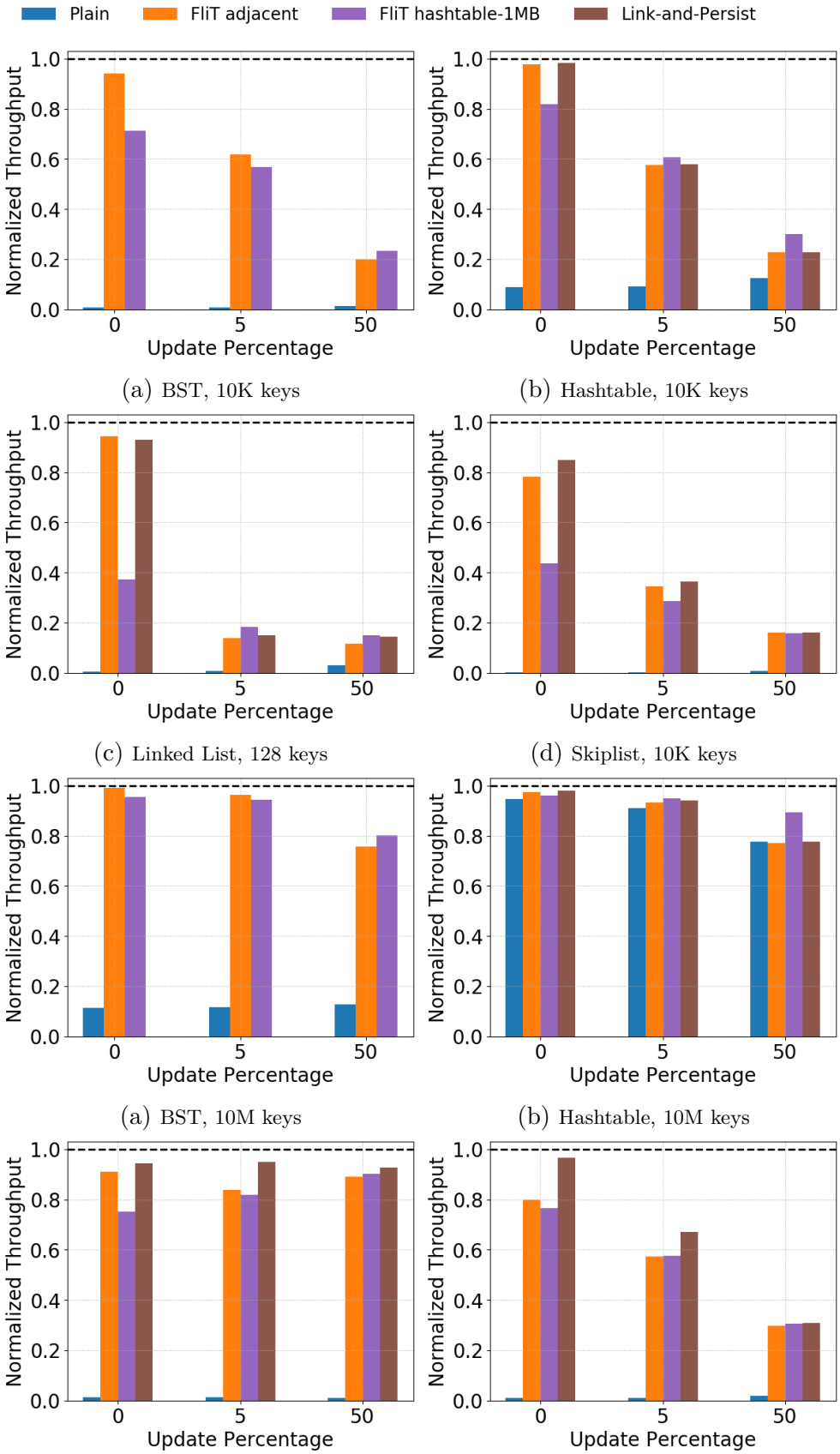
of the non-persistent baseline for each data structure. It is easy to see that the more updates executed, the worse the performance of all persistent versions when compared to the non-persistent baseline. This is expected; flush and fence instructions are executed more the more update operations occur. Note that in 0% update workloads, no flush or fence instructions are executed, other than in initialization and at the end of each operation in the FliT and link-and-persist versions, since loads only ever execute a flush if the location is tagged (and only p-stores can tag memory locations).

Furthermore, in 0% updates, the flit-adjacent and the link-and-persist do better than the hash-table variant. This is because the latter implementations never have to incur an extra cache miss to access the flit-counter, whereas the flit-HT incurs L2 misses every time it accesses the counter.

6.6.6 Comparing FliT and Link-and-Persist

We note that in general, the flit-adjacent and the link-and-persist implementations perform almost identically. This is because they both avoid this extra cache miss when accessing the flit-counter (or flush-bit in the case of link-and-persist). The exception to this rule is in the skiplist, where link-and-persist outperforms the flit-adjacent. This is because flit-adjacent doubles the size of each node. In most data structures, it goes unnoticed, since each node still fits in a single cache line. However, the skiplist node stores many pointers, and thus can overflow a cache line when each word in it is doubled to fit the flit-counter. This problem does not occur with the link-and-persist. However, we note that link-and-persist is not as general, and cannot be implemented with the BST, since this BST algorithm makes use of all bits in each word.

Interestingly, while flit-adjacent/link-and-persist perform best when there are 0% updates, this is not always the case when more updates occur. This is most noticeable in the smaller hash table and linked-list implementations (Figures 6.7b and c). This is because, while in our implementation, we use Intel's *clwb* instruction to perform flushs, which should not invalidate cache lines update flushing, invalidations still occur. Indeed, Intel confirms that clwb, while available for use, is not currently implemented in hardware. Therefore, p-stores in the flit-adjacent incur a cache miss when decrementing the flit-counter, since they always do so after having executed a clwb on that cache line, thereby evicting it from memory. The same thing happens in the link-and-persist implementation, when flipping the flush-bit after having flushed the cache line. Since the flit-HT does not place the flit-counter on the same cache line as its p-store is accessing, decrementing the flit-counter does not incur this cache miss. This effect is less prominent in larger data structures, in which traversing the data structure dominates the overall execution time. Furthermore, we believe that this effect will disappear once Intel implement their non-invalidating flush option in hardware.



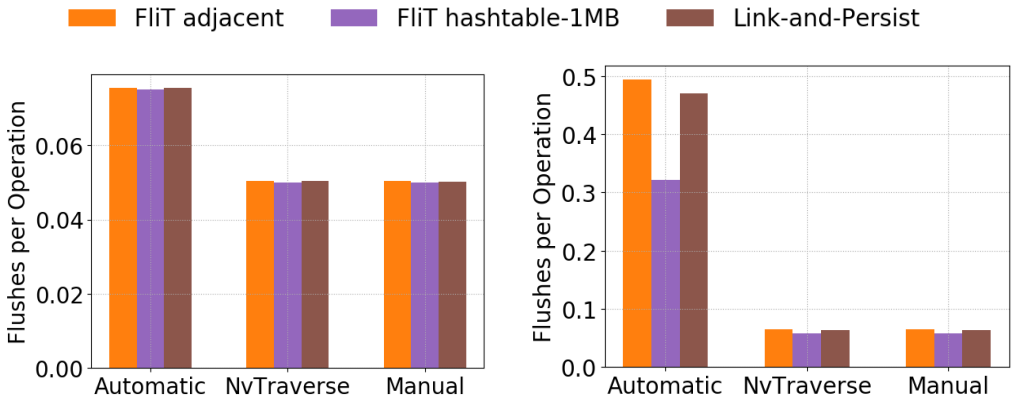
(a) BST, 10K keys (b) Hashtable, 10K keys

(c) Linked List, 128 keys (d) Skiplist, 10K keys

(a) BST, 10M keys (b) Hashtable, 10M keys

(c) Linked List, 4K keys (d) Skiplist, 10M keys

Figure 6.7: Throughput results for 44 threads, automatic, normalized to the throughput of the non-persistent version of each data structure.



(a) Hashtable, 10K keys (b) List, 128 keys

Figure 6.8: Number of flushes per operation, 5% updates

6.6.7 Effect of Data Structure Size on Performance

Like the advantage of the flit-HT over the flit-adjacent, some patterns observed in the smaller data structures become less prominent when run on a larger instantiation of data structure. In particular, for large data structures, the persistent versions almost match the performance of the non-persistent baseline. This is because in large data structures, the runtime of an operation is dominated by the need to traverse long chains of nodes, incurring many cache misses (large BST, Hash Table, and Skiplist do not fit in cache). The overhead of occasional flushs and fences becomes less significant.

6.6.8 Measuring Number of Flushes

Finally, we also measure the number of flush instructions executed in each of our runs. Figure 6.8 shows some representative results. In general, the number of flushs executed is approximately the same across the different FliT implementations. This shows that redundant flushs, resulting from a memory location still being tagged when a p-load accesses it, almost never occur. The exception is seen in the linked-list automatic durability setting, where flit-adjacent and link-and-persist perform significantly more flushs than flit-HT. This again confirms the observation that the clwb instruction does invalidate cache lines; since accessing the flit-counter for decrementing it incurs a cache miss in flit-adjacent, the location remains tagged for a longer period of time. In the automatic setting, in which all loads are p-loads, this means that it is not uncommon for another thread to read the flit-counter while the location is tagged, causing an extra flush.

6.7 Related Work

There have been many papers focusing on finding how to easily and efficiently program for NVRAM. Izraelevitz et al. [59] present the notion of *durable linearizability*, a cor-

rectness condition for persistent data structures. At a high level, durable linearizability requires a data structure to be linearizable despite any number of system crashes that occur during its execution. Izraelevitz el al. also showed how to place flush and fence instructions in linearizable code with acquire-release consistency to guarantee durable linearizability. In this chapter we show how to rewrite this construction in terms of the P-V Interface, and use the FliT library to optimize this implementation. Izraelevitz et al also introduce a weaker correctness guarantee, called buffered durable linearizability, which we do not consider in this chapter.

Researchers have also introduced other correctness criteria for persistence and explored how to support them efficiently [11, 41, 13, 7, 8]. These works consider not only the state of shared memory upon recovery from a system crash, but also whether processes can continue their previous execution. For example, detectability [41], requires that each process be able to find out whether its most recently called operation had completed before a crash. These conditions can be achieved by storing extra metadata beyond what is stored in a non-persisted execution. We believe that using the P-V Interface when designing algorithms for these other correctness criteria can improve performance and portability just as much as it does for durably linearizable implementations.

Many algorithms have been designed for NVRAM in the context of file systems and database indexes [22, 63, 65, 92, 87, 94, 64]. These algorithms are often lock-based, rather than the lock-free data structures that we have compared to in our evaluation. We believe that the P-V Interface, and its implementation in the FliT library, can also be used to enhance such algorithms. Indeed, Lee et al. use a technique (like link-and-persist [32]) in their B-tree algorithm [64]. However, we focused on lock-free data structures in our evaluation since the largest benefits in the FliT library's implementation can be seen in contended workloads, which are less prominent in lock-based algorithms. Still, the P-V Interface captures lock-based algorithms as well, leaving room for optimized solutions by treating private instructions (those inside a lock) separately from shared instructions. Similarly, we believe the P-V Interface can be used to write and reason about efficient persistent transactional memories, a topic that has also drawn significant attention in recent years [30, 84, 55, 10].

Several papers provide other programming interfaces for NVRAM. Mnemosyne [88] provides an interface for using persistent memory through *persistent regions*. Atlas [20] provides persistence for general lock-based programs, but does not capture lock-free algorithms. Gogte et al [44] propose semantics for persistent synchronization-free regions. Other works capture the persistence semantics offered by modern architectures, like Intel-X86 [81] and ARMv8 [83]. This line of work differs from ours in its goals; we propose an interface for easy persistent programming, which can be implemented in hardware using the semantics formalized in these papers.

6.8 Conclusion

In this chapter, we introduce the FliT library, a `C++` library for designing simple and efficient persistent programs for NVRAM. FliT avoids unnecessary flushing by using *flit-counters* to track dirty cache lines.

We test FliT on an Intel machine with Optane DC memory, and demonstrate that the FliT library not only achieves remarkable speedups over even the most optimized persistent data structures, but is also widely applicable.

Our implementation tested two different ways of allocating the flit-counters and mapping them to memory locations. Many other variants are possible, and it would be interesting to see the effects of different counter allocation strategies on algorithms that use FliT. One natural option that we did not explore is to assign one counter per cache line rather than at the granularity of words.

While FliT's default mode makes any linearizable data structure durable with minimal code changes and impressive performance, it also allows further optimizations. In particular, it allows a programmer to specify some instructions that do not need to be persisted. We capture this flexibility with the P-V Interface, which defines the semantics of code in which some memory instructions can remain volatile, while others must be persisted. This interface is language- and architecture-agnostic, and we show it captures persistence behavior in many algorithms; we believe that the P-V Interface can be implemented on different architectures, and would achieve similar performance gains as those achieved by FliT. Note that the algorithm for maintaining flit-counters is more general than the P-V Interface and can be used to implement other persistent interfaces as well.

Chapter 7

Whole Program Persistence

7.1 Introduction

A natural question that arises is whether we can find general mechanisms that would port algorithms for current machines over to the new persistent setting. One approach has been the development of persistent transactional memory frameworks [61, 66, 71, 30].

This can be an effective approach, although it does not handle code between transactions. Another approach to achieve arbitrary persistent data structures is the design of persistent universal constructions [59, 26]. In particular, Cohen *et al.* [26] present a universal construction that only requires one flush per operation, thereby achieving optimality in terms of flushes. However, universal constructions often suffer from poor performance, because they sequentialize accesses to the data structure. Furthermore, universal constructions are only applicable to data structures with clearly defined operations, and cannot apply to a program as a whole. For these reasons, even a seemingly efficient universal construction leaves more to be desired.

In this chapter, we consider simulators that take any concurrent program and transform it by replacing each of its instruction with a simulation that has the same effect. We define *persistent simulations*, which exhibit a tradeoff between their *computation delay*, meaning the overhead introduced by the simulation in a run without crashes, and their *recovery delay*, which is the maximum time the simulation takes to recover from a system crash. Our first result is the presentation of a general persistent simulator, called the *Constant-Delay Simulator*, that takes any concurrent program using Reads, Writes, and compare-and-swap (CAS) operations, and simulates it with constant computation and recovery delays. Our general simulator provides *detectability*, meaning that the program can be continued after recovering from a crash [41].

Theorem 7.1. *Any concurrent program that uses Reads, Writes, and CAS operations can be simulated in the persistent memory model with constant computation delay and constant recovery delay.*

Here, as opposed to previous chapters, we assume the Parallel Persistent Memory (PPM) model [15, 7]. The model consists of P processors, each with a fast local ephemeral memory of limited size, and sharing a large persistent memory. The model allows for each processor to fault and restart (independently or together). On faulting all the processor's state and local ephemeral memory are lost, but the persistent memory remains. On restart, each processor has a location in persistent memory that points to a context from which it loads its registers, including program counter, and restarts. The PPM model decouples concerns of memory consistency after a fault from detectability [41]. Thus, our simulation is a general way to achieve detectability for concurrent algorithms.

The Constant-Delay Simulator is achieved by using a technique called *capsules* [15], which in effect introduces checkpoints on a per processor basis from which the program continues after recovering from a crash. In general, the more capsules there are in a program, the smaller the size of each capsule, and therefore, recovery time decreases. The idea behind the Constant-Delay Simulator is to show that we can have constant sized capsules, and that each capsule can be implemented with constant overhead.

In practice, crashes are relatively rare. It is thus important to minimize the computation delay introduced by a persistent simulator, even at the cost of increased recovery time. In the rest of the chapter, we therefore present optimizations that can be applied to the simulator to decrease the computation delay. The first such optimization is just as general as the Constant-Delay Simulator in that it applies to any concurrent program. The difference is that we use fewer capsules; we show where boundaries between capsules can be removed, creating capsules that are larger (not necessarily constant sized), to arrive at a smaller computation delay but a larger recovery delay. The second optimization applies to a large class of lock-free data structures called *normalized data structures* [86]. In this setting, we show how to further reduce the number of capsules and thus the computation delay.

The idea behind capsules is that the program is broken up into contiguous chunks of code, called *capsules*. Between every pair of capsules, information about the state of the execution is persisted. When a crash occurs in the middle of a capsule, recovery reloads information to continue the execution from the beginning of the capsule. This means that some instructions may be repeated several times. Blelloch *et al.* [15] noted the need for *idempotent* capsules, that is, capsules that are safe to repeat, and gave sufficient conditions to ensure that sequential code is idempotent. However, repetitions of concurrent code can be even more hazardous, as other processes may observe changes that should not have happened. We therefore formalize what it means for a capsule to be correct in a concurrent setting, and show how to build correct concurrent capsules.

To build correct capsules for general concurrent programs, we must be able to determine whether a modification of a shared variable can safely be repeated. This can be problematic, because often, the execution of a process depends on the return value of its accesses to shared memory. A bad situation can occur if a process has already

made a persistent change in shared memory, but crashed before it could persist its operation's return value [15]. Attiya *et al.* [7] consider this problem and define *nesting-safe recoverable linearizability (NRL)*, a correctness condition for persistent objects that allows them to be safely nested. In particular, Attiya *et al.* [7] introduced the *recoverable CAS*, a primitive which ensures that if a compare-and-swap by process p has successfully changed its target, this fact will be made known to p even if a crash occurs.

At a high-level we show how to combine the ideas of capsules with a recoverable CAS in a careful way to achieve our results for programs with shared reads and CASes. To make the simulation work with writes requires some additional ideas. We use a modification of this recoverable CAS primitive in our capsules to ensure that a program can know whether it should repeat a CAS. We show that the recoverable CAS algorithm satisfies a stronger property than NRL, allowing the recovery to be called even if a crash occurs after the operation has terminated. This property is very important for use in capsules, because after a crash, all operations of the capsule must be recovered, rather than just the most recent one.

We test our simulations by applying them to the lock-free queue of Michael and Scott (MSQ) [75], and comparing their performance with two other state-of-the-art implementations: one using the transactional memory framework Romulus [30], and the other a hand-tuned detectable queue, known as the LogQueue [41]. We did not expect general constructions to match the performance of specialized implementations, but it turns out to be very close. The LogQueue outperforms our transformations on 2-6 running threads, but only by an average of 5%; our most optimized transformation even outperforms the LogQueue on 1, and 7-8 threads. In comparison, the original MSQ is usually between 3.57× to 1.9× faster than the LogQueue, showing that the inevitable cost of persistence outweighs the extra cost paid for generality by our transformations. We further show that our optimizied transformation outperform Romulus for the thread counts we tried.

7.2 Preliminaries

We use the Parallel Persistent Memory (PPM) model [15, 7]. It consists of a system of n asynchronous processes $p_1 \dots p_n$. Each process has access to a *persistent* shared memory of size M that it may access with Read, Write, and compare-and-swap (CAS) instructions, as well as a smaller private *ephemeral* memory, which can be accessed with standard RAM instructions. A process can *persist* the contents of its ephemeral memory by writing them into a persistent memory location. The ephemeral memory is explicitly managed; it does not behave like a cache, in that no automatic evictions occur. Memory operations are sequentially consistent, and all memory locations can hold $O(\log M)$ bits.

Each process may *crash* (crash) between any two instructions. Upon a crash, the

contents of a process's private ephemeral memory is lost, but the persistent memory remains unchanged. After a process crash, it *restarts*. On restart, the ephemeral memory can be in an arbitrary state, but the persistent memory is not affected by the crash. To allow for a consistent restart, each processor has a fixed memory location in the persistent memory referred to as its *restart pointer location*. This location points to a context from which to restart (i.e., a program counter and some constant number of register values). On restart this is reloaded into the registers, much like a context switch. Processors can checkpoint by updating the restart pointer. Furthermore a process can know whether it has just crashed by calling a special `crash()` function that returns a boolean flag, and resets once it is called. We call programs that are run on a processor of such a machine *persistent programs*.

A similar model for non-volatile memories allows a shared cache and automatic cache evictions [26, 25, 41, 40, 42, 91]. In this model only the whole system can crash. In Section 7.7, we discuss the difference between the models in more detail, and point out that all our ideas are also valid on the shared cache variant. Throughout the chapter we will use *private cache model* to refer to the PPM model, and *shared cache model* for the shared cache variant.

For our simulations we also consider a standard concurrent RAM (C-RAM) consisting of n asynchronous processes with local registers and a shared memory of size M. As with the PPM we assume the C-RAM supports Reads, Writes and CAS instructions on the shared memory, that the memory is sequentially consistent, and registers and memory locations can hold $O(\log M)$ bits.

7.2.1 Capsules

Our goal is to create concurrent algorithms that can recover their execution after a crash. The main idea in achieving this is to periodically persist *checkpoints*, which record the current state of the execution, and from which we can continue our execution after a crash. We call the code between any two consecutive checkpoints a *capsule*, and the checkpoint itself a *capsule boundary*. At a boundary, we persist enough information to continue the execution from this point when we recover from a crash. This approach was use by Blelloch *et al.* in [15] and similar to approaches by others [33, 69, 27, 67]. An *encapsulation* of a program is the placement of such boundaries in its code to partition it into capsules.

Capsule Implementation. We now briefly discuss how capsule boundaries are implemented.

Recall that stack-allocated local variables, as well as the program counter, are updated in volatile memory, and their new values must be made persistent at each capsule boundary. We therefore keep a copy of the stack in persistent memory. We add to each stack frame a program counter that holds the program location of the last capsule boundary. Furthermore each stack frame maintains two persistent copies of each

stack-allocated variable, as well as a bit indicating which of the copies is currently valid. All these bits are kept together in a single word as a validity mask, and this word is updated at the end of a capsule boundary to atomically indicate which copy of each variable is valid. We therefore assume that each stack frame contains a constant number of variables; in particular, that the number of such variables is not more than the number of bits in a single word.

Heap-allocated variables, including dynamically allocated objects like arrays, are placed immediately in persistent memory, and are handled differently to preserve idempotence (by avoiding write-after-read conflicts as shown in Section 7.6). Any non-constant sized data can be allocated on the heap.

At a capsule boundary within a function, we update the values of the stack-allocated variables that have been changed during the previous capsule. To do so, for each such variable, we check its validity bit, and overwrite its *outdated* copy in persistent memory. After overwriting the outdated copies of each changed variable, we atomically move to the next capsule by writing out a new validity mask (bits of changed variables are flipped) along with the new program counter. We assume the bits and counter fit in one word (or atomically writable object).

When making or returning from a function call we simply change the restart pointer to point to the appropriate stack frame. This way when a processor crashes and restarts, it will restart from the previous capsule boundary.

Capsule Correctness. When executing recoverable code that is encapsulated, it is possible for some instructions to be repeated. This happens if the program crashes in the middle or a capsule, or even at the very end of it before persisting the new boundary, and restarts at the previous capsule boundary. To reason about the correctness of encapsulated programs after a crash, we define what it means for a capsule to be *correct* in a concurrent setting, intuitively meaning that it can be repeated safely.

Definition 7.2.1. An instruction I in an execution history E is said to be *invisible* if E remains legal even when I is removed.

Definition 7.2.2. A capsule C inside algorithm $\mathcal{A}$ is *correct* if:

1. Its execution does not depend on the local values of the executing thread prior to the beginning of the capsule, and

2. For any execution E of $\mathcal{A}$ in which C is restarted from its beginning at any point during C's execution an arbitrary number of times, there exists a set of invisible operations performed by C such that when they are removed, C appears to only have executed once in the low-level execution history.

Definition 7.2.3. A program is *correctly encapsulated* if all of its capsules are correct.

7.3 k-Delay Simulations

We are interested in efficiently simulating arbitrary computations on a reliable concurrent RAM (C-RAM) on a faulty PPM machine. The performance of concurrent algorithms heavily depends on factors like contention, disjoint access parallelism, and remote accesses. While these factors are difficult to theoretically characterize, they are monumental in their effect on the algorithm's performance. Therefore, when creating algorithms for the persistent setting, it is important to be able to preserve the structure of tried and tested efficient concurrent algorithms to their persistent counterparts. We formalize this notion of 'preserving structure' with the definitions of *computation delay*, *recovery delay* and *contention delay*. Roughly, the first refers to the the number of instructions on the PPM required to simulate each instruction on the C-RAM when there is no fault, the second refers to the number of instructions needed to recover from each crash, and the third takes into account the delay caused by concurrent accesses to the same object.

Before talking about delays, we define what we mean by a simulation. We allow for arbitrary computations on the machine that is being simulated, as long as the machine is sequentially consistent. For simplicity we consider computations from beginning to end.

Definition 7.3.1 (Linked Simulation)**.** Consider a concurrent *source* machine S with n processes for which each process executes a sequence of atomic (linearizable) operations, and a concurrent *target* machine T, also with n processes. A *linked simulation* of S on T is a simulation that allows for any computation on S, preserves its behavior, and for each process p the operations taken by T (the history) can always be partitioned, contiguously, so that:

1. there is a one-to-one order-maintaining correspondence between these partitions and operations for p in S, and
2. each partition atomically simulates the corresponding source operation, linearized at some point between the partition's first and last instruction.

We refer to each partition on each process as a *step* of the simulation.

Note that our definition includes all local operations as well as any access to shared objects. In our initial simulation, the operations are machine instructions, but more generally they could include linearizable operations on shared objects. We say a step of a linked simulation is *crash free* if no fault happened on the step's process on T during that step. For a sequence of instructions s on a single process of either the source or target machine let $t(s)$ be the number of instructions, i.e., the time.

Definition 7.3.2 (Computation Delay)**.** A linked simulation has k *computation delay* if for each crash-free step s of the target machine simulating an operation o of the source machine, $t(s) = O(t(o) + k)$.

Ben-David et al. use a similar concept of delay for measuring the efficiency of transactions [12]. Persistent computations are tightly coupled with their recovery mechanisms. When discussing a computation for a persistent setting, it is important to also discuss how it recovers from crashes. Note that, if all processes crash together during a system crash, simply running a concurrent program as is in a persistent setting yields a trivial 1 computation delay simulation of itself; all steps of the program remain exactly the same. However, upon a crash, the entire computation has to be restarted, and all progress is completely lost. Thus, the recovery time of this 'simulation' is unbounded; it grows with the length of the execution. We therefore also formalize the notion of a *recovery delay*; how long it takes for a persistent program on any process to recover from a crash (processes can crash independently).

Definition 7.3.3 (Recovery Delay)**.** A linked simulation of a source machine S on a faulty target machine T has k *recovery delay* if each fault that occurs within a step of the simulation on T adds at most k instructions to the step.

Note that the k for computation delay and recovery delay need not be the same. Furthermore, although k implies a constant, and our simulations guarantee a constant, in general the k could be a function of other parameters of the machine or computation (e.g. input size or number or processes).

A stronger notion of a k computation delay simulation is one in which the amount of contention experienced by an algorithm cannot grow by more than a factor of k either. Accounting for contention helps to capture the structure of an algorithm, since scalability is highly associated with keeping contention as low as possible on all accesses. To be able to discuss contention formally, we follow the definition of contention presented by Dwork *et al.* in [35]; the amount of contention experienced by an operation *op* on object O is the number of *responses* to operations on O received between the invocation and response of *op*[1]. We now extend the definition of computation delay to account for contention.

Definition 7.3.4 (Contention Delay)**.** A linked simulation has k *contention delay* if for any operation during the simulated computation on the source S the contention is C, then the corresponding step in the simulation on the target T has contention at most $k \times C$.

In this chapter, we consider persistent simulations that have small computation, recovery and contention delay. We say that an algorithm is X*-delay free* or if it has c X delay for a constant c, where X is one of computation, contention, or recovery.

[1]For atomic memory operations, which don't have invocations and responses, we can split them into two instructions, one for the invocation and one for the response.

7.4 Building Blocks

7.4.1 Capsule Implementation

We now briefly discuss how to implement the capsule boundaries themselves, and in particular, how we handle persisting the ephemeral variables between capsules. We keep a copy of each ephemeral variable in persistent memory. At a high level, we update the persistent copy of each ephemeral variable with its current value at the end of each capsule, and read these values into ephemeral memory after a crash. However, we must be careful to avoid inconsistencies that could arise if a crash occurs after updating some, but not all, of the persistent copies. Such inconsistencies could occur if ephemeral variables suffer *write-after-read* conflicts in a capsule [15]. In a nutshell, these conflicts occur if a variable can be read and then written to in the same capsule. If the write was persistent and a crash occurs after such a scenario, causing the code repeats itself from the read instruction, then the read sees a different value than it did originally. These conflicts can be avoided if it can be guaranteed that after a new value is persisted, the program will never repeat an earlier instruction that reads it. Note that in general, it is not a problem if an ephemeral variable suffers a write-after-read conflict, since their writes are to ephemeral memory and thus are not persistent. However, since we persist their new values at the capsule boundary, we must be careful not to overwrite their previous value if we might need to use it again.

Thus, for ephemeral variables that are read and subsequently written to in this capsule, we do not write out their new values into their persistent copies right away. Instead, we keep two persistent *write buffers*, along with a persistent bit indicating which one of them is currently *valid*. At the boundary at the end of the capsule, the *invalid* write buffer is cleared, and for each write-after-read conflicted ephemeral variable v, a tuple of the form $(v, newVal)$ is written into it, where $newVal$ is the value of v at the end of the capsule. After writing out all of these tuples to the write buffer, the validity bit for the write buffers is flipped. At the beginning of each capsule, before executing any of its code, the *valid* write buffer is read, and the persistent copy of each ephemeral variable that appears in it is updated with this ephemeral variable's value in the buffer.

Note that persisting the values of ephemeral variables that are either not updated at all, or updated before being read in a given capsule, is not difficult. We don't have to do anything at the capsule boundary for an ephemeral variable that was not updated in this capsule. For ephemeral variables that were updated before being read in this capsule, we can simply copy in their new value into their persistent copy, without going through the write buffer. This is because the current capsule's execution does not depend on the value of such an ephemeral variable. This copying is done before the validity bit of the write buffer is flipped.

7.4.2 Recoverable Primitives

One problem that arises from volatile registers and caches is that the return values of atomic operations can be lost. For example, consider a CAS operation that is applied to a shared memory location. It must atomically read the location, change it if necessary, and return whether or not it succeeded. Return values are stored in volatile registers. If a crash occurs immediately after a CAS is executed, the return value could be lost before the process can view it. When the process recovers from the crash, it has no way of knowing whether or not it has already executed its CAS. This is a dangerous situation; repeating a CAS that was already executed, or skipping it altogether, can render a concurrent program incorrect. In fact, any primitive that changes the memory suffers from the same problem.

This issue was pointed out by Attiya *et al.* in [7]. To address the problem, they present several recoverable primitives, among them a *recoverable CAS algorithm*. This algorithm is an implementation of a CAS object with three operations: read, CAS, and recover. The idea of the algorithm is that when CASing in a new value, a process writes in not only the desired value, but also its own ID. Before changing the value of the object, a process must *notify* the process whose ID is written on the object of the success of its CAS operation. The recovery operation checks this notification to see whether its last CAS has been successfully executed. Attiya *et al.* show that their recoverable CAS algorithm satisfies *nesting-safe recoverable linearizability (NRL)*, intuitively meaning that as long as recovery operations are always run immediately after crashes, the history is linearizable. Attiya *et al.*'s algorithm uses classic CAS as a base object, and assumes that CAS operations are ABA-free. This is easy to ensure by using timestamps.

It turns out Attiya *et al.*'s recoverable CAS algorithm satisfies strict linearizability [2], a stronger correctness property than NRL. The main difference between the two properties is that while NRL only allows recovering operations that were pending when the crash happened, strict linearizability is more flexible. This means that we can define the recovery function to work even on operations that have already completed at the time of the crash. This property is very important for use in the transformations provided in the rest of the chapter, in which we may not know exactly where in the execution we were when a crash occurred. To satisfy strict linearizability, we need to tweak the recoverable CAS algorithm slightly, to include the use of sequence numbers on each CAS. In contrast to Attiya *et al.*, we treat the recovery function as another operation of the recoverable CAS object, whose sequential specification is as follows.

Each *Recover*(i) operation R returns a sequence number *seq* and a flag f with the following properties:

- If $f = 1$, then *seq* is the sequence number of the last successful CAS operation with process id i.

- If $f = 0$, all successful CAS operations before R with process id i have sequence number less than seq.

We also further modify the recoverable CAS algorithm to create a version that has constant recovery time (instead of $O(P)$), and uses less space ($O(P)$ instead of $O(P^2)$). The pseudocode of our version of the recoverable CAS is given in Figure 7.1. This code also shows the tweaks that we do to the original recoverable CAS algorithm of Attiya *et al.*. Note that it implements a recoverable CAS object using $O(1)$ steps for all three operations. The idea is simple: whenever any process executes a CAS on the object, it not only writes in its value, but also its id and the sequence number of its current operation. Before doing that, however, process p_i takes a few set-up steps. Before executing its CAS on the object, it first reads the object's state. The state is always of the form $\langle val, j, seq \rangle$, containing the process id j and sequence number seq of the most recent successful CAS in addition to its value. Once p_i reads this state, it must notify p_j of its recent success, by trying to flip the success flag in $A[j]$ from 0 to 1. However, p_i will only change $A[j]$ if the sequence number written there is the same one that is read. In this way, a notifying process can never overwrite a more recent value in the announcement array. Process p_i can then start its own CAS. To do so, p_i first prepares its announcement slot by writing the sequence number of this new operation in $A[i]$, with a flag set to 0 to indicate that this CAS has not yet been successfully executed. It then proceeds to execute its CAS on the object. To recover from a crash, p_i simply needs to check the object's state to see if its value is written there, and then read its own slot in the announcement array. It is guaranteed to have been notified of its most recent success.

We now prove that the algorithm presented in Figure 7.1 is correct. Our correctness condition is *strictly linearizability* [2]; that is, that all operations are linearizable, and are either linearized before a crash event, or not at all. This condition lets us safely repeat an operation if the recovery says that it didn't happen. When a process reads x and performs a CAS on $A[i]$ for some i, we can view this as a notify operation. In Algorithm 7.1, lines 20 and 21 of *Recover*, as well as lines 11 and 14 of CAS form notify operations. The purpose of the notify operation is to let a process know that its CAS has been successful before overwriting the value. The following lemma captures the key property that we require from notify operations.

Lemma 7.4.1. *Let N be an instance of a notify method that reads $x = \langle *, seq, i \rangle$. Then after N's execution, $A[i] = \langle seq, 1 \rangle$ or $A[i] = \langle seq', * \rangle$, where $seq' > seq$.*

Proof We first show that at the first step of N, the sequence number in $A[i]$ is at least seq. This is because in order for x to be $\langle *, seq, i \rangle$, there must have been a successful CAS operation with sequence number seq by process id i. $A[i]$ is set to $\langle seq, 0 \rangle$ before the linearization point of this CAS operation. $A[i]$ changes only when the process with id i updates $A[i]$ with an increasing sequence number, or when a process calls the *notify*

Figure 7.1: Recoverable CAS algorithm

```
1  class RCas {
2    <Value, int, int> x;  // shared persistent
3    <int, bool> A[P];     // shared persistent
4
5    Value Read() {
6      <v, *, *> = x;
7      return v;
8    }
9
10   bool Cas(Value a, Value b, int seq, int i) {
11     <v, pid, seq'> = x;       // notify
12     if (v != a)
13       return false;
14     CAS (A[pid], <seq', 0>, <seq', 1>);
15     A[i] = <seq, 0>;          // announce
16     return CAS (x,<a, pid, seq'>,<b, i, seq>);
17   }
18
19   <int, bool> Recover (int i) {
20     <v, pid, seq'> = x;       // notify
21     CAS (A[pid], <seq', 0>, <seq', 1>);
22     return A[i];
23   }
24 };
```

method. A *notify* attempt that has a different sequence number than *seq* would fail. Thus, $A[i]$ may only contain $\langle seq, 01\rangle$ or $A[i] = \langle seq', *\rangle$, where $seq' > seq$ after an instance of a *notify* method. ■

Now we are ready to prove that Algorithm 7.1 is strictly linearizable. Its linearization points are also given by the following lemma.

Lemma 7.4.2. *Algorithm 7.1 is a strictly linearizable implementation of a recoverable CAS object with the following linearization points:*

- *Each CAS operation that sees $v \neq a$ on line 12 is linearized when it performs line 11. Otherwise, it is linearized when it performs line 16.*
- *Each Read operation is linearized when it performs line 6.*
- *Each Recover operation is linearized when it returns.*

Proof From the linearization points of the algorithm, we see that if an operation stalls indefinitely before reaching its linearization point, then it will never be linearized. Therefore, proving linearizability is equivalent to proving strict linearizability.

To show that CAS and Read operations linearize correctly, we can ignore operations on A because they do not affect the return values of these operations. At every configuration C, the variable x stores the value written by the last successful CAS operation linearized before C. This is because the value of x can only be changed by the linearization point of a CAS operation. So x always stores the current value of the persistent CAS object.

Each Read operation R is correct because R reads x at its linearization point and returns the value that was read. Each CAS operation C is either linearized on line 11 or line 16. If C is linearized on line 11, then it behaves correctly because x does not contain the expected value at the linearization point of C. Suppose a is the value C expects and b is the value it wants to write. If C is linearized on line 16, then we know that $x = \langle a, j, s' \rangle$ at line 11 of C for some process id j and sequence number s'. If C is successful, then x contained the expected value at the linearization point of C which matches the sequential specifications. Otherwise, we know that x changed between lines 11 and 16 of C. Since this recoverable CAS object can only be used in a ABA free manner, we know that x does not store the value a at the linearization point of C, so C is correct to return false and leave the value in x unchanged.

In the remainder of this proof, we argue that *Recover* operations are correct. Let R be a call to *Recover* by process p_i and let C be the last successful CAS operation by process p_i linearized before the end of R. Let $seq(C)$ be a function that takes a CAS operation and returns its sequence number. We just need to show that R either returns $\langle seq(C), 1 \rangle$ or it returns $\langle s', 0 \rangle$ for some sequence number s' greater than $seq(C)$.

First note that the sequence number in $A[i]$ is always increasing. This is because its sequence number can only change on line 15 of CAS and each process calls CAS with non-decreasing sequence numbers. Thus, the sequence number returned by R is at least $seq(C)$ since $A[i]$ is set to $\langle seq(C), 0 \rangle$ on the line before the linearization point of C. First we show that R cannot return $\langle s', 1 \rangle$ for any $s' > seq(C)$ and then we show that R cannot return $\langle seq(C), 0 \rangle$.

Now we show that R cannot return $\langle s', 1 \rangle$ for any $s' > seq(C)$. R returns the value of $A[i]$, so suppose for contradiction that $A[i] = \langle s', 1 \rangle$ at some configuration before the end of R. Then there must have been a *notify* operation that set $A[i]$ to this value. This notify operation must have seen $x = \langle *, s', i \rangle$ when it performed its first step. This means that a successful CAS by p_i with sequence number $s' > seq(C)$ has been linearized which contradicts our choice of C.

Using Lemma 7.4.1, we can complete the proof by showing that there is a *notify* operation that reads $x = \langle *, seq(C), i \rangle$ and completes before R reads $A[i]$. To show this we just need to consider two cases: either x is changed between the linearization point of C and line 20 of R, or it is not. In the second case, the notify operation performed by R on lines 20 and 21 reads $x = \langle *, seq(C), i \rangle$. This is because R and C are performed by the same process so R starts after C ends. In the first case, some other successful CAS operation must have been linearized after the linearization point

of C and before R reads $A[i]$. Let C' be the first such CAS operation. Then we know that $x = \langle *, seq(C), i \rangle$ between the linearization points of C and C'. Furthermore, in order for C' to have been successful, the read on line 11 must have occurred after the linearization point of C (otherwise C' would not see the most recent process id and sequence number). Therefore, the notify operation on lines 11 and 14 of C' see that $x = \langle *, seq(C), i \rangle$ and this notify completes before R reads $A[i]$ as required. ■

Lemma 7.4.2, plus the fact that each operation only performs a constant number of steps, immediately lead to the following theorem.

Theorem 7.2. *The algorithm presented in Figure 7.1 is a strictly linearizable, contention-delay free and recovery-delay free implementation of a recoverable CAS object.*

The problem of recoverability also applies to atomic write operations; if a process p executes a write that may be seen and possibly overwritten by other processes, it is important that p never repeat its write after a crash, since this can cause an inconsistent state. To handle shared write operations, we reduce the problem to shared CAS operations, and then use the recoverable CAS primitive presented above. We note that often, a shared write can be replaced with a CAS with no effect on the algorithm; that is, an algorithm A that uses CASs and Writes can be simulated by algorithm A_s in which each write operation is implemented with a single CAS. If the CAS fails, this is treated as if the simulated write succeeded, but was overwritten before any other process saw the value. However, in some cases, replacing a write with a CAS does not have the same effect. Intuitively, this could occur if in algorithm A, the write races with a CAS operation on the same location; if the write happens before the CAS, then the CAS would fail, and the value of the write would not be overwritten.

To handle this case, we present an algorithm that gets rid of races between CAS and write operations on the same location. At a high level, our algorithm does this by adding a level of indirection, leading the racy CAS and Write to actually access different locations. We describe the CAS-Write algorithm in detail in Section 7.6.3, and show that it is computation-delay free. Once the CAS-Write algorithm is applied, all writes can be replaced with CAS operations, which can be recovered with the recoverable CAS algorithm presented above. Hence, all of our presented simulations apply not only to programs that use CAS and read operations on shared memory, but also to those that additionally use write operations.

7.5 Persisting Concurrent Programs

One way to ensure that a program is tolerant to crashes is to place a capsule boundary between every two instructions. We call these *Single-Instruction* capsules. Can this guarantee a correctly encapsulated program? Even with single-instruction capsules, maintaining the correctness of the program despite crashes and restarts is not trivial.

In particular, a crash could occur after an instruction has been executed, but before we had the chance to persist the new program counter at the boundary. This would cause the program to repeat this instruction upon recovery.

Trivially, if the single instruction I in a capsule C does not modify persistent memory, then I is invisible, and thus C is correct. But what if I does modify persistent memory? Using the algorithm from Section 7.6.3, we can implement any writable CAS object using just CAS, so we only need to worry about CASs to shared memory and non-racy writes to private persistent memory. A private persistent write is invisible because the process simply overwrites the effect of its previous operation, and no other process could have changed it in between. So, we only have CASs left to handle. This is where we employ the recoverable CAS operation.

We replace every CAS object in the program with a recoverable CAS. We show that it is safe to repeat a recoverable CAS if we wrap it with a mechanism that only repeats it if the recovery operation indicates it has not been executed; any repeated CAS will become invisible to the higher level program. When recovering from a crash, we simply call a `checkRecovery` function, that takes in a sequence number, and calls the `Recover` operation of the recoverable CAS object. The `checkRecovery` function returns whether or not the CAS referenced by the sequence number was successful. If it was, then we do not repeat it, and instead continue on to the capsule boundary. Otherwise, the CAS is safe to repeat. Pseudocode for the `checkRecovery` function is given in Figure 7.2.

With this mechanism to replace CAS operations, single-instruction capsules are correct. The formal proof of correctness is implied by the proof of Theorem 7.4, which we show later.

Figure 7.2: Check Recoverable CAS

```
1 bool check_recovery (RCas X, int seq, int pid) {
2   <last, flag> = X.Recover(pid);
3   if (last >= seq && flag == true)
4     return true;
5   else
6     return false;
7 }
```

We now show that this transformation applied to any concurrent program C is a contention-delay free simulation of C.

Theorem 7.3. *For any concurrent algorithm A written in the C-RAM model, if A_s is the program resulting from encapsulating A using single-instruction capsules, then A_s is a c-contention-delay, c'-recovery-delay simulation of A, where c and c' are constants.*

To prove the theorem, we first show a useful general lemma, that relates the way a simulated object is implemented to the contention-delay of a simulation algorithm.

Lemma 7.5.1. *Let A_s be a k-computation-delay simulation of A. If for every two base objects O_1 and O_2, the set of primitive objects used to implement O_1 is disjoint from the set used to implement O_2 in A_s, then A_s is a k-contention-delay simulation of A.*

Proof Consider an execution E of A in which an operation op by process p on object O_1 experiences $k * C$ contention. Since O_1 is implemented with primitive objects that are not shared with any other object, all contention experienced by op must be from other operations that are accessing O_1. Note that each step by another process accessing O_1 can cause at most one contention point for op. Thus, there must be at least $k * C$ steps by other processes on the primitive objects op is accessing within op's interval. Since A is a k-delay simulation of A', and O_1's primitive objects are not shared with any other base object, there must be at least C other accesses of O_1 that are concurrent with op. So, E must map to an execution E' of A' in which all C of these accesses to O_1 happen before op's corresponding access, but after the last operation of p. Therefore, in E', op experiences at least C contention. ■

Proof of Theorem 7.3. Since each recoverable CAS and each capsule can be used to recover in constant time, it is easy to see that A_s has constant recovery delay. We implement each base object O of A by calling the operations of O, followed by a capsule boundary. For CASs, we implement it by replacing the CAS object with a recoverable CAS object and also calling a capsule boundary. Because both the recoverable CAS algorithm and the capsule boundary take a constant number of instructions, we have shown that our transformation is a k-computation-delay simulation of A. Furthermore, each recoverable CAS object uses primitive objects that are unique to it, and not shared with any other object. Note that while the capsule boundary does use primitive objects that are shared among other capsule boundaries, the capsule boundaries are in fact local operations, since each process uses its own space for persisting the necessary data. So capsule boundaries do not introduce any contention. The rest of the proof therefore follows from Lemma 7.5.1. ■

7.6 Optimizing the Simulation

Although capsule boundaries only consist of a constant number of uncontended instructions, they can still be expensive in practice, as they require persisting several pieces of data and use up to two fence instructions. We now discuss how to reduce the number of required capsule boundaries in a program, while still maintaining correctness. Fewer capsule boundaries means more instructions per capsule, so more progress could be lost due to a crash.

In this section, we focus on programs that use CASs and reads as their mechanisms for accessing shared data. Using our implementation of writable CAS objects (presented in Section 7.6.3), we can extend this to programs that use writes as well, thereby covering many concurrent programs.

7.6.1 CAS-Read Capsules

We begin by showing that at a high level, as long as there is only one CAS operation per capsule, and this operation is the first of the capsule, the program remains correctly encapsulated.

We note that Blelloch *et al.* [15] comprehensively showed how to place capsule boundaries in non-racey persistent code to ensure idempotence. Their guideline is to create capsules that avoid *write-after-read* conflicts (See Section 7.4.1). Therefore, in addition to the capsule boundaries dictated by instructions on shared memory as outlined above, we also place a capsule boundary between a read of a persistent private location in memory and the following write to that location. However, note that we don't always add this extra boundary; if a capsule begins with a persistent write of a private variable v, any number of reads and writes to v may be executed in the same capsule, since there is no write-after-read conflict.

Another potentially dangerous situation arises when we have a branch that depends on a shared memory read and the two paths after the branch write to different persistent memory locations. If this is all placed within a single capsule, then crashing could cause an incorrect execution where both persistent memory locations are written to. For example, there could be a crash immediately following the write to one location, and after recovering, the process could see a different shared value and decide to follow the other branch, leading to a write on the other persistent memory location.

We call this construction a *CAS-Read* capsule. We also allow for capsules that do not modify any shared variables at all. We call such capsules *Read-Only* capsules. Intuitively, all read operations are always invisible, as long as their results are not used in a persistent manner. So, a capsule that has at most one recoverable CAS operation, followed by any number of shared reads, is correct.

Note that we assume that every process has a sequence number that it keeps locally, and increments once per capsule. At the capsule boundary, the incremented value of the sequence number is persisted (along with other ephemeral values, like, for example, the arguments for the next recoverable CAS operation). Therefore, all repetitions of a capsule always use the same sequence number, but different capsules have different sequence numbers to use.

We now describe in more detail how to use the recovery function of the recoverable CAS object. This assumption is realistic, since in most real systems, there is a way for processes to know that they are now recovering from a crash. We use the `crash()` function to optimize some reads of persistent memory— if we are recovering from a crash, we read in all ephemeral values we need for this capsule from the place where the previous capsule persisted them. Otherwise, there is no need to do so, since they are still in our ephemeral memory. We show pseudocode for the CAS-Read capsule in Algorithm 7.3. Read-Only capsules are a subset of the code for CAS-Read capsules.

We now show that the CAS-Read capsule is correct. This fact trivially implies that

Figure 7.3: CAS-Read Capsule

```
1  if (fault()) {
2    *Read all vars persisted by the
3    previous capsule into ephemeral memory.*
4    seq = seq+1;
5    flag = check_recovery (X, seq, pid);
6    if (!flag) { //Operation 'seq' wasn't done
7      c = X.Cas(exp,new,seq, pid);
8    } else {
9      c = 1;
10   }
11 } else {
12   seq = seq+1;
13   //exp and new are from prev capsule
14   c = X.Cas(exp,new,seq,pid);
15 }
16 *Any number of persistent memory reads
17 and ephemeral memory operations*
18 *Writes to private persistent memory are
19 allowed under certain conditions*
20 capsule_boundary()
```

Read-Only capsules are correct as well, so we do not prove their correctness separately. We wrap up this section by showing that a transformation that applies CAS-Read and Read-Only capsules is a c-contention-delay simulation for constant c. Intuitively, removing capsule boundaries can only improve the contention delay.

Note that the definition of correctness is with respect to an algorithm that contains the capsule. Here, we prove the claim in full generality; we show that this capsule is correct in *any* algorithm that could use it. For this, we argue that its repeated operations are invisible in any execution, despite possible concurrent operations. Note that this implies correctness for any context in which the capsule might be used.

Theorem 7.4. *If C is a CAS-Read capsule, then C is a correct capsule. We also require that each process increments the sequence number before calling CAS.*

Proof Consider an execution of C in which the capsule was restarted k times due to crashes.

First note that if a crash occurred, then the capsule never uses any of the local variables before overwriting them. Therefore, its execution does not depend on local values from previous capsules. This includes the sequence number for the capsule, which must have been written in persistent memory before the capsule started, and is therefore the same in all repetitions of the capsule.

Note that in all but the first (partial) run of the capsule, the `crashed` function must return true. Furthermore, note that the code only repeats `X.Cas()` if `checkRecovery`

returns false. Due to the correctness of the recovery protocol, this happens only if each earlier operation with this capsule's sequence number has not been executed in a visible way. This means they are either linearized and invisible, or they have not been linearized at all. The partial executions have not been linearized cannot become linearized at any later configuration because X is strictly linearizable. Therefore the `X.Cas()` call is only ever repeated if the previous calls that this capsule made to it were invisible, so all but the last instance of the `X.Cas()` operations executed are invisible. Furthermore, since `X.Cas()` is a strictly linearizable implementation of CAS, the effect of instances together is that of a single CAS. Note that the rest of the capsule is composed of only invisible operations; the recovery of an object is always invisible, as are Reads and local computations. ■

Theorem 7.5. *A program that uses only CAS-Read, Read-Only, and Single-Instruction capsules is correctly encapsulated, and is a contention-delay-free simulation of its underlying program.*

Proof Since CAS-Read, Read-Only, and Single-Instruction capsules are all correct (corollary of Theorem 7.4), by definition, a program that uses only these capsules is correctly encapsulated. Furthermore, since by Theorem 7.3, a program encapsulated with single-instruction capsules only is a constant-contention-delay simulation of its underlying program, and CAS-Read and Read-Only capsules use strictly less instructions, programs encapsulated with these capsules are also contention-delay free simulations. ■

7.6.2 Normalized Data Structures

Timnat and Petrank [86] defined *normalized data structures*. The idea is that the definition captures a large class of lock-free algorithms that all have a similar structure. This structure allows us to reason about this class of algorithms as a whole. In this section, we briefly recap the definition of normalized data structures, and show optimizations that allow converting normalized data structures into persistent ones, with less persistent writes than even our general Low-Computation-Delay Simulator would require. We will show two optimizations; one that works for any normalized data structure, and one that is more efficient, but requires a few more (not-too-restricting) assumptions about the algorithm.

Normalized lock free algorithms use only CAS and Read as their synchronization mechanisms. At a high level, every operation of a normalized algorithm can be split into three parts. The first part, called the *CAS Generator*, takes in the input of the operation, and produces a list of CASs that must be executed to make the operation to take effect. The second part, called the *CAS Executor*, takes in the list of CASs from the generator, and executes them in order, until the first failure, or until it reaches the end of the list. Finally, the *Wrap-Up* examines which CASs were successfully executed by the executor, and determines the return value of the operation, or indicates that

the operation should be restarted. Interestingly, the Generator and Wrap-Up methods must be *parallelizable*, intuitively meaning that they do not depend on a thread's local values, and can be executed many times without having lasting effects on the semantics of the data structure.

Optimizations Our Low-Computation-Delay Simulator works for all concurrent algorithms that access shared objects using only read, write and CAS. In particular, works for normalized data structures. However, we can exploit the additional structure of normalized algorithms to optimize the simulation.

Note that placing capsule boundaries around a parallelizable method yields a correct capsule. This is implied from the ability of parallelizable methods to be repeated without affecting the execution, which is exactly the condition required for capsule correctness. The formal definition of parallelizable methods is slightly different, but a proof that this definition implies capsule correctness appears in [24]. Thus, there is no need to separate the code in parallelizable methods into several capsules. Furthermore, there is also no need to use recoverable CAS for some of the CAS operations performed by parallelizable methods; for normalized data structures, we can simply surround the CAS generator and the Wrap-Up methods in a capsule, and do not need to alter them in any other way.

All that remains now is to discuss the CAS executor, which simply takes in a list of CASs to do, and executes them one by one. No other operations are done in between them. Note that we can convert CAS operations to use the recoverable CAS algorithm, and then many consecutive CASs could be executed in the same capsule, as long as they access different objects. In the case of normalized data structures, however, we do not have the guarantee that the CASs all access different shared objects. Therefore, we cannot just plug in that capsule construction as is. However, we note another quality of the CAS executor that we can use to our advantage: the executor stops after the first CAS in its list that fails. Translated to the language of persistent algorithm, this means that we do not actually need to remember the return values of each CAS in the list separately; we only need to know the index of the last successful CAS in the list. Fortunately, the recovery operation of the recoverable CAS algorithm actually gives us exactly that; it provides the sequence number of the last CAS operation that succeeded. Therefore, as long as we increment the sequence number by exactly 1 between each CAS call in the executor, then after a crash, we can use the recovery function to know exactly where we left off. We can then continue execution from the next CAS in the list. Note that if the next CAS in the list actually was executed to completion but failed before the crash, there is no harm in repeating it. We simply execute it again, see that it failed, and skip to the end of the executor method.

For a recoverable CAS to work correctly, all CASs to that object must use the recoverable CAS algorithm. Whenever a generator or wrap-up method performs CAS on an object that could also be modified by a CAS-executor, it must use recoverable

CAS instead of classic CAS. This is because even though the generator or wrap-up method never needs to recover a CAS's results, it is still important to notify other processes of the success or failure of their last CAS.

We now discuss a method that allows removing the capsule boundary between the executor and the wrap-up. We argue that as long as we can recover the arguments and results of each executor CAS, it is safe restart the execution from the beginning of the executor. Suppose a combined executor plus wrap-up section faults and repeats multiple times, we first argue that as long as the wrap-up part cannot overwrite the notification of any CAS in the cas-list, then in every repetition, the executor returns exactly the same index in the cas-list. Recall that we assume CAS operations are ABA-free in the original program (i.e. the object cannot take on a previous value) and that each process calls CAS using a value it previously read as the expected value. This means that if a CAS operation fails the first time, then the same CAS operation will also fail the second time. Furthermore, since we do not overwrite the result of the executor CASs outside the executor, it can always use the recovery properly to know which CAS in the list was the last to succeed. Therefore the index returned by the CAS-executor will be the same across all repetitions. This means that the executor plus wrap-up capsule basically behaves as if there were a capsule boundary between the two methods. Since the wrap-up method is parallelizable, we know this capsule is correct.

To remove the capsule immediately after the executor, we need to ensure that the wrap-up does not corrupt the ability of the recoverable CAS to tell whether the most recent *executor CAS* on each object succeeded. If the wrap-up does not access any CAS location accessed by the executor, this property is guaranteed. However, if there is a CAS in the wrap-up that accesses the same location as some CAS in the executor, we can still ensure that we can recover. Let C_w be such a CAS in the wrap-up executed by process p. Note that C_w never needs to use the recovery function for itself; since the wrap-up is parallelizable, it is always safe to repeat C_w after a crash. Therefore, when C_w is executed using a recoverable CAS, it can leave out its own ID and sequence number, so that other processes do not notify p. Thus, the previous notification that p received (i.e. a notification about p's executor CAS on the same object) remains intact.

Notice that if we have two parallelizable methods, A and B, next to each other, we can actually put them in a single capsule as long as the inputs to A and B are the same whenever the capsule restarts. Since A and B have the same inputs, we know by parallelizability that A and B each appear to execute once regardless of how many times the capsule restarts. Also A must appear to finish before B because there was a completed execution of A before any invocation of B. Therefore, this capsule appears to have executed only once.

So, we can avoid an additional capsule boundary between the current iteration's wrap up method and the next iteration's generator method, as long as we now use the same notification trick in the CASs of the generator as well. So as long as there are

capsule boundaries before and after each call to a normalized operation, we only need one capsule boundaries in each iteration of the main loop: only before the executor. We call this simulation the *Persistent Normalized Simulator*. The details of our encapsulation are shown in Algorithm 7.4. The results of this section are summarized in Theorem 7.6.

Theorem 7.6. *Any normalized data structure N can be simulated in a persistent manner with constant-contention-delay using one capsule boundary per repetition of the operation.*

Figure 7.4: Persistent Normalized Simulator

```
1  result_type NormalizedOperation (arg_type input) {
2    do {
3      cas_list = CAS_Generator(input);
4      capsule_boundary();
5      if (fault()) {
6        cas_list = read (CAS_list);
7        seq = read(seq);
8      }
9      idx = CAS_Executor(cas_list, seq);
10     <output, repeat> = Wrap_Up(cas_list, idx);
11   } while (repeat == true)
12   return output;
13 }
14
15 int CAS_Executor(list CASs, int seq) {
16   //CASs is list of tuples <obj, exp, new>
17   bool faulted = fault();
18   bool done = false;
19   for (i = 0; i < CASs.size(); i++) {
20     if (faulted) {
21       done = check_recovery(CASs[i].obj, seq, p);
22     }
23     if (!done) {
24       if (!RCAS(CASs[i]))
25         return i;
26     }
27     seq++;
28   }
29   return CASs.size();
30 }
```

7.6.3 Handling Write-CAS races

Recall from Section 7.4.2 that it is often possible to replace shared variable writes with a read followed by a CAS without impacting the correctness of the algorithm. In this section, we handle the few cases where this is not possible by implementing an array of M writable CAS objects with constant-computation-delay using $O(M + P^2)$ regular CAS objects. The idea is to use a level of indirection to separate out the racy writes and CASes to different memory locations and then replace the non-racy write operations with CAS.

Agazadeh, Golab and Woffel [1] presented a general technique which can be used to implement a writable CAS object with constant step complexity using $O(P^2)$ CAS objects. Our algorithm is based on their technique. At a high level, their algorithm maintains an array `B` of $O(P^2)$ CAS objects and a pointer `Ptr` which stores the index of the currently active CAS object. High-level Read() and CAS() operations read `Ptr` and apply their corresponding low-level operation to `B[Ptr]`. A high-level Write(v) operation looks for a reuseable location `B[j]` and writes v into `B[j]` with a CAS (this CAS is guaranteed to succeed). Then it tries to write j into `Ptr` with a CAS. If it is successful, the write operation is linearized at this CAS. Otherwise, it must have been interrupted by the successful CAS of some other write operation, so this write operation can linearize immediately before that successful CAS. The algorithm for efficiently finding the reusable CAS object `B[j]` is described later on.

We extend these ideas to implement M writable CAS objects by increasing the size of `B` to $M + \Theta(P^2)$ and turning `Ptr` into an array of size M. The value of the j-th simulated object will be stored in `B[Ptr[j]]`. To find reusable locations in `B`, each process maintains a set of $\Theta(P)$ locations that it owns. These sets are disjoint and do not contain any location that is pointed to by an element of `Ptr`. Each write operation picks a reusable location from the set that it owns and if it successfully updates some pointer `Ptr[j]`, then it loses ownership of that location and gains ownership of the location that was previously in `Ptr[j]`.

Next, we explain how to locate reusable locations in `B[i]`. We first describe an implementation with amortized constant step complexity, and then breifly explain how to deamortize it. Just like in Agazadeh *et al.*'s algorithm, we use a variant of Hazard Pointers [73] to keep track of which CAS objects are in use. To compute the reusable locations, process p_i scans the announcement array and makes a list L_i of all the indicies in `B` it owns which were not announced. We use helping to ensure that the locations in L_i are safe to reuse. When performing a Write() operation, p_i allocates from its local list L_i until it runs out. Then it has to compute a new list. If each process owns $2P$ locations in `B[i]`, then the new list can be computed in $O(P)$ time and this happens at most once every P Write() operations. Therefore this algorithm has constant amotrized complexity. The pseudo-code can be found in Figures 7.5 and 7.6 in the appendix. To deamortize, each process can maintain two lists of reusable locations and whenever it

allocates from one list, it performs a constant amount of work towards populating the other.

It is also possible to modify this algorithm to support the allocation of new writable CAS objects in case the number of writable CAS objects needed is not known in advance.

7.7 Practical Concerns

Implementing Capsules in Real Programs. When interpreting real program code in terms of our model, we assume that all stack-allocated local variables, including the program counter, are updated in ephemeral memory, and that heap-allocated variables are created and updated directly on persistent memory. That is, the ephemeral variables in our model map to stack-allocated variables, and we add to each stack frame the program counter at the last capsule boundary. We therefore keep a copy of the stack in persistent memory, and use a doubling trick similar to the one discussed in Section 7.4.1 to handle stack-allocated variables with write-after-read conflicts.

Instead of using write buffers to only duplicate the variables that have write-after-read conflicts, another possibility is to duplicate all stack-allocated variables, that is, have *two* persistent copies of each stack frame, and toggle between the two, copying all values over from one to the other at the end of every capsule. This technique seems wasteful, but is very robust, and works well when the number of variables is small.

Note that from the control flow graph of the program, we can know which variables suffer write-after-read conflicts in a capsule. If these variables stay the same across many capsules, we can duplicate only them, and have only one copy for the rest of the variables. A validity bit can indicate which copy of the duplicated variables is valid, similarly to how the write buffers work. The validity bit must be flipped after all other changes are made, and flipping it represents the atomic transition between capsules. This technique allows us to use space only as necessary, as the write buffer technique does, but reduces the number of times values are copied over to different locations. If between two capsules the variables with write-after-read conflicts change, we can use the copy-all technique to remap the memory usage to have copies of the correct variables.

Note also that generally, flushes are done at the granularity of a cache line. Thus, if all stack-allocated variables (with the necessary duplications) fit in a single cache line, we can simply ensure that they are all updated before the validity bit is flipped, and only need a single fence to commit the capsule.

Since there may be many heap-allocated variables, we do not handle them in the same way as stack-variables. Instead, they are directly written on persistent memory. Therefore, we must make sure that repeated code does not corrupt their values by setting capsule boundaries in between instructions that may harm each other, just like we do for the shared memory instructions.

Figure 7.5: Implementing M writable CAS objects using regular CAS objects. Code for proccess p_i

```
1  Object B[M+2*P*P];
2  int Ptr[M];
3  int free_ptr_i;
4
5  Value read(int obj_id) {
6    int idx = getObjectIdx(obj_id);
7    return B[idx].read();
8  }
9
10 Value CAS(int obj_id, Value old, Value new) {
11   int idx = getObjectIdx(obj_id);
12   return B[idx].CAS(old, new);
13 }
14
15 Value Write(int obj_id, Value new_val) {
16   int new_ptr = free_ptr_i;
17   // This CAS cannot fail
18   B[new_ptr].CAS(B[new_ptr], new_val);
19   int old_ptr = Ptr[obj_id];
20   if (Ptr[obj_id].CAS(old_ptr, new_ptr))
21     free_ptr_i = recycle(old_ptr);
22 }
23
24 Struct Announcement {
25   int index;
26   int seq;
27   bool help;
28 } A[P];
29
30 Struct Status {
31   int pid;
32   bool announced;
33 } status[M+2*P*P];
34
35 List free_list_i;
36 List retired_list_i;
37
38 Object getObjectIdx(int obj_id) {
39   int seq = A[i].seq+1;
40   // This CAS cannot fail
41   A[i].CAS(A[i], <obj_id, seq, 1>);
42   int ptr = Ptr[obj_id];
43   A[i].CAS(<obj_id, seq, 1>, <ptr, seq, 0>);
44   return A[i].index;
45 }
```

Figure 7.6: Recycle operation of a writable CAS object

```
1  int recycle (int ptr) {    // amortized version
2    retired_list_i.push(ptr);
3    // This CAS cannot fail
4    status[ptr].CAS(status[ptr], <i, 0>);
5    if (free_list_i.empty()) {
6      List ann_list;
7      for (int j = 0; j < P; j++) {
8        Announcement a = A[j];
9        if (a.help) {
10         int ptr = Ptr[a.index];
11         A[j].CAS(a, <ptr, seq, 0>);
12       }
13       a = A[j];
14       if (!a.help && status[a.index] == i) {
15         ann_list.push(a.index);
16         // This CAS cannot fail
17         status[a.index].CAS(status[a.index], <i, 1>);
18       }
19     }
20     List new_retired_list;
21     for (retired_ptr in retired_list_i) {
22       if (status[retired_ptr].announced)
23         new_retired_list.push(retired_ptr);
24       else
25         free_list_i.push(retired_ptr);
26     }
27     retired_list_i = new_retired_list;
28     for (ann_ptr in ann_list)
29       // This CAS cannot fail
30       status[a.index].CAS(status[a.index], <i, 0>);
31   }
32   return free_list_i.pop();
33 }
```

Shared vs Private Model. Recall that in the private cache model, we assume that the only way for processes to communicate is through persistent memory (i.e., all shared memory is persistent). Furthermore, the volatile memory is explicitly managed, and no automatic flushes occur. A similar model has been considered in several other works [26, 41, 59]. On the other hand, in a shared cache model, processes communicate through objects in volatile memory rather than persistent memory. The values in these objects are persisted when the program issues an explicit flush instruction or when a cache line is evicted automatically. The shared model is more faithful to current cache coherent machines, while the private model helps to abstract away machine-specific flushes.

A simple transformation to convert an algorithm for the private cache variant into an algorithm that works in the shared cache variant was presented by Izraelevitz *et al.* in [59]. This transformation applies to programs written in the release-consistency memory model. After every load-acquire, it adds a flush and a fence. Before every store-release, it adds a fence, and after every store and store-release, it adds a flush. When transforming an algorithm from the shared cache model to a model in which cache lines may be automatically evicted, one needs to consider not only shared variables, but also local ones. Inconsistencies can occur in code that might repeat changes to persisted local variables (due to a crash). This can be handled again by avoiding write-after-read conflicts [15], as is discussed for heap-allocated variables in Section 7.6.

Compiler. Note that we treat shared variables differently from private ones; private heap-allocated variables may be written to many times in a single capsule, but this is disallowed for shared variables. It is thus important that the compiler be able to distinguish between these two types of variables. One way to achieve this is to have the user annotate shared variables. We also need annotations to allow the compiler to determine which of our constructions should be used; for normalized data structures, we assume the user can annotate the generator, executor, and wrap-up sections. We also assume that for simple cases, the compiler can determine if a variable is ever used again. Therefore a capsule boundary only needs to persist the variables that may be used in the future.

Constant Stack Frames. Recall that for the stack-allocated variables, we assume that there is only a constant number of them (around the same as the number of bits in a word) in each stack frame. This is important to be able to atomically update the validity mask of the variables in each capsule boundary, as in Section 7.2.

CAS. Also recall that the recoverable CAS algorithm requires storing not only the value, but also an ID and sequence number in each CAS location. This can be achieved by using a double-word CAS, which is common in modern machines.

Flushes. In a capsule boundary, if all the local variables fit on the same cache line, then we only need one fence for the capsule since the cache line gets flushed all at once in the private model. On the other hand, note that on modern machines with automatic cache evictions, writing all variables on one cache line does not guarantee atomicity,

since an eviction can happen part way through updating the cache line. However, we can still assume, as is done in [25], that writes to the same cache line are flushed in the order they are written. Intuitively, this is because on real machines, the following three properties generally hold: (1) total store order (TSO) is preserved, (2) individual words are written atomically, and (3) each cache line is evicted atomically.

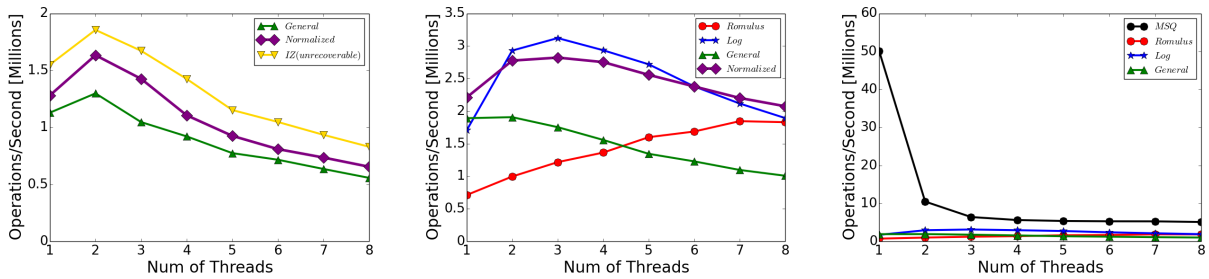
Memory mapping. Note that for our algorithms to recover, we assume that after a crash, the each process can always find the memory in which the capsule boundary stored information. This requires persisting the page table. We assume that this is done by the operating system. We further assume that each process is assigned the same virtual address space as it was before the crash. These two assumptions together ensure that each process receives the same *physical* address space before and after a crash. More details about the virtual to physical mapping for persistent memory is given in [20].

7.8 Experiments

We measured the overhead of our general and normalized data structure transformations by applying them to the lock-free queue of Michael and Scott [75]. We also compare against Romulus [30], a persistent transactional memory framework, and LogQueue [41], a hand-tuned persistent queue. Both Romulus and LogQueue provide durability [59] and detectability [41] in the shared cache model. We use the shared cache model for our experiments because it's closer to the machine that we test on. But this means we need to somehow translate our detectable queues from the private cache model to the shared cache model. We consider two different ways of doing this translation: automatically by Izraelevitz *et al.*'s durability transformation [59] or manually by hand.

We ran our experiments on Amazon's EC2 web service with Intel(R) Xeon(R) Platinum 8124M CPU model (8 physical cores, 2-way hyperthreading, 3GHz and 25MB L3 cache), and 16GB main memory. The operating system is Ubuntu 16.04.5 LTS. Just like in [41, 20, 30], we assume that the cost of flushing on this system will be similar to what we will see in real NVM systems.

All functions were implemented in C++ and compiled using the g++ 5.4.0 with -O3. We only measured the performance on 1-8 threads (each on a separate core), as queues are not a scalable data structure. As in previous work [75, 41], we evaluated the performance with threads that run enqueue-dequeue pairs concurrently. In all the experiments we present, the queue is initiated with 1M nodes; however, we also tested on a nearly-empty queue and verified that the same trends occur. The *flush* operation consists of two instructions: *clflushopt*, and *sfence*. *Clflushopt* has store semantics as far as memory consistency is concerned. It guarantees that previous stores will not be executed after the execution of the *clflushopt*. According to Intel, flushing with *clflushopt* is faster than executing flushes using *clflush* [54]. We also tried using *clwb*



a) Flushes according to Izraelevitz Construction

b) Comparing our queues with manual flushes to prior work

c) Comparing persistent queues to original MSQ

Figure 7.7: Throughput of persistent and concurrent queues under various thead counts. Normalized and General are the result of our transformations.

instead of *clflushopt* and found no difference in performance. The *sfence* instruction guarantees that the *clflushopt* instruction is globally visible before any following store instruction in program order becomes globally visible. We omitted some fences when the ordering of the flushes is not important. All the presented results use the *recoverable CAS algorithm* that was proposed by Attiya *et al.* [7]. In our experiments, their algorithm performed slightly better than ours and thus was the one that was presented.

Each of our tests were run for 5 seconds and we report the average throughput over 10 runs. In general, queues that contain less flushes perform better, which is consistent with what we expected.

In our experiments, our goal is to understand the overhead general programs would observe if they were made persistent using various methods. When we run the queue experiments, we keep in mind that these queues should be used within general programs. So, before calling each of the queue operations, the general program has to execute a capsule boundary. This is true for *all* queues that we test, including the LogQueue and Romulus. Therefore, since this additional overhead would be the same for all queues tested, we omit it in our comparative experiments. However, we note that the LogQueue and Romulus produce stand-alone data structures, that maintain more information than our queues do if the initial capsule boundary is removed. This means that in some specific contexts, for example when a few queue operations are executed consecutively, a capsule boundary can be avoided before calling LogQueue or Romulus operations, whereas our constructions still require it. It is possible to store some extra information in our queue constructions to match the properties of the other queues, but this requires some careful manipulations, which are outside of the scope of this chapter.

Using the Izraelevitz Construction One general way to automatically achieve correctness in the shared model is to use the construction presented by Izraelevitz *et al.* [59] (described in Section 7.7). Figure 7.7 a) shows the result of applying our transformations along with Izraelevitz's construction to the Michael-Scott queue (MSQ) [75].

To isolate the overhead of our transformations, we also show the performance of a Michael Scott queue with just the Izraelevitz construction. We call this the *IZ* queue and it is an upper bound on how well our transformations can perform. The result of the Low-Computation-Delay Simulator is called *General*. *Normalized* represents the normalized data structure transformation introduced in Section 7.6.2.

As the *General* queue contains more *capsule boundaries* than the *Normalized* queue, we can see that *Normalized* performs 1.25x better when there are 2 running threads, and 1.17x better when there are 8 threads. Without any of the detectability transformations, the *IZ* queue performs better than *Normalized* by 1.13x and 1.26x for 2 and 8 threads respectively.

Competitors Another way to make our transformed queues correct in the shared model is to add flushes manually. The flushes we add are very similar to those in Friedman *et al.*'s Durable Queue [41]. The difference is that we flush both the head and tail to allow for faster recovery and we omit the return value array; Friedman *et al.* used the return value array to recover return values following a crash, but this functionality is handled by our transformations.

We compared the manual flush version of our transformed queues with the *Log* [41] as well as a queue written using *Romulus* [30]. We chose the *RomulusLR* version as it performed better for every thread count. The results are depicted in Figure 7.7 b). For scalable memory allocation, our transformations use jemalloc and *Romulus* uses its own scalable memory allocator. We ran *Log* queue both with and without jemalloc and found that it tends to be faster without jemalloc. The faster set of numbers are reported.

Our *Normalized* queue performs much better than Romulus when the thread count is low, which could partly be because Romulus incurs the extra overhead of implementing a persistent memory allocator and general transactional memory. *Romulus* scales very well and outpaces *General* because it uses flat combining [50], a technique where update transactions are aggregated and processed with a single lock acquisition and release.

When compared to *Log* queue, we found that *Normalized* performs better by 29% on one running thread and by up to 9% on 7-8 threads. The *Log* queue is better by up to 10% on 2-6 threads. We believe this is because *Normalized* performs less overall fences compared to *Log* queue, however, in some places, *Normalized* performs more work in between a read and its corresponding CAS. These instructions bottleneck performance at higher thread counts. With clever cache line usage, it is also possible to reduce *Log* queue enqueues by one flush, but we did not implement this in our experiments. Given that the *Log* queue was tailored to one particular data structure, we were impressed that our *Normalized* automatic construction gets comparable performance. Although we did not measure this experimentally, our transformation also has lower recovery time compared to *Log* queue in situations where the size of the queue is larger than the number of processes. This is because the recovery function of *Log* queue traverses the

entire queue starting from the head. In contrast, our recovery function just involves loading the previous capsule and performing the recovery function of a recoverable CAS object. Since we are using Attiya et al's implementation of recoverable CAS [7], recovery time is linear in the number of threads.

Figure 7.7 c) shows how these persistent queues compare to the original MSQ. From the graph, it looks like the cost paid by our transformations to ensure generality and quick recovery is not so much compared to the inevitable cost of persistence.

Chapter 8

Conclusion

Non-volatile memories, which have recently became available, are expected to change the way that future systems are built. On the one hand, they offer a new way of accessing persistent data, i.e., using byte-addressable load and stores, and provide durability at a minimal cost. On the other hand, careful thought needs to be invested when designing durable data structures since caches and registers are still volatile. By providing libraries that solve the challenging problems under the hood, we can guarantee programmers a safe-way to use NVMM, avoiding inconsistencies in the wake of a crash. Throughout this dissertation, we presented a collection of data structures and libraries that provides programming support for persistent memory. This collection guarantees different provable correctness criteria and is easy to use. It eliminates the need for having an expertise at hand when designing concurrent and durable programs. One of the main building blocks of our construction is relying on *lock-freedom*, which guarantees that if writes are written to the NVMM in the order they are executed, the data structure will be in a consistent state after the crash.

In Chapter 3 we presented three lock-free queues that satisfy different correctness guarantees; durable linearizability, buffered durable linearizability (with a `sync` operation), and detectable execution (which was introduced in Chapter 2). These queue designs are optimized for NVMM and they demonstrate options for how to deal with volatile caches. This was the first practical work to present a lock-free data structure in the context of NVMM. We proposed several guidelines for designing durable lock-free data structures, which we try to capture more formally in Chapter 4.

NVTraverse, which is presented in Chapter 4, is a general construction for a broad class of linearizable concurrent data structures. One main challenge in our work on NVTraverse was capturing the 'dependencies' among different operations, and understanding which of the dependent writes needs to be persisted. One observation that was made is that many data structures spend much of their time traversing a data structure before doing a relatively small update. Therefore, the goal is to avoid any flushes and fences during the traversal (read-only portion). Experiments show a significant performance improvement using our approach, even beating hand-tuned algorithms on sev-

eral workloads. Our construction, however, necessitates satisfying different properties, which may require some effort and proving that it suffices all the required conditions requires some expertise.

Therefore, we presented another general transformation in Chapter 5. This transformation applies to any linearizable lock-free data structure and is very easy to use, but requires some extra space. The implementation uses two copies of the data structure, one in a volatile replica, and the second in a persistent replica. By this, it exploits the hybrid system where non-volatile main memory co-exists with conventional DRAM. As a result, the generated data structures are extremely efficient, sometimes beating by a factor of up to 10× state-of-the-art transformations. It involves type annotation to replace the usage of `std::atomic` with `patomic`, replacing the calls to the system allocator with the Mirror allocator, and providing a traversal function for the nodes in the data structure.

Figuring out that the algorithms can be optimized even more, we introduced the FliT library in Chapter 6. FliT avoids unnecessary flushing by using *flit-counters* to track dirty cache lines. This library is widely applicable, making it very easy to improve many algorithms, and achieves tremendous performance enhancements. In addition, we propose the P-V interface, which defines the semantics of code in which some memory instructions can remain volatile, while others must be persisted. This interface is language- and architecture-agnostic, and we show it captures persistence behavior in many algorithms.

Finally, we introduced a way to guarantee detectability for general programs. This general construction takes any concurrent *program* with reads, writes and CASs, and makes it persistent. The idea is breaking up the program into contiguous chunks of code and saving information about the location and the state of the program in-between them. This means that some instructions may be repeated several times, which can be hazardous. We guarantee saving the minimal needed data and using some primitives that allow safely repeating these instructions.

All together, the work described in this dissertation illustrates that rethinking the key data structures and algorithms for NVMM not only improves performance and operational cost, but also simplifies a programmer's work.

Bibliography

[1] Zahra Aghazadeh, Wojciech Golab, and Philipp Woelfel. Making objects writable. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 385–395, 2014.

[2] Marcos K Aguilera and Svend Frølund. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241*, 2003.

[3] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, pages 99–125, 2000.

[4] AMD. Amd64 architecture programmer's manual. URL `https://www.amd.com/system/files/TechDocs/24594.pdf`.

[5] ARM. Arm architecture reference manual armv8, 2018. URL `https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf`.

[6] ARM. Arm architecture reference manual armv8, 2018. URL `https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf`.

[7] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 7–16. ACM, 2018.

[8] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Tracking in order to recover-detectable recovery of lock-free data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 503–505, 2020.

[9] Hillel Avni and Trevor Brown. Persistent hybrid transactional memory for databases. *Proceedings of the VLDB Endowment (PVLDB)*, page 409–420, 2016.

[10] H. Alan Beadle, Wentao Cai, Haosen Wen, and Michael L. Scott. Nonblocking persistent software transactional memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, page 429–430, 2020.

[11] Naama Ben-David, Guy Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory,. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.

[12] Naama Ben-David, Guy Blelloch, Yihan Sun, and Yuanhao Wei. Multiversion concurrency with bounded delay precise garbage collection. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.

[13] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *Conf. on Principles of Distributed Systems (OPODIS)*, volume 46, 2016.

[14] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 677–694, 2016.

[15] Guy Blelloch, Phillip Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[16] Trevor Brown. A template for implementing fast lock-free trees using htm. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 293–302. ACM, 2017.

[17] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, volume 49, pages 329–342. ACM, 2014.

[18] C++. Std::atomic library. URL `https://en.cppreference.com/w/cpp/atomic`.

[19] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. Understanding and optimizing persistent memory allocation. In *ACM Symposium on Memory Management (ISMM)*, ISMM 2020, page 60–73, 2020.

[20] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 49, pages 433–452. ACM, 2014.

[21] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. Nvmove: Helping programmers move to byte-based persistence. In *INFLOW*.

[22] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment (PVLDB)*, 8(7):786–797, 2015.

[23] Joel Coburn, Adrian Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *asplos*, 2011.

[24] Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 254–263, 2015.

[25] Nachshon Cohen, Michal Friedman, and James R Larus. Efficient logging in non-volatile memory by exploiting coherency protocols. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 1, page 67. ACM, 2017.

[26] Nachshon Cohen, Rachid Guerraoui, and Mihail Igor Zablotchi. The inherent cost of remembering consistently. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2018.

[27] Alexei Colin and Brandon Lucia. Termination checking and task decomposition for task-based intermittent programs. In *International Conference on Compiler Construction*, 2018.

[28] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 133–146, 2009.

[29] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. 2010.

[30] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–282. ACM, 2018.

[31] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[32] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. 2018.

[33] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent processor architecture. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011.

[34] J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and H. Wehrheim. Verifying correctness of persistent concurrent data structures: a sound and complete method. *Formal Aspects of Computing*, 2021.

[35] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *J. ACM*, 1997.

[36] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing (PODC)*, volume 10, pages 131–140. ACM, 2010.

[37] Lokesh Gidra Abraham Alcantara Sergio Gonzalez Sergei Uversky Evan Kirshenbaum, Susan Spence. Hewlett Packard Labs: Data structures managed like never before on The Machine. *https://www.youtube.com/watch?v=3fN5_Qt9OCs*, 2016.

[38] Facebook. Rocksdb. URL `https://github.com/facebook/rocksdb`.

[39] Keir Fraser. Practical lock-freedom. 2003.

[40] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy Blelloch, and Erez Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[41] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, volume 53, pages 28–40. ACM, 2018.

[42] Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: Making lock-free data structures persistent. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1218—1232, 2021.

[43] Ellis Giles, Kshitij Doshi, and Peter J. Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2015.

[44] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M Chen, and Thomas F Wenisch. Persistency for synchronization-free regions. *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2018.

[45] Google. Leveldb. URL `https://github.com/google/leveldb`.

[46] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *USENIX Annual Technical Conference (ATC)*, USA, 2019.

[47] Rachid Guerraoui and Ron R Levy. Robust emulations of shared memory in a crash-recovery model. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 400–407. IEEE, 2004.

[48] Rachid Guerraoui, Alex Kogan, Virendra J Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. *International Symposium on Distributed Computing (DISC)*, 2020.

[49] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing (DISC)*, pages 300–314. Springer, 2001.

[50] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, 2010.

[51] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 197–206, 1990.

[52] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *International Colloquium on Structural Information and Communication Complexity*, pages 124–138. Springer, 2007.

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

[54] Intel. Developers intel64 and ia-32 architectures software manuals combined, . URL `https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html`.

[55] Intel. Persistent memory library, . URL `https://pmem.io`.

[56] Intel. pmemkv-bench., . URL `https://github.com/pmem/pmemkv-bench`.

[57] Intel. Intel architecture instruction set extensions programming reference, . URL `https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html`.

[58] Ramin Izadpanah, Christina Peterson, Yan Solihin, and Damian Dechev. Petra: Persistent transactional non-blocking linked data structures. *ACM Transactions on Architecture and Code Optimization*, 2021.

[59] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing (DISC)*, pages 313–327. Springer, 2016.

[60] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Dhtm: Durable hardware transactional memory. In *ACM International Symposium on Computer Architecture (ISCA)*, pages 452–465, 2018.

[61] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 399–411, 2016.

[62] Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free fifo queues. *Distributed Computing*, pages 323–341, 2008.

[63] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. Wort: Write optimal radix tree for persistent memory storage systems. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 257–270, 2017.

[64] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: converting concurrent dram indexes to persistent-memory indexes. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 462–477. ACM, 2019.

[65] Herwig Lejsek, Friðrik Heiðar Ásmundsson, Björn Þór Jónsson, and Laurent Amsaleg. Nv-tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(5):869–883, 2009.

[66] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–343. ACM, 2017.

[67] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.

[68] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence: Efficient transactions in persistent memory. *ACM Transactions on Storage*, pages 3:1–3:29, 2016.

[69] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. *PLDI*, 2015.

[70] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, et al. Persistent memory transactions. *arXiv preprint arXiv:1804.00701*, 2018.

[71] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *ACM European Conference on Computer Systems (EuroSys)*, pages 499–512, 2017.

[72] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and fast persistence for volatile data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 789–806, 2020.

[73] Maged M Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 21–30. ACM, 2002.

[74] Maged M. Michael. Hazard pointers, 2017. URL `https://github.com/facebook/folly/tree/master/folly/experimental/hazptr`.

[75] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275. ACM, 1996.

[76] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 103–112, 2013.

[77] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM, 2014.

[78] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency: Semantics for byte-addressable nonvolatile memory technologies. *International Symposium on Microarchitecture (MICRO)*, pages 125–131, 2015.

[79] Azalea Raad and Viktor Vafeiadis. Persistence semantics for weak memory: Integrating epoch persistency with the tso memory model. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.

[80] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the intel-x86 architecture. *ACM Symposium on Principles of Programming Languages (POPL)*, 2019.

[81] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the intel-x86 architecture. *Proc. ACM Program. Lang.*, (POPL), 2019.

[82] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of armv8 and transactional models. *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2019.

[83] Azalea Raad, John Wickerson, and Viktor Vafeiadis. Weak persistency semantics from the ground up: Formalising the persistency semantics of armv8 and transactional models. *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, 3(135), 2019.

[84] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. 2019.

[85] Steve Scargall. *pmemkv: A Persistent In-Memory Key-Value Store*, pages 141–153. 2020.

[86] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, volume 49, pages 357–368. ACM, 2014.

[87] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *USENIX Conference on File and Storage Technologies (FAST)*, volume 11, pages 61–75, 2011.

[88] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 39, pages 91–104. ACM, 2011.

[89] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment (PVLDB)*, 7(10):865–876, 2014.

[90] Tianzheng Wang, Levandoski Justin, and Larson Per-Ake. Easy lock-free indexing in non-volatile memory. In *IEEE International Conference on Data Engineering (ICDE)*, pages 461–472. IEEE, 2018.

[91] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy Blelloch, and Erez Petrank. Flit: A library for simple and efficient persistent algorithms, 2021.

[92] Jian Xu and Steven Swanson. {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 323–338, 2016.

[93] Chaoran Yang and John Mellor-Crummey. A wait-free queue as fast as fetch-and-add. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 16:1–16:13, 2016.

[94] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 167–181, 2015.

[95] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets - under review. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2019.

להתאושש מנפילה במספר קבוע של פעולות). טרנספורמציה זו מתבססת על הוספת נקודות ציון השומרות את מצב התוכנית ומאפשרת עמידות בפני קריסת מערכת. לאחר נפילה, התוכנית תחזור לנקודת הציון האחרונה אשר נשמרה ולמצב המשתנים הלוקליים באותה נקודת זמן, ותתחיל לבצע את כל הפעולות מחדש החל מנקודה זו.

כדי להתמודד עם הקושי בתכנון מבני נתונים מקביליים ועמידים בפני נפילות, ישנם מספר אלגוריתמים אשר ניתן לסווג אותם ל-3 שיטות עיקריות: טרנזקציותת טרנספרומציות ידניות, וטרנספורמציות כלליות.

שיטת הטרנזקציות הינה שיטה כללית היכולה לספק מקביליות ועמידות למבנה נתונים באופן פשוט. אך, כלליות ופשטות השיטה פוגעות בביצועי התוכנית. פגיעה זו נובעת מהצורך בשמירה נוספת של נתונים ב- NVMM בעת ביצוע הטרנזקציות. יתר על כן, הן מסדרות כתיבות באופן סדרתי, דבר המגביל באופן משמעותי סקלביליות בתוכניות עם עומסי כתיבה משמעותיים.

שיטת הטרנספורמציות הידניות הינה שיטה המתבססת על ניצול התכונות הייחודיות של מבני נתונים ספציפיים ומאפשרת ריצה יעילה. בעבודה זו, מוצגים מימושים חדשניים עבור תור מקבילי המקיים את תכונת ה- lock-freedom. מימושים אלה ממחישים את האתגרים האלגוריתמיים הקיימים בבניית תור lock-free העמיד בפני נפילות, תחת הבטחות נכונות שונות. שיטות אלה אינן כלליות, ומסתמכות על הסמנטיקה והמבנה של התור. תורים אלה הינם המבנים הראשונים מסוגם המותאמים לזיכרון ראשי שאינו נדיף. למרות ייחודיות הבנייה עבור התור, אנו מציגים מספר קווים מנחים כלליים הנדרשים להמרת מבני נתונים לינאריזביליים לעמידים בפני נפילות. בנוסף לבנייה ולקווים המנחים, אנו מציגים תכונה חדשה הנקראת detectable-execution. מבנה נתונים מקיים תכונה זו כאשר בסוף פעולת ההתאוששות ניתן לומר האם פעולה מסויימת בוצעה או לא.

טרנספורמציות כלליות הינה שיטה שלישית לתכנון מבני נתונים לזיכרון אשר אינו נדיף. תחת שיטה זו, אנו מציגים שתי טכניקות כלליות, NVTraverse ו- Mirror המשמשים לבניית מבני נתונים עמידים ו- lock-free. שתי טכניקות אלה מוצגות כספריות עבור NVMM, וממזערות את מאמץ התכנות וההבנה הנדרשים לצורך הוספת עמידות למבני נתונים שאינם עמידים לנפילות מלכתחילה. הניסויים שלנו מראים כי בניות אלה נמצאו יעילות יותר בהשוואה לגישות האחרות הקיימות. הטרנספורמציה של NVTraverse הינה טרמספורמציה כללית הלוקחת מבנה נתונים lock-free מקבוצה כללית הנקראת traversal data structure והופכת אותו באופן אוטומטי למבנה נתונים יעיל ועמיד המתאים ל- NVMM. הטרנספורמציה מסתמכת על האבחנה כי פעולות רבות של מבני נתונים מתחילות בשלב סריקה המבוסס על קריאה בלבד ללא צורך בשמירתם. הצורך בשמירת הנתונים בא לידי ביטוי רק באיזור אליו מגיעים בסיום הסריקה, אשר בו מתבצעים השינויים בפועל. Mirror אשר שונה מ-NVTraverse , הינה טרנספורמציה פשוטה ואוטומטית, ומוסיפה עמידות לכל מבנה נתונים שהוא lock-free. המבנה החדש שנוצר ע"י Mirror הוא בעל עלות ביצועית נמוכה המנצל את הזיכרון ההיברידי הבנוי מ-NVMM ומ-DRAM. בהמשך לפיתוח הטרנספורמציות הכלליות, בעבודה זו מוצגת ספרייה הנקראת Flit. זוהי ספריית C++ שמאפשרת כתיבת קוד יעיל ועמיד. השימוש בברירת המחדל של ספרייה זו הופך כל מבנה נתונים ליניאריזבילי, לעמיד בפני נפילות, ומצריך שינוי מינימלי של הקוד בכדי לאפשר זאת. ספריית ה- Flit נמנעת מביצוע פקודות flush (המעבירות נתונים מזיכרון המטמון לזיכרון הראשי) מיותרות על ידי שימוש באלגוריתם חדשני העוקב אחר שורות אשר השתנו בזיכרון המטמון. הספרייה מסתמכת על מונים העוקבים אחר הכתיבות המתבצעות לכל משתנה. בדרך זו, פקודות ה-flush מתבצעות רק כשיש בהן צורך, וחוסכות קריאה לפקודות מיותרות. בחלק האחרון בעבודה, מוצג פיתוח של בנייה כללית המוסיפה עמידות לכל תוכנית מקבילית. פיתוח זה מאפשר את שמירת המצב הכללי של התוכנית, ושיחזורו בזמן התאוששות מנפילה. בנייה זו מיועדת לכל תוכנית, ולא בהכרח רק למבני נתונים. בתוכנית המתקבלת לאחר הבנייה ישנה השהייה חישובית קבועה (כמות הפעולות מוכפלת בקבוע לכל תהליך), כמו גם השהיית התאוששות קבועה (תהליך יכול

תקציר

השימוש בזיכרון משני למטרת איחסון מידע מהווה צוואר-בקבוק משמעותי בעת ביצוע תוכניות ברוב מערכות המחשוב הקיימות. תפקיד הזיכרון הופך להיות משמעותי אף יותר עם התפתחות טכנולוגיות המחשוב המודרניות המתבססות על שמירת נתוני התוכנית, גם כאשר התוכנית אינה רצה. הפיתוח החדשני של הזיכרון הראשי הלא-נדיף (NVMM), מייצר מהפכה באופן בו אנו משתמשים בזיכרון המשני. זיכרון זה מספק עמידות גבוהה בפני נפילות, מאפשר כתיבה בבתים, בעל קיבולת גדולה יותר מהזיכרון הראשי (DRAM) ובעל ביצועים טובים יותר ביחס לזיכרון המבוסס על טכנולוגיית ה- SSD.

במחקרים רבים הנעשו לאחרונה בתחום, מציעים דרכים שונות למימוש יתרונות ה-NVMM. בחלק ממחקרים אלו, ישנו ניסיון לתכנן מבני נתונים עמידים בפני נפילות. מבני נתונים אלה מאפשרים אחסון מידע באופן יעיל ומהווים חלק בלתי נפרד מכל תוכנית. כיום, מבני נתונים בזיכרון הראשי נשמרים בפורמט בתים, ואילו בזיכרון המשני, הם נשמרים בפורמט מבוסס בלוקים. כתוצאה מכך, הייצוג של מבני הנתונים בזיכרון הראשי והמשני שונה. השוני בין הפורמטים טומן בחובו עלות ביצועית גבוהה. כדי להימנע מעלות זו, ניתן להשתמש בזיכרון הראשי הלא-נדיף (NVMM) במקום בזיכרון המשני הסטנדרטי. ישנו אתגר רב בתכנון מבני נתונים ואלגוריתמים הנשמרים בזיכרון הראשי שאינו נדיף, היות וזיכרון המטמון והרגיסטרים עדיין נדיפים. אתגר זה נובע מהעובדה כי תכני זיכרון המטמון והרגיסטרים, המשקפים את הנתונים המעודכנים ביותר, הולכים לאיבוד בעת קריסה. מצב זה עלול ליצור אי עקביות במצב ה-NVMM לאחר התאוששות מנפילה, עקב אי שמירת הנתונים האחרונים שנותרו בזיכרון המטמון וטרם הגיעו לזיכרון הראשי. כדי להתמודד עם תופעה זו, ניתן להשתמש בהוראות מיוחדות המעבירות נתונים מזיכרון המטמון לזיכרון הראשי. הוראות אלו מבטיחות את כתיבתם בזיכרון וקיומם לאחר נפילה. יחד עם זאת, העברת נתונים מהמטמון לזיכרון הראשי באופן מפורש כרוכה בעלות ביצועית גבוהה, וישנה עדיפות להשתמש בהוראות אלה בתדירות נמוכה. שמירת הנתונים המעודכנים ביותר בזיכרון הראשי בעלות ביצועית נמוכה הופך להיות מורכב אף יותר בעת תכנון מבני נתונים עמידים ומקביליים. עבודת הדוקטורט הזו נועדה לתת מענה לבעיית תכנון מבני נתונים ואלגוריתמים מקביליים, יעילים, ועמידים עבור ה- NVMM.

אחד הכלים שבאו לידי ביטוי במהלך עבודת הדודטורט הינו שימוש בתכונת ה-lock-freedom. lock-freedom זוהי תכונה המבטיחה כי הזיכרון תמיד נמצא במצב עקבי, גם במהלך עדכונים ארוכים. תכונה זו דורשת מהתהליכים להיות מסוגלים לבצע פעולות בזיכרון המשותף ללא קשר להתקדמות איטית או אפילו לכישלון של תהליכים אחרים במערכת. המשמעות היא, שגם אם תהליך מסויים לא רץ כרגע, תהליכים אחרים יכולים להמשיך ולבצע את הפעולות שלהם. לכן, השימוש באלגוריתמים שהם lock-free מתאים באופן טבעי בעבודה עם ה- NVMM.

המחקר בוצע בהנחייתו של פרופסור ארז פטרנק, בפקולטה למדעי המחשב.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי-עת במהלך תקופת מחקר הדוקטורט של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, volume 53, pages 28–40. ACM, 2018.

Naama Ben-David, Guy Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory,. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2019.

Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy Blelloch, and Erez Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: Making lock-free data structures persistent. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1218—1232, 2021.

Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy Blelloch, and Erez Petrank. Flit: A library for simple and efficient persistent algorithms, 2021.

תודות

ברצוני להביע את הערכתי ותודתי העמוקה למנחה יוצא הדופן, ארז פטרנק בעבור הנחייתו ותמיכתו. הייתה לי זכות ענקית לעבוד עם חוקר ומנחה שלימד אותי סט כלים מקצועיים ובין אישיים במהלך לימודי הדוקטורט. ברצוני להודות לארז על אמונתו התמידית ביכולותיי ובכך שהיה הרבה יותר ממנחה בעבורי.

היה לי התענוג לעבוד עם קולגות שמהם למדתי הרבה, נהניתי מהדרך ונתנו לי המון השראה: גיא בללוך, נעמה בן-דוד, מוריס הרליהי, יואנהאו ווי, נחשון כהן, ג'ים לרוס, וירנדרה מרתה, ארז פטרנק, יואב צוריאל, פדרו רמלתה וגלי שפי.

אני מודה לוועדת הבוחנים בזמן המעבר לישיר לדוקטורט, יהודה אפק, יורם מוזס, רועי פרידמן ואסף שוסטר על הזמן שלהם, התובנות והעצות שהם נתנו לי. אני מודה גם לוועדת הבוחנים הסופית, יהודה אפק ורועי פרידמן על ההערות הבונות.

ברצוני גם להודות לדויד בקון שאירח אותי בהתמחות קיץ בגוגל ניו יורק בקיץ 2017. התמחות זו אמנם לא הייתה מקושרת באופן ישיר לעבודת הדוקטורט, אך תרמה רבות לסט הכלים שברשותי.

הפקולטה למדעי המחשב הייתה הבית השני שלי ואני מודה לכל הגורמים האדמיניסטרטיביים והעובדים שתרמו להרגשה זו. אני מודה לחברים המדהימים שפגשתי לאורך הדרך. אני ברת מזל שיש לי אותכם.

אני מודה למשפחתי ובייחוד לאימי על העידוד למצויינות והאמונה האינסופית בי. ותודה אחרונה ומיוחדת, לאחד והיחיד שלי, השותף שלי לחיים, בעלי עמית. לא הייתי יכולה לעשות את זה בלעדיך. אתה מעניק לי מוטיבציה, ייעוץ, אהבה אינסופית ותמיכה שתמיד שמים הכל בפרופורציות המתאימות. אני מקדישה לך את עבודה זו.

הכרת תודה מסורה לטכניון וקרן עזריאלי על מימון מחקר זה.

מבני נתונים מקביליים לזיכרון שאינו נדיף

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

מיכל-יבגניה קורנברג (פרידמן)

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
תשרי התשפ"ב חיפה ספטמבר 2021

מבני נתונים מקביליים
לזיכרון שאינו נדיף

מיכל-יבגניה קורנברג (פרידמן)