Recoverable Mutual Exclusion in Detectable Lock-Based Data Structures

by

Ahmed Fahmy

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2024

© Ahmed Fahmy 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Tianzheng Wang Associate Professor, Dept. of Computer Science, Simon Fraser University
Supervisor(s):	Wojciech Golab Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Member:	Mahesh Tripunitara Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal Member:	Seyed Zahedi Assistant Professor, Dept. of Electrical and Computer Engineering, University of Waterloo
Internal-External Member:	Trevor Brown Associate Professor, Dept. of Computer Science, University of Waterloo

Author's Declaration

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

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

Abstract

Persistent memory (PM) is an emerging technology that offers the speed of DRAM combined with the persistence of traditional storage. This advancement provides unique opportunities and challenges for designing data structures that remain consistent and *recoverable* after system failures.

This thesis presents significant advancements in the design and implementation of recoverable synchronization algorithms and data structures optimized for PM. The research focuses on addressing the challenges of *recoverable mutual exclusion* (RME) and the development of *detectable* lock-based data structures, offering innovative solutions that enhance performance and reliability in concurrent systems.

A major contribution of this work is the introduction of the *Recoverable Filter* (RF) lock, a novel approach that enhances RME lock performance in the *Non-Uniform Memory Access* (NUMA) multi-processor architecture, where memory access time depends on the memory location relative to the processor. Such a technique transforms NUMA-*oblivious* RME locks into NUMA-*aware* ones without requiring modifications to the underlying locks. This solution tackles the dual challenges of recoverability and NUMA-awareness, a combination not previously addressed in the literature on RME locks. Comprehensive empirical evaluations using prominent RME algorithms, specifically GH and JJJ, demonstrate that the RF lock significantly boosts performance by up to 45% in multi-socket configurations, while maintaining minimal overhead in single-socket setups. These findings underscore the effectiveness of the RF lock in leveraging memory locality to improve efficiency.

Additionally, this thesis introduces DULL, a Detectable Unrolled Lock-based Linked List, designed for persistent memory environments. DULL distinguishes itself as the fastest *detectable* lock-based linked list and the first to achieve *strict-linearizability*. The implementation of DULL utilizes volatile RME locks to address the intricate challenge of preserving essential lock properties, such as mutual exclusion and deadlock-freedom, across system failures. Performance evaluations reveal that DULL outperforms existing solutions in update-intensive scenarios and maintains scalability even under high processor subscription levels.

By exploring the challenges of designing concurrent recoverable algorithms with incorporated locks, this thesis makes substantial contributions to the field of concurrent data structures and synchronization algorithms. The innovative solutions presented herein lay a robust foundation for future research and practical applications, enhancing the reliability and performance of concurrent systems in persistent memory environments.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Wojciech Golab, for his invaluable guidance, mentorship, and unwavering support throughout my PhD journey. I also extend my sincere thanks to the members of the examining committee—Professor Tianzheng Wang, Professor Mahesh Tripunitara, Professor Seyed Zahedi, and Professor Trevor Brown—for their insightful feedback and efforts, which greatly contributed to the refinement of this work.

Dedication

This is dedicated to my parents, Hany and Samia, my parents-in-law, Osama and Khadija, my beloved wife, Basma, my sister, Lamia, and my brothers-in-law, Amr and Abdulrahman. Thank you for your unconditional support. There is no way I would have done it without you. Alhamdu-lil Allah!

Table of Contents

List of Figures x

1 Introduction 1

2 Preliminaries 4

2.1 System Architecture . . . 4
2.2 Persistent Memory . . . 5
2.3 Failure Model . . . 6
2.4 Mathematical Notations . . . 6
2.5 Recoverable Mutual Exclusion . . . 7
2.5.1 Histories . . . 8
2.5.2 Complexity Measures . . . 9
2.5.3 Correctness Properties . . . 9
2.6 Detectable and Persistent Data Structures . . . 10
2.6.1 Modeling Shared Objects . . . 12
2.6.2 Data Types and Sequential Specifications . . . 12
2.6.3 Histories . . . 14
2.6.4 Correctness Properties . . . 16
2.6.5 Progress Properties . . . 17
2.6.6 Detectable Unordered Set . . . 17
2.7 Background Algorithms . . . 19
2.7.1 Unrolled Linked List . . . 19

3 Related Work **24**

4 A NUMA-Aware Recoverable Mutex Lock **29**

4.1 Recoverable Filter Lock . . . 30
4.2 Transformation of RF lock . . . 31
4.3 Implementation Details . . . 33
4.3.1 Filtering Operation . . . 35
4.3.2 Filter Handoff Operation . . . 38
4.3.3 Filter Recovery Operation . . . 39
4.4 Correctness . . . 40
4.5 Experimental Evaluation . . . 74
4.5.1 Scalability . . . 75
4.5.2 Single Socket Overhead . . . 78
4.5.3 Closer Look at Single Socket Overhead . . . 80

5 DULL: A Fast Scalable Detectable Unrolled Lock-Based Linked List **82**

5.1 Persistent Objects . . . 82
5.1.1 Persistent Pointers . . . 83
5.1.2 Persistent Pair . . . 84
5.1.3 Descriptor . . . 85
5.2 Linked List Structure . . . 87
5.3 Implementation Details . . . 88
5.3.1 Memory Layout . . . 89
5.3.2 Traversal and Search . . . 89
5.3.3 Insertion . . . 91
5.3.4 Removal . . . 94
5.3.5 Resolve . . . 99
5.3.6 Recovery . . . 99

5.4 Correctness . . . 102
5.4.1 Safety . . . 102
5.4.2 Liveness . . . 132
5.4.3 Correctness Theorem . . . 134
5.5 Experimental Evaluation . . . 134
5.5.1 Recoverability Testing . . . 134
5.5.2 Performance . . . 136
5.5.3 Results . . . 138

6 Conclusion **145**

References **148**

List of Figures

2.1 CC-NUMA system with 4 fully connected nodes and 3 levels of cache . . . 5
2.2 Axioms for unordered set operations . . . 14
2.3 Simplified (SUDSS) of base type T, denoted $SUD\langle T\rangle$. . . . 18
2.4 Axioms for detectable unordered set $SUD\langle Set\rangle$ operations . . . 20

4.1 Throughput in multi-socket configuration using round-robin mapping. . . . 76
4.2 Throughput in multi-socket configuration using node-by-node mapping. . . . 78
4.3 Throughput when using a single socket. . . . 79
4.4 Average execution time per (super-)passage per thread of the entry and exit sections using a single socket. . . . 80

5.1 Memory layout for DULL. . . . 89
5.2 The representation invariant $I(r)$ of *DULL*. . . . 106
5.3 The definition of the **getSig**, **getTag** and **getRes** functions in $A(r)$ . . . 108
5.4 The abstraction function $A(r)$ that maps a representation value (r) to a set of abstract values of a *DSet*'s object (v). . . . 108
5.5 Balanced workload – key space 1000 . . . 139
5.6 Balanced workload – key space 5000 . . . 140
5.7 All-write workload – key space 1000 . . . 141
5.8 All-read workload – key space 1000 . . . 142
5.9 Throughput relative to the Vanilla-ULL - 20 threads - balanced workload. . . 144

Chapter 1

Introduction

Non-volatile random access memory, or *persistent memory* (PM), has been a research focus for the past few years. Such technology enables computer systems to maintain data in memory across transient failures, such as power outages. The evolution of PM blurs the line in current computer architectures between the primary *dynamic random-access memory* (DRAM) and the secondary permanent storage (e.g., solid-state drives). In lieu of utilizing the DRAM for high-speed memory access and the secondary storage for data recovery, the PM promises the best of the two worlds, performance and durability. PM is byte-addressable like the DRAM and non-volatile like the secondary storage, allowing computer programs to directly recover in-memory data structures without sequentially transferring data from the secondary storage. The durability of PM stimulated the need for designing data structures that are aware of the underlying memory hardware to employ its recovery feature.

With the emergence of PM, data structure algorithms are potentially able to preserve their consistent state after recovering from a crash failure. Nowadays, most multiprocessor computers use the *cache-coherent* (CC) memory model, where each processor has its local cache while all processors share memory modules that can be accessed through some interconnect. In this model, a process stores a local copy of the variable in its local cache for faster subsequent access to the data. Designing recoverable, or persistent, data structures for CC systems is challenging. In such systems, caches constitute a volatile level in the memory hierarchy that falls between the processing units and the PM. If a data structure changed its state based on data in a volatile cache that is not yet written back to memory, a subsequent system-wide failure erases those data, and recovering from that crash might lead to inconsistency. Therefore, persistent data structures should be designed carefully to establish consistency. For example, instructions for writing back cache lines and mem-

ory fences can impose order in which data are written into memory. Recently, Intel has introduced a new technology on the 3rd Generation Xeon processors, called the extended asynchronous DRAM Refresh (eADR), that includes CPU caches into the system-wide failure protected domain [68]. Such a technology removes the need for cache line write-back instructions. However, memory fences and recovery protocols are still required to ensure consistency.

Many recoverable concurrent data structures have been presented [70, 19, 7, 46, 21, 34, 69, 4]. Some data structures are said to be detectable [6, 8, 9, 34], which enables detecting the response of a crashed operation. Published lock-based detectable and persistent data structures [15, 43, 61, 66] are not as common or practical as lock-free structures. For example, some algorithms [15, 43] use logs which degrades the performance. Others [61, 66] adopt privileged persistence instructions (e.g., flush-the-whole-cache) and specific techniques that limit scalability. Designing detectable lock-based data structures is challenging. The correctness properties of the underlying locks, such as mutual exclusion and deadlock-freedom, must be preserved when adding persistence or detectability. Such a requirement is beyond the definition of the *mutual exclusion* (ME) problem [28], as the classic formalization assumes failure-free systems. Recently, the definition of the classical ME problem was generalized by modeling the novel *recoverable mutual exclusion* (RME) problem that considers crash failures [38].

My research concentrates on investigating the challenges and potential advantages of using locks in the context of recoverability and detectability. I aim to achieve this through designing a detectable lock-based data structure. I believe the underlying locks can be exploited to facilitate adding detectability and recoverability to a data structure. The idea of exploiting locks to add recoverability is not far from reality. In fact, it has been done before [15, 43]. This leads to the following interesting research questions:

Q1. What is the performance impact of using RME locks on modern multicore multiprocessor systems, and how can we improve it?

Q2. Is the current definition of the RME problem include volatile locks?

Q3. Can we employ data structures' underlying (RME) locks to achieve detectability, and how such an approach affects performance in practice?

In addressing Q1, I begin by identifying key gaps in the existing literature regarding mutex locks, which are essential building blocks of lock-based data structures. Specifically, I focus on overcoming the performance limitations of current RME algorithms in *non-uniform memory access* (NUMA) systems (Chapter 4), by introducing a black-box

transformation approach that leverages memory locality to convert a NUMA-oblivious recoverable mutex lock into a NUMA-aware one. To the best of my knowledge, this transformation is the first of its kind to be both black-box and recoverable. Applying my algorithm, I transform two leading RME algorithms [36, 51] into NUMA-aware RME locks. Experimental results indicate that my transformation enhances the performance of existing RME locks by up to 45% and even outperforms the well-known non-recoverable cohort transformation [27] by up to 9%.

Additionally, I address Q2 by demonstrating that *persistent* locks are not strictly required to achieve recoverability in lock-based data structures. Specifically, in Chapter 5, I incorporate recoverability into the fastest known linked list [62], which relies solely on volatile locks, illustrating that volatile locks can effectively support recoverable systems. Furthermore, I address Q3 by extending the recoverable linked list to include detectability, leveraging the underlying locks to realize this added feature. The result is a fast and scalable detectable lock-based and, to the best of my knowledge, the first detectable strictly-linearizable linked list. I validate the scalability of my list by comparing my list against various lock-free and lock-based, recoverable and non-recoverable, and detectable and non-detectable lists. Experimental results show that DULL significantly outperforms competitors across all update-intensive workloads when the keyspace size ≥ 512. Moreover, the experiments reveal that my algorithm scales effectively, even when the multiprocessor is oversubscribed, a limitation in some previous works.

In this thesis, I present each proposed algorithm with detailed pseudocode, along with a thorough analysis to establish correctness. Furthermore, I implement C++ prototypes of my algorithms and other state-of-the-art methods, conducting experiments to validate theoretical results with practical evidence.

The structure of the thesis is organized as follows: Chapter 2 introduces the system model and background concepts that underpin this work. Chapter 3 provides an overview of the current state of the art in related research. In Chapter 4, I propose a black-box approach for transforming a NUMA-*oblivious* RME algorithm into a NUMA-aware recoverable lock. Chapter 5 presents the design and implementation of a detectable and recoverable lock-based linked list. Finally, Chapter 6 offers the concluding remarks and summarizes the key contributions of the thesis.

Chapter 2

Preliminaries

In this chapter, I will introduce the system model, along with key terminologies and algorithms utilized in the design and correctness analysis of my algorithms.

2.1 System Architecture

Modern distributed systems are often designed based on multi-socket architecture, where each socket, or node, has a processor that comprises multiple cores. Hereafter, I shall use the terms socket and node interchangeably. My focus is on the cache-coherent model where each core has a local cache. Each node has shared memory modules accessed uniformly by processor cores located on the same socket. Such a system is known as CC-NUMA architecture. Figure 2.1 illustrates an example of a CC-NUMA system consisting of 4 nodes and 3 cache levels.

Data in volatile caches are not guaranteed to be written back to the memory unless explicit cache-line write-back instructions are used. In addition, the order of writing back cache lines to the memory is not ensured. The order can be enforced by using memory fences (more in Section 2.2). In CC-NUMA systems, processes running on a NUMA node can access memory locations on remote nodes through an interconnect. Therefore, processes on different nodes can communicate by accessing shared variables. A shared variable is said to be *local* to a NUMA node if it is accessed only by processes running on that node. The latency for a process to access variables that reside in remote NUMA nodes is higher than accessing variables in its local node. Due to cache coherence, the cost for a process to access a variable written or modified by a process on another node is greater than the

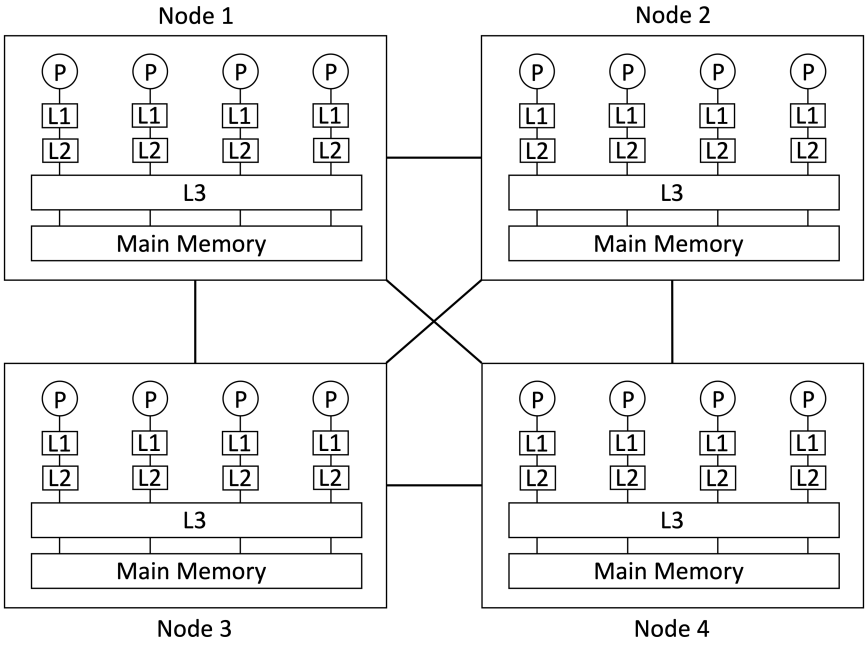


Figure 2.1: CC-NUMA system with 4 fully connected nodes and 3 levels of cache

cost when both processes are co-located on a single node. I refer to this additional cost as the *cross-socket communication overhead*.

2.2 Persistent Memory

In this thesis, I adopt a storage model where byte-addressable *persistent memory* (PM) is directly linked to the memory bus. I assume no support for eADR, meaning that while main memory retains data after a failure, CPU caches and registers are volatile, meaning their contents are lost in events like power outages. One of the challenges with volatile caches is that there is no control over when or in what order data is written back to PM, as cache line write-backs can happen unpredictably. To ensure data is explicitly written to PM, a cache line flush instruction is used. For instance, on Intel systems, the **CLFLUSH** instruction takes a memory address and flushes the entire cache line containing that address to memory, while also invalidating it. A newer instruction, **CLWB**, enables the cache line to be written back without invalidation.

It is crucial to guarantee that cache line write-backs occur in the correct order. To

enforce this, the **SFENCE** instruction, a type of memory barrier, ensures the completion of all preceding store instructions before any subsequent stores are executed. By using **SFENCE** along with flush instructions, I can achieve both proper data durability and ordering. In the algorithms presented in this thesis, I use a flush instruction, denoted as **Flush**(), that takes a memory address to represent both cache line flushes and memory barriers.

2.3 Failure Model

I adopt a crash-recovery failure model [37, 50, 22, 16, 38] that considers both individual process and system-wide failures. In the case of individual process failures, a process may crash at any point during its execution and recover by restarting from the beginning of its program, with private variables reset to their initial values. Shared variables remain intact in the cache and PM levels. For system-wide failures, all processes crash simultaneously, and all data in the cache is lost, with only the values stored in PM being retained. Processes use flush instructions to ensure cached data is written to PM, guaranteeing persistence in case of a crash.

2.4 Mathematical Notations

I use $(\circ)$ for sequence concatenation. Let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$ be two sequences. The *concatenation* of A and B, denoted by $A \circ B$, is the sequence $C = (c_1, c_2, \ldots, c_{n+m})$, where:

$$c_i = \begin{cases} a_i & \text{for } 1 \leq i \leq n \\ b_{i-n} & \text{for } n+1 \leq i \leq n+m \end{cases}$$

In other words, the first n elements of C are the elements of A, and the next m elements are the elements of B.

Furthermore, I adopt the standard subscript notation to refer to elements of a tuple. Specifically, for a tuple $t = (t_1, t_2, \ldots, t_n)$, I denote the i-th element by t_i.

2.5 Recoverable Mutual Exclusion

In distributed computing, simultaneous access to a shared resource that involves at least one write operation by multiple processes, or threads, leads to race conditions where program outcomes become inconsistent. The *mutual exclusion* (ME) problem was first introduced by Dijkstra in [28]. To mitigate race conditions, the ME problem prescribes imposing sequential execution of a code segment called the *critical section* (CS), in which shared resources are accessible. The mutex lock is a widely-used solution to the ME problem.

ME algorithms typically assume system stability, where processes are guaranteed to be free of failures, particularly while acquiring or releasing locks or executing within the critical section (CS). In practice, however, such failures can occur, leading to deadlock or corrupted mutex locks, which violate both safety and liveness properties. For instance, if a process crashes during the execution of the CS, another thread in the entry section may starve indefinitely, waiting for the lock to be released.

Recently, Golab and Ramaraju generalized the classical ME problem by introducing the novel *recoverable mutual exclusion* (RME) problem, which accounts for crash failures [38]. The definition of the RME problem is inspired by, but not exclusively tied to, the concept of persistent memory (PM). For example, an RME algorithm could handle individual process crashes without relying on PM hardware. Nonetheless, leveraging such technology elevates fault tolerance to a new level, enabling recoverable mutex locks to withstand system-wide failures, like power outages, in addition to individual process crashes.

An RME algorithm can be executed concurrently by N asynchronous processes $p_1, ..., p_N$ prone to crash failures. The execution of an RME algorithm involves five sequential sections [38], as shown in Algorithm 1. A process is in the *non-critical* section (NCS) when it is

Procedures of an RME lock.

```
1 while True do
2     NCS
3     Recover()
4     Enter()
5     CS
6     Exit()
```

Algorithm 2.1: Execution path of an RME algorithm

initialized for the first time or restarted after a crash failure. The *recovery* section is where

processes perform lock state recovery and the associated clean-up after a crash failure. The recovery section is accessed immediately after the NCS. Subsequently, a process executes the *entry* section. A process must complete the entry section before it proceeds to the CS. A process executes the *exit* section immediately after the CS. A process is in the CS if and only if it has completed the entry section and has not started executing the exit section.

The recovery, entry, and exit sections are expressed as subroutines of the RME algorithm performed by the processes executing the algorithm. A process executing an RME algorithm can take three types of *steps*: an ordinary step, a flush step or a crash step. An ordinary step is an operation in which a process accesses exactly one shared variable. A flush step is a persistence instruction that writes a cache line back to the PM; it can also be triggered spontaneously by the system. A crash step represents an individual crash failure, after which a process is reset to its initial state, including restarting from the NCS. I also consider system-wide failures as another type of crash steps that are not associated with any specific process, in which all processes simultaneously perform a crash step and all *unflushed* values are lost. Each process has a *program counter*, a private variable that indicates the next ordinary or flush step in the RME algorithm to be executed by the process. Initially, all program counters point to the NCS. A program counter of a process is advanced at every ordinary or flush step, and reset by every individual crash step, taken by that process, or a system-wide crash step.

2.5.1 Histories

RME algorithm executions are represented as *histories*. A history is a sequence of steps taken by the processes executing the RME algorithm. A *passage* is a sequence of steps taken by some process from when the process starts the recovery section until it completes the exit section, or crashes, whichever occurs first. A process p_i is said to have completed a *passage*, in some history H, if all the ordinary and flush steps from the beginning of the recovery section to the end of the exit section are in H, or p_i crashes outside the NCS in H. A *super-passage* is a maximal sequence of consecutive passages by the same process of which only the last one can be free of failures. A super-passage is said to be *complete* only if there exists a crash-free passage in that super-passage. Otherwise, the super-passage is said to be *pending*. A process p_i is said to be *executing a super-passage* if and only if it has started a super-passage and has not yet completed it. Following the model in [38], I assume that if a process crashed while executing a (pending) super-passage, it is repeatedly restarted until it completes the super-passage.

A process is said to be *active* in some history H if the process executes at least one ordinary step in H. The length of a finite history H is the number of steps in H, denoted by

$|H|$. A history H is said to be *fair* if and only if H is finite, or H is infinite and each process is not active, or halts while not executing a super-passage, or executes infinitely many steps. For analysis simplicity the CS is assumed to be bounded. A history H is considered *valid* if it can be produced by processes, subject to independent crashes, executing the RME algorithm.

2.5.2 Complexity Measures

A *remote memory reference* (RMR) is defined based on the multi-processor architecture [74]. In the CC-NUMA model, described in Section 2.1, a process stores a local copy of a variable in its local cache for faster subsequent access to the data. If the variable data is modified by another process, caches with stale versions of the variable data are invalidated or updated, incurring an RMR. A subsequent read operation by a process of the data invalidated by another process also incurs an RMR.

In this thesis, when analyzing the upper bound for RMR complexity of an RME algorithm, I conservatively assume that every read and write operation incurs an RMR, with the exception of *simple read* operations. A simple read is when some process reads the same variable it has read earlier, and no other process writes to that variable in between the two read operations.

2.5.3 Correctness Properties

In [38], Golab and Ramaraju defined several correctness properties for an RME algorithm. The following are the ones considered in proving RME algorithms in this work:

Definition 2.1. *Mutual Exclusion (ME).* At the end of any finite history H, a process p_i is in the CS only if no process p_j, where $i \neq j$, is in the CS.

Mutual exclusion ensures that only one process can enter and execute the critical section at any given time. This prevents multiple processes from accessing shared resources simultaneously, avoiding race conditions.

Definition 2.2. *Terminating exit (TE).* For any infinite fair history H, any execution of the exit section by a process p_i completes in a finite number of p_i's steps, or else there are infinitely many crash steps in H.

Terminating exit ensures that if a process started executing the exit section, the process eventually completes the section.

Definition 2.3. *Deadlock-Freedom (DF).* For any infinite fair history H, if a process p_i is in the recovery or entry section after some finite prefix of H, then eventually some process p_j (possibly $j \neq i$) is in the CS, or else there are infinitely many crash steps in H.

Deadlock-freedom guarantees that even when multiple processes are competing for a lock, at least one will always be able to make progress and acquire it. This avoids scenarios where two or more processes are stuck indefinitely, waiting for each other to release locks they hold, preventing the system from grinding to a halt.

Definition 2.4. *Starvation-Freedom (SF).* For any infinite fair history H, if a process p_i is in the recovery or entry section after some finite prefix of H, then eventually p_i is in the CS, or else there are infinitely many crash steps in H.

Starvation-freedom ensures that every process attempting to obtain a lock will eventually succeed in doing so. It prevents certain processes from being perpetually overlooked or blocked while others keep acquiring the lock, thereby ensuring fairness. In essence, it guarantees that no process is left *starved* without access to the critical section. This property implies DF.

Definition 2.5. *Critical Section Re-entry (CSR).* For any history H and for any process p_i, if p_i crashes inside the CS then no other process may enter the CS before p_i re-enters the CS after the crash failure under consideration.

This property guarantees that if a process p_i crashes while it is inside the CS, no other process can enter the CS until p_i has recovered and re-entered the CS. In other words, after a crash, p_i is given priority to complete its operations within the CS before any other process is allowed to enter. In some applications, CSR ensures that p_i's work remains consistent and is not interrupted by other processes, even in the case of a failure.

2.6 Detectable and Persistent Data Structures

Informally, a data structure is considered *persistent* if it retains its state even after crash failures, enabling operations to resume smoothly upon recovery. In a persistent concurrent data structure, updates made by a thread are flushed to be preserved following a crash, ensuring that the state of the data structure upon recovery accurately reflects all operations that were completed before the failure occurred.

One challenge with persistent data structures is determining the outcome of the last operation invoked before a crash, as the result is usually stored in volatile CPU registers. Consider a lock-free linked list as an example. The insertion of such a list is shown in Algorithm 2.2. The algorithm first creates a new node with the given value and traverses

Procedure Insert(*head*, *value*)

```
7  n := new Node(value)                          // Create a new node with the given value
8  while True do                                 // Keep searching until insertion succeeds
       // Traverse the list to find the correct position
9      pred := head
10     curr := head.next
11     while curr ≠ ⊥ ∧ curr.value < value do
12         pred := curr
13         curr := curr.next
14     n.next := curr                            // Set the new node's next to point to the current node
       // Attempt to link pred to the new node using CAS
15     if CAS (pred.next, curr, n) then
16         return true                           // Insertion successful
17     else
18         continue                              // CAS failed, retry the insertion
```

Algorithm 2.2: Lock-Free Linked List Insertion

the list to locate the appropriate position for insertion. It then employs a *Compare-And-Swap* (CAS) operation to atomically modify the *next* pointer of the predecessor node, linking it to the new node. If the CAS operation fails due to concurrent modifications by another thread, the algorithm retries the insertion by restarting from the head of the list. Now, if a thread is in the middle of performing a CAS operation to insert a new node into the list just before a crash occurs, the thread needs to be able to identify what happened to that operation upon recovery. If the insertion did not successfully complete, the thread should be able to retry the operation with the same node. Informally, a data structure is considered *detectable* if it allows a thread to determine the result of its most recent operation before the crash [34].

2.6.1 Modeling Shared Objects

In this thesis, I approach challenges related to concurrent data structures by using a model inspired by the work of Herlihy and Wing [41]. A concurrent algorithm in this context is executed by N independent processes, or threads, labeled $p_1, \ldots, p_N$, which interact with each other via *shared objects* using basic memory access operations. These processes can access PM through atomic memory instructions, such as reading, writing, and read-modify-write operations. The internal state of each process, including elements like the program counter and stack pointer, is stored in the processor's volatile registers.

Each shared object is associated with a unique identifier and is defined by its data type, which offers a set of operations that dictate how the object's state can be modified. Once a process initiates an operation on a shared object, it must wait for the operation to complete before initiating another, assuming no system crash occurs in the meantime. Multiple processes can concurrently invoke operations on the same shared object.

Operation executions are modeled through two key events, the invocation and the response. The invocation, represented as $\langle INV, p_i, X, op \rangle$, indicates that process p_i has called operation op on object X. The response event, denoted as $\langle RES, p_i, X, r \rangle$, marks when p_i receives the result r from the operation on X. For atomic operations, the invocation and response are considered to happen simultaneously and can thus be treated as a single event.

2.6.2 Data Types and Sequential Specifications

An object is characterized by a *data type*, which defines the set of possible values that represent its state and provides a set of operations that are the sole means of interacting with or modifying that state. The *sequential specification* of an object X belonging to a data type T defines the behavior when an operation op is performed on X.

In this section, I formalize the sequential specification of an object based on the axiomatic style introduced in [41]. In this approach, the sequential specification uses a state transition relation to represent the object's state before an operation is invoked and after its response. A data type T is formally defined by its sequential specification [41], which includes:

- a set S representing all possible states for an object X of type T,
- an initial state $s_0 \in S$,

- a set OP of operations provided by T that may or may not alter X's state,
- a set R containing all possible return values for the operations in OP,
- a state transition function $\delta : S \times OP \to S$, and
- a response function $\rho : S \times OP \to R$.

When $\delta(s, op) = s'$, it implies that after invoking the operation op on an object in state s, the object transitions to the new state s'. Similarly, $\rho(s, op) = r$ indicates that invoking operation op on an object in state s will return the result r, which could be a value or a void response, to the invoking process.

In this axiomatic style, each axiom consists of a precondition, the operation invocation, and a post-condition. The precondition specifies the object's state before the operation is performed, while the post-condition describes the object's state and the result after the operation completes. When an operation with signature op is executed by a process p on an object X, it is denoted as $[X, op/r, p]$. If the process p and object X are implicit from the context, they may be omitted. The sequential specification for an unordered set, also referred to as a set, which supports the operations **Insert**, **Remove**, and **Contains**, is presented in Figure 2.2.

Axiom 2.1 defines the behavior of a successful **Insert** operation. The precondition states that before calling **Insert**(e), the element e is not present in the set s, meaning the operation can add the element. The operation is denoted as [**Insert**(e)/**true**], where **Insert**(e) is executed, and the result is **true**, indicating a successful insertion. The post-condition defines that the set is updated to s', where $s' = s \cup \{e\}$, meaning the element e is added to the set. Axiom 2.2 describes a failed insertion. In this case, the precondition $\{e \in s\}$ indicates that e is already in the set, and thus, the operation **Insert**(e) cannot add it again. Invoking the operation returns **false**, meaning the insertion failed, as the element was already present. Axiom 2.3 describes the behaviour of the **Remove**(e) operation regardless of the state of the set as the precondition is **true**. After calling **Remove**(e), the result is the predicate function $P(e \in s)$, which evaluates to **true** if the element e exists in the set, and returns **false** otherwise. The post-condition dictates that if e is in the set $(e \in s)$, the element is removed, and the set is updated to $s' = s \setminus \{e\}$. Finally, axiom 2.4 defines the behavior of the **Contains** operation. The precondition is always true, allowing **Contains**(e) to be called at any time. After calling the operation, the result is **true** if e is in the set, or **false** if e is not in the set, without modifying the state of the set.

$$\begin{gathered}\{e \notin s\} \\ [\textbf{Insert}(e)/\textbf{true}] \\ \{s' = s \cup \{e\}\}\end{gathered} \tag{2.1}$$

$$\begin{gathered}\{e \in s\} \\ [\textbf{Insert}(e)/\textbf{false}] \\ \{\}\end{gathered} \tag{2.2}$$

$$\begin{gathered}\{\textbf{true}\} \\ [\textbf{Remove}(e)/P(e \in s)] \\ \{s' = s \setminus \{e\}\}\end{gathered} \tag{2.3}$$

$$\begin{gathered}\{\textbf{true}\} \\ [\textbf{Contains}(e)/P(e \in s)] \\ \{\}\end{gathered} \tag{2.4}$$

Figure 2.2: Axioms for unordered set operations

2.6.3 Histories

The execution of a concurrent data structure algorithm is modeled as a series of events, referred to as a *history*. A history comprises invocation, response, flush, and crash events. A *sub-history* of a history H is simply a sub-sequence of events from H. For an invocation event I in history H, the first subsequent response event R, executed by the same process p on the same object X and prior to any further invocations by p on X, is called the *matching* response of I. If an invocation does not have a matching response, it is said to be *pending* in the history H. A history H is called *sequential* if:

1. every response event immediately follows the invocation it matches, and
2. every invocation event immediately precedes its matching response.

Sequential histories are composed only of operation events and do not include crash events. A history H is said to be *fair* if and only if H is finite, or H is infinite and each process executing an operation takes a step in the algorithm.

As outlined in Section 2.2, cache line write-backs may occur at unpredictable times, even without explicit flush instructions. In addition, cache write-backs may be explicitly

triggered by executing a flush instruction, denoted **Flush**(). All cache line write-backs in a history are represented as *flush events*.

In line with prior work, I focus solely on system-wide crash failures, defined in Section 2.3, when considering algorithms for concurrent data structures. *Crash events* represent when a failure occurs and the system's subsequent recovery.

For a given process p, the process sub-history of a history H, denoted $H|p$, refers to the sub-sequence of events in H that involve p. Similarly, for an object X, the object sub-history $H|X$ refers to the sub-sequence of events in H that pertain to X. Since crash events impact all processes in the system, both $H|p$ and $H|X$ include all crash events. Two histories H and H' are considered *equivalent* if, for every process p, the sub-histories $H|p$ and $H'|p$ are identical. A history H is considered *well-formed* if, for every process p:

1. every response event in $H|p$ immediately follows the invocation it matches, and
2. every invocation event in $H|p$, except potentially the last one, immediately precedes its matching response or a crash event.

Unlike sequential histories, well-formed histories may contain crash events. All histories in concurrent data structure algorithms considered in this thesis are assumed to be well-formed.

An operation op in history H consists of an invocation event I and, if it exists, its matching response R. If no matching response is present, the operation op consists solely of the invocation event I. An operation is considered *complete* if the matching response exists and *pending* if no matching response is present. Let op_0 and op_1 be two complete operations in H. We say that operation op_0 *happens before* op_1 if the response event for op_0 occurs before the invocation event for op_1 in H. This relation, called the happens-before relation, defines an irreflexive partial order on the operations in H, denoted $<_H$. If neither $op_0 <_H op_1$ nor $op_1 <_H op_0$ holds, then op_0 and op_1 are considered *concurrent*, otherwise, the operations are *sequential*. For sequential histories, the order $<_H$ is total.

As explained in Section 2.6.2, each object has a type that specifies its abstract state and the effects of various operations on that state. A type can be formally described using a sequential specification. A sequential history H is deemed *legal* if, for every object X of some type T, the operations' invocations and their matching responses in $H|X$ are consistent with a valid sequence of state transitions allowed by the sequential specification of T.

2.6.4 Correctness Properties

Herlihy and Wing's *linearizability* [41] has been widely used as a correctness property for concurrent data structures. Informally, a shared object is *linearizable* if each operation, in every possible concurrent execution, *takes effect* at some point in time between its invocation and response, called the *linearization point*, such that if operations occur atomically at their corresponding linearization points, they constitute a legal sequential execution. As a result, concurrent operations may take effect in arbitrary order, while the order between non-overlapping operations is preserved.

Definition 2.6. Given a history H, its *completion* H' is a modified version of H such that:

1. for some pending invocations in H, matching responses are added, and
2. the remaining pending invocations in H are removed.

Definition 2.7. *Linearizability* [41]. A well-formed history H that does not include crash events is considered linearizable if there exists a completion H' of H and a legal sequential history S, such that:

1. H' is equivalent to S, and
2. for any operations op_1 and op_2, if op_1 happens-before op_2 in H' ($op_1 <_{H'} op_2$), then op_1 also happens-before op_2 in S ($op_1 <_S op_2$).

Linearizability requires that the happens-before relationship in the completion history H' must be preserved in the sequential history S. A correctness property P is said to be *local* if a history H satisfies P provided that, for every object X accessed in H, the sub-history $H|X$ satisfies P. Linearizability is an example of a local property.

The definition of linearizability does not include crash-restart failures. Therefore, more variants of the correctness property have been proposed to accommodate failures [10]. *Strict Linearizability* [1] requires crashed operations to either take effect before the crash or never take effect. The property is valid under the system-wide and individual process crash failure models.

Definition 2.8. Given a history H, its *strict completion* H' is a modified version of H such that:

1. for some pending invocations in H, matching responses are added prior to the next crash event, if one occurs, or at the end of H if no crash event takes place, and

2. the crash events and remaining pending invocations in H are removed.

Definition 2.9. *Strict Linearizability* [1]. A well-formed history H is considered strictly linearizable if there exists a strict completion H' of H and a legal sequential history S, such that:

1. H' is equivalent to S, and
2. for any operations op_1 and op_2, if op_1 happens-before op_2 in H' ($op_1 <_{H'} op_2$), then op_1 also happens-before op_2 in S ($op_1 <_S op_2$).

Strict linearizability mandates that, following a crash, any pending operation is prevented from taking effect post-crash. Strict linearizability is also a local property.

2.6.5 Progress Properties

Given the model of shared objects described in this thesis, I define *recoverable deadlock-freedom* as the progress property, as presented in Definition 2.10.

Definition 2.10. *Recoverable deadlock-freedom.* Given a fair infinite history H, if a process p_i executes a pending operation O in H, there exists a process (possibly p_i) that completes operations, by reaching a matching response event, infinitely many times in H, or a system crash occurs after the invocation of O.

2.6.6 Detectable Unordered Set

As in previous works [56, 60], I provide a formal description of detectability by deriving a detectable sequential specification from an object's sequential specification. Precisely, I adopt a simplified version of the *unified detectable sequential specification* (UDSS) proposed by Moridi et al [60]. The key distinction between UDSS and its simplified variant is that the latter comprises detectable operations only for operations that may change the object's state, and non-detectable operations only for operations that do not change the object's state, e.g., read-only operations.

Consider a shared object type $T = (S, s_0, OP, R, \delta, \rho)$, where S represents a set of possible states, s_0 is the initial state, OP is the set of operation signatures, R is the set of responses, δ is the state transition function, and ρ is the response function. The *simplified unified detectable sequential specification* (SUDSS) of T is a sequential specification $SUD\langle T\rangle = (\bar{S}, \bar{s_0}, \bar{OP}, \bar{R}, \bar{\delta}, \bar{\rho})$, derived from T as follows:

$$\begin{gathered}\{\textbf{true}\} \\ [\textbf{Detectable-}op(t)/\rho(s, op)] \\ \{s' = \delta(s, op) \wedge \mathcal{A}[p_i] = op \wedge \mathcal{T}[p_i] = t \wedge \mathcal{R}[p_i] = \rho(s, op)\}\end{gathered} \tag{2.5}$$

$$\begin{gathered}\{\textbf{true}\} \\ [op/\rho(s, op)] \\ \{s' = \delta(s, op)\}\end{gathered} \tag{2.6}$$

$$\begin{gathered}\{\textbf{true}\} \\ [\textbf{Resolve}()/(\mathcal{A}[p_i], \mathcal{T}[p_i], \mathcal{R}[p_i])] \\ \{\}\end{gathered} \tag{2.7}$$

Figure 2.3: Simplified (SUDSS) of base type T, denoted $SUD\langle T\rangle$.

- Each state $\bar{s} \in \bar{S}$ is structured as a tuple $(s, \mathcal{A}, \mathcal{T}, \mathcal{R})$, where $s \in S$, and the mappings $\mathcal{A}$, $\mathcal{T}$ and $\mathcal{R}$ associate processes with operation signatures, tags and responses, respectively.
- In the initial state $\bar{s_0}$, $s = s_0$, and each of $\mathcal{A}$, $\mathcal{T}$ and $\mathcal{R}$ assign $\perp$ to every process.
- For each operation $op \in OP$, $\bar{OP}$ includes
  - **Detectable-**op, if $\exists s \in S \mid \delta(s, op) \neq s$
  - op if $\forall s \in S \mid \delta(s, op) = s$
  - An auxiliary operation, called **Resolve**
- The state transition function $\bar{\delta}$, response function $\bar{\rho}$, and the response set $\bar{R}$ are outlined in Figure 2.3, using an axiomatic approach similar to what is described in Section 2.6.2.

Informally, in Axioms 2.5, for each $op \in OP$ of type T, if op may change the state of the shared object, the operation **Detectable-**$op \in \bar{OP}$ is defined to apply op in a detectable manner. If op does not change the state of the base object, then detectability is not needed and only op is in $\bar{OP}$ (Axiom 2.6). Much like UDSS, the SUDSS enables detectability as follows: if a process calls **Resolve**, defined in Axiom 2.7 immediately after executing **Detectable-**$op(t)$ with some tag t and response r, then **Resolve** will return

(op, t, r). If no detectable operation was applied by the process, then **Resolve** will return $(\perp, \perp, \perp)$.

From the sequential specification of the unordered set type (*Set*), defined in Figure 2.2, I apply the SUDSS transformation to derive the sequential specification for the detectable variant ($SUD\langle Set\rangle$). The sequential specification for the type $SUD\langle Set\rangle$ is shown in Figure 2.4. The specifications of $SUD\langle Set\rangle$ resemble those of *Set*, with the following key differences:

1. The **Insert** function (Axioms 2.1 and 2.2) is replaced by its detectable variant **Detectable-Insert** (Axioms 2.8 and 2.9). The post-condition shows that the maps $\mathcal{A}$, $\mathcal{T}$ and $\mathcal{R}$ of the executing process p_i are updated to the operation signature **Insert**(e), tag t and result $P(e \notin s)$, respectively.

2. The **Remove** function (Axioms 2.3) is replaced by its detectable variant **Detectable-Remove** (Axioms 2.10). The post-condition shows that the maps $\mathcal{A}$, $\mathcal{T}$ and $\mathcal{R}$ of the executing process p_i are updated to the operation signature **Remove**(e), tag t and result $P(e \in s)$, respectively.

3. An auxiliary function **Resolve** is introduced (Axiom 2.12) to allow any process p_i to check the last operation it executed. If the last operation executed by p_i is [**Detectable**-$op(t), r$], then invoking **Resolve** will return (op, t, r). If p_i has not executed any operation, then calling **Resolve** will return $(\perp, \perp, \perp)$.

2.7 Background Algorithms

2.7.1 Unrolled Linked List

In 2020, Platz et al. [62] proposed a lock-and-chunk-based list known as the unrolled linked list, which they demonstrated to be up to 300% faster, in terms of throughput, than lock-free linked lists. To the best of my knowledge, the unrolled linked list [62] is the fastest linked list-based set structure reported in the literature. This section provides an overview of how the algorithm operates. An unrolled linked list consists of nodes, each having the following attributes:

- *count*: the number of key-value pairs stored in the node.

$$\{e \notin s\}$$
$$[\textbf{Detectable-Insert}(t, e)/\textbf{true}] \tag{2.8}$$
$$\{s' = s \cup \{e\} \wedge \mathcal{A}[p_i] = \textbf{Insert}(e) \wedge \mathcal{T}[p_i] = t \wedge \mathcal{R}[p_i] = \textbf{true})\}$$

$$\{e \in s\}$$
$$[\textbf{Detectable-Insert}(t, e)/\textbf{false}] \tag{2.9}$$
$$\{\mathcal{A}[p_i] = \textbf{Insert}(e) \wedge \mathcal{T}[p_i] = t \wedge \mathcal{R}[p_i] = \textbf{false})\}$$

$$\{\textbf{true}\}$$
$$[\textbf{Detectable-Remove}(t, e)/P(e \in s)] \tag{2.10}$$
$$\{s' = s \setminus \{e\} \wedge \mathcal{A}[p_i] = \textbf{Remove}(e) \wedge \mathcal{T}[p_i] = t \wedge \mathcal{R}[p_i] = P(e \in s)\}$$

$$\{\textbf{true}\}$$
$$[\textbf{Contains}(e)/P(e \in s)] \tag{2.11}$$
$$\{\}$$

$$\{\textbf{true}\}$$
$$[\textbf{Resolve}()/(\mathcal{A}[p_i], \mathcal{T}[p_i], \mathcal{R}[p_i])] \tag{2.12}$$
$$\{\}$$

Figure 2.4: Axioms for detectable unordered set $SUD\langle Set\rangle$ operations

- *anchor*: the key that uniquely characterizes a node. If a key exists in a node, then the key $\geq$ the *anchor* of that node and $<$ the *anchor* of the next node. Hence, nodes are arranged in ascending order based on their anchor keys.
- *next*: a pointer to the next node in the list.
- *pairs*$[K]$: an array of unordered K key-value pairs, where K is a constant representing the maximum number of pairs allowed in a node.
- *marked*: a boolean flag that indicates whether the node has been logically deleted (removed from the list).
- m: a mutex lock used only during updates (e.g., insertions and removals) in the list.

An unrolled linked list has at least three nodes: the *head* sentinel node, which is the first node in the list where the *anchor* $\leq$ keys in the key space; the tail sentinel node, which

is the last node in the list where the *anchor* $\geq$ keys in the key space; and at least one non-sentinel node between them, which holds key-value pairs. The head and tail sentinel nodes do not store key-value pairs. However, the non-sentinel nodes store key-value pairs. Thus, an unrolled linked list always consists of at least these three nodes.

There are eight procedures that constitute the algorithm. The unrolled linked list is a concurrent implementation of the set ADT. A set supports three operations: **Contains**, **Insert**, and **Remove**. I summarize the eight procedures by describing the three set operations.

Operation Contains

A **Contains** operation searches for a given key in the list and returns the corresponding value if the key is found, or returns a special value ($\perp$) if the key is not found. Searching in an unrolled linked list is divided into two procedures. The first is **Scan**, where the algorithm searches for the node that might contain the target key (i.e., the node's *anchor* $\leq$ *key* $<$ the successor node's *anchor*). The procedure then returns that node, along with its predecessor and successor. The second procedure is **Seek**, which linearly searches the array of the node for the target key. If the target key is found, the procedure returns the index of the pair; otherwise, it returns $\perp$. If **Seek** returns a valid index, the key-value pair is read atomically. If the found key still equals the target key, the corresponding value is returned by **Contains**. If **Seek** returns $\perp$, then **Contains** returns $\perp$.

Operation Insert

An **Insert** operation first scans the list, using the **Scan** procedure, to locate the appropriate node for inserting a new key-value pair based on the anchor keys. Once a suitable node is found, the lock m of that node's predecessor is acquired to prevent concurrent updates (**Insert** and **Remove**) on the same node and to prevent removing the predecessor node. Then, the algorithm checks that neither the current node nor the predecessor node is *marked* deleted, and that the predecessor's *next* still points to the current node. This validation step was originally proposed by Heller et al. [40]. If the validation is successful, the operation proceeds; otherwise, the predecessor's lock is released, and the entire operation is repeated. Next, the **Insert** operation seeks the current node (found by **Scan**) to verify if the key to be inserted already exists (i.e., if **Seek** returns a valid index). If the key already exists, **Insert** returns false; otherwise, the operation continues. The algorithm then looks for an empty slot to store the new pair atomically. If no empty slot is found, the current

node's lock is acquired, the node is split into two nodes (by invoking the **Split** function), and the key-value pair is added to one of the new nodes based on their anchor keys. The **Split** operation creates two new nodes: one receives the floor of half the key-value pairs from the current node, and the other receives the ceiling of the half. The current node is then replaced by these two newly created nodes.

Operation Remove

A **Remove** operation scans the list to locate the node that may contain the target key, acquires the predecessor's lock m, and performs Heller's validation, identical to the steps in the **Insert** operation. Then, the **Remove** operation seeks the key to be deleted in the current node (identified by **Scan**). If the key is found in the node's array, the key's slot is cleared; otherwise, the operation returns false. After a key is deleted, the **Remove** operation may perform additional actions under the following conditions:

1. The current node has no elements: In this case, the node's lock m is acquired, the node is marked as deleted (unless it is the only non-sentinel node), and all locks are released.

2. The current node contains very few elements.[1] Then, either:

   (a) The total number of elements in the current and successor nodes is greater than or equal to a predefined threshold (referred to as MaxFull in [62]). In this case, the elements from the current and successor nodes are redistributed between the two nodes through the **Redistribute** procedure. This procedure creates two new nodes, finds the M-th smallest key (key_M) in the successor node (where $M = \lceil (current.count + successor.count)/2 \rceil - current.count$), copies all key-value pairs with keys $\leq key_M$ from the current and successor nodes to one new node, and copies the remaining pairs to the other new node. The current node and its successor are then replaced by the two newly created nodes. The value $M \geq 0$ as $successor.count > current.count$ (see Section 5.3.4 for more details.)

   (b) The total number of elements in the current and successor nodes is less than MaxFull: In this case, the current node and its successor are merged into a single node through the **Merge** procedure. The **Merge** procedure creates a new node, sets its anchor to that of the current node, copies all key-value pairs

[1] If the current node's *count* is less than a predefined threshold, referred to as MinFull in [62].

from the current and successor nodes into the new node, and then replaces the current and successor nodes with the new node.

Otherwise, no further action is required, and the **Remove** operation returns true. In all cases (1, 2a, and 2b), the mutexes of the current node and its predecessor are acquired. Additionally, the lock of the successor node is required in cases 2a and 2b. Afterward, all locks are released, and the **Remove** operation returns true.

Chapter 3

Related Work

Various recoverable concurrent data structures have been developed in the literature [70, 19, 7, 46, 21, 34, 69, 4]. Researchers have established recoverability with different techniques. Volos et al. [70] and Coburn et al. [19] employed redo and undo logging to track threads' progress prior to a crash. A persistent hybrid algorithm was proposed by Brown and Avni [7] that utilizes software and hardware transactional memory in PM. These approaches depend on logging, which comes with a significant performance overhead. An efficient logging technique, JUSTDO, presented by Izraelevitz et al. minimizes the log size via tracking the most recent store instruction only, enabling instant operation recovery after a crash without the need for rolling back [46]. JUSTDO assumes non-volatile caches, which requires additional specific hardware to flush caches on system-wide failures. Other researchers designed log-free recoverable concurrent data structures [21, 34, 69, 4]. David et al. [21] presented set of approaches to construct log-free persistent data structures from their volatile original algorithms. Friedman et al. [34] presented three novel concurrent lock-free queue algorithms with different persistence guarantees. Venkataraman et al. [69] developed a persistent concurrent B-Tree using a log-free versioning scheme that enables instant roll-back. Attiya et al. [4] provided a generic tracking technique aimed at lock-free data structures that achieves recoverability. Dali is a persistent hashmap proposed by Nawab et al. [61] that relies on periodic persistence rather than flushing every update to the PM. The idea is to adopt a weaker correctness property, buffered durable linearizability [47], for better overall performance. Some data structures are said to be detectable [34, 6, 8, 9], which is an additional feature on top of recoverability that enables detecting the response of a crashed operation. In, [6] and [9], Attiya et al. and Ben-David et al. presented a detectable persistent CAS synchronization primitive that uniquely identifies each operation. One of the queues proposed by Friedman in [34] offered detectability at the expense of using

bookkeeping logs. The lower and upper bound of the space complexity associated with achieving detectability was investigated by Ben-Baruch et al. [8]. They also presented a detectable CAS with bounded space complexity.

Until recently, there was no formal definition for detectability. Attiya et al. in [6] defined detectability through a correctness property they have proposed, *nesting-safe recoverable linearizability* (NRL). NRL assumes that each process has access to an auxiliary persistent object that stores the program counter, indicating the next operation a process was about to execute before a crash. The applicability of such an assumption is not clear in a practical setting. Li and Golab [56] proposed a more robust definition by providing a systematic approach of obtaining a *detectable sequential specification* (DSS) of an object type from its non-detectable sequential specification. Using DSS, [56] presented a detectable lock- and log-free queue. Based on Li and Golab's work, Moridi et al. [60] proposed a new correctness definition, called *unified detectable sequential specification* (UDSS), that has a simpler code base and object's interface. In addition, they presented a black box approach that transforms a DSS object into a UDSS one.

Researchers are currently active in developing persistent and detectable lock-free algorithms [4, 5, 75, 45, 70, 33, 35]. Attiya et al. built on top of the work in [4] and presented in [5] a technique to obtain a detectable persistent implementation of lock-free data structures. Zuriel et al. [75] designed an efficient persistent concurrent set. Izadpanah et al. [45] proposed PETRA, an optimization to *persistent transactional memory* (PTM) introduced by Volos et al. [70]. NVTraverse [33] is a general transformation by Friedman et al. that automatically makes a lock-free data structure persistent. However, it requires changing its representation into *a traversal form*, which needs programming expertise and potentially degrades performance [35]. Friedman et al. [35] presented another transformation where a copy of the base data structure is stored in each of the DRAM and PM, resulting in a faster lock-free recoverable data structures than NVTraverse.

Published lock-based detectable and persistent data structures are not as common nor practical as lock-free structures. Atlas [15], by Chakrabarti et al., identifies failure-atomic regions using locks in lock-based data structures to simplify recovery, but logs are used intensively in critical sections which degrades the performance and recovery time. Ching-Hsiang et al. [43] exploit mutexes to coarsely divide the code base into synchronization points of consistent states, then record buffered writes between points to reduce the overhead of logging. Nawab's Dali hashmap [61] is lock-based, but it depends on the flush-the-whole-cache privileged x86 instruction and tends to flush and invalidate useful lines. A detectable lock-based flat-combining technique [66] was recently proposed by Rusanovsky et al., which is a generic way to construct detectable lock-based data structures. Flat-combining techniques involve one thread to perform operations submitted by other

threads on the system. Such an approach utilizes the cache more efficiently but limits scalability.

Achieving detectability and recoverability when designing lock-based data structures is challenging. According to [35], "...lock-free data structures are a natural choice for being adapted to persistence." One of the reasons is the challenges associated pertaining locks' correctness properties (e.g. mutual exclusion and deadlock-freedom) when imposing persistence. This problem was recently formalized by Golab and Ramaraju as the novel *recoverable mutual exclusion* (RME) problem that considers crash failures [38]. Several RME algorithms have been published in the literature [37, 36, 51, 49, 23, 38, 16, 48, 52, 55]. In [36], Golab and Hendler's (GH) RME algorithm was proposed to solve the RME problem in CC systems with a sub-logarithmic *Remote memory reference* (RMR) complexity. RMR complexity is a standard complexity measure for RME algorithms [17]. An RMR is a costly memory operation that uses the interconnect between processor and memory [37]. The RMR complexity of an RME algorithm is the number of RMRs incurred by a single process while acquiring and releasing a lock once [11]. The GH [36] algorithm provides fault tolerance against failures of one process at a time. However, a new transformation of GH was introduced in [37] to accommodate simultaneous (system-wide) failures of processes. Another sub-logarithmic algorithm is Jayanti, Jayanti, and Joshi's (JJJ) [49], which guarantees the same complexity bound as GH in both *distributed shared memory* (DSM) and CC systems. Dhoked and Mittal [23] proposed an adaptive approach that improves performance, compared to previous works, in case of frequent failures. The idea is to use a (weaker) variant of the RME problem that violates the ME property, allowing multiple threads to be in their critical section simultaneously but in a controlled manner, and use that variant as a building block. A recent publication by Chan and Woelfel [17] proved that a deadlock-free n-process RME algorithm has a tight bound of sub-logarithmic RMR complexity, showing that GH [36] and JJJ [49] are optimal.

In the literature, many ME algorithms are optimized for NUMA systems [58, 64, 57, 26, 27, 13, 14, 54, 24, 25, 53], including the Hierarchical CLH (HCLH) lock developed by Luchangco, Nussbaum, and Shavit [58]. The HCLH algorithm maintains the desirable properties of CLH while adding a hierarchical structure to favor handing over locks to processes of higher memory locality. The idea of hierarchical locks was initially proposed, by Radović and Hagersten, in [64] as the hierarchical BO (HBO) lock. Though, HBO locks are highly biased to memory locality. Processes from the same node/cluster are more likely to consecutively acquire the lock, letting processes of other clusters starve. Hence, HCLH queue-based locks provide better performance and higher fairness but have a higher memory footprint [58].

In [57], Lozi et al. proposed a *remote core locking* (RCL) technique that minimizes the

number of cache misses incurred when threads contend for a lock. The idea is to utilize a dedicated *server* core on which a server thread executes the CS instead of the application threads that want to execute the CS, called *client* threads. Server threads have access to the *requests* array of size $C \times L$, where C is the max number of client threads, and L is the cache-line size. The body of the CS is represented as a function called the CS function. Client threads write 3-word requests into the request array. Requests comprise the addresses of the lock to be acquired, the *context* (an encapsulation of the variables accessed in the CS, declared by the CS function), and the CS function. Subsequently, clients spin on the CS function address word until the CS function is executed by a server thread. Server threads iterate through the request array, looking for a non-`NULL` CS function address. Once a server thread finds a CS function address, it acquires the lock referenced by the address in the first word of the corresponding request. Subsequently, the server thread executes the CS function, and then changes the CS function address word in the request to a `NULL` value, signaling the client to stop spinning.

Dice, Marathe, and Shavit, proposed in [26] a hierarchical MCS lock based on flat-combining, where threads located in the same socket construct a local queue efficiently, then merge local queues into a global one with minimal cross-socket traffic. In [27], they generalized designing two-level hierarchical locks by introducing a technique that transforms any classic lock that is NUMA-oblivious into a NUMA-aware lock while preserving its liveness and safety properties. The two-level hierarchical design consists of a global lock, called the *cohort* lock, and a local lock per socket. Local locks are acquired by processes on the same node, while the global lock is acquired by processes on different nodes. Processes on the same node hand over the ownership of the node's local lock until the global lock is released for processes on a different node to acquire. The global lock ownership is handed over between different nodes periodically, every predefined number of CS executions, to establish long-term fairness. This approach is a trade-off between performance and short-term fairness. Based on the cohort technique, Chabbi, Fagan, and Mellor-Crummey in [13] proposed a multi-level hierarchical lock (HMSC) for NUMA systems with a deeper memory hierarchy (i.e., more than two levels), where it outperforms the cohort lock. An improved version of HMSC (AHMSC) was proposed in [14] that utilizes hardware transactional memory to enhance performance in various degrees of contention.

Hierarchical locks, including the cohort lock, are proportional in size to the number of nodes in a multi-socket system, making the memory footprint quite large in situations where millions of locks are used (e.g. database systems, OS kernels) [24]. In [54], Kashyap et al. designed the CST lock where per-socket lock allocations are delayed until the locks are accessed for the first time. Such a technique enhances memory utilization in systems where threads are located in a subset of the available NUMA nodes. Dice and Kogan in

[24] proposed a variation of the queue-based MCS lock that provides NUMA-awareness in a compact design called CNA. The idea is to have two queues for the waiting processes, one for the processes on the same node as the lock owner, called the main queue, and one for the rest of the processes, called the secondary queue. In addition to the size occupied by QNodes (one per thread), a CNA lock only requires a constant size of one word for the tail address of the main queue irrespective of the number of sockets in the underlying system. In contrast, previous hierarchical locks, like the cohort-MCS lock, create a lock instance for each NUMA node [27]. This makes Dice and Kogan's lock more compact than, and as scalable as, hierarchical locks. Besides, Dice and Kogan's lock requires only one atomic instruction to acquire a lock, while hierarchical locks execute multiple atomic operations to achieve the same task. As a result, CNA performs better than most earlier hierarchical locks in single-socket settings [24].

Despite the CNA algorithm's competence under high degrees of contention, the lock suffers from an overhead associated with its lock-ownership handover procedure. In [25], Dice and Kogan recently addressed this issue and proposed an adaptive lock called the *Fissile* lock. Fissile lock adds an upper level to the CNA lock hierarchy. The outer (upper) lock is a basic *test-and-set* (TAS) lock. TAS locks [3] are simple spin-locks, where each thread performs an atomic TAS operation on a global variable repeatedly until it successfully writes into that variable. TAS is faster than CNA under low contention, but it is not scalable. Fissile lock uses TAS lock as a fast path in low contention situations and uses CNA lock as a slow path when many threads fail to acquire the outer TAS lock (high contention). Thus, the Fissile lock is a NUMA-aware lock that is fast and scalable under various degrees of contention and has the benefits of both TAS and CNA locks. A similar idea was proposed by Kashyap et al. as the *Shuffle* lock [53]: a NUMA-aware lock with a small memory footprint and higher performance than CST and cohort locks. Like the Fissile lock, the Shuffle lock consists of a TAS outer lock and an MCS-like inner lock. In the Shuffle lock, one of the waiter threads in the MCS queue becomes the *shuffler*. The shuffler reorders waiter threads off the critical path of the lock, based on some enforced policy (e.g. NUMA-awareness). The Shuffle and Fissile locks are similar in design. However, a key algorithmic difference is the *bypass* feature of the Fissile lock, where threads can bypass the TAS lock consecutively, avoiding the complexity of acquiring/releasing the inner (CNA) lock. Hence, the Fissile lock is found, practically, to be faster than the Shuffle lock [25].

This work aims to investigate using locks in concurrent data structures to facilitate achieving persistence and detectability. This includes studying some performance concerns regarding current recoverable locks, validating the practicality of the current definition of the RME problem in the context of persistent data structures, and converting the fastest lock-based linked list [62] into a recoverable persistent one.

Chapter 4

A NUMA-Aware Recoverable Mutex Lock

In this section, I attempt to answer Q1 by improving the performance of RME locks on modern multi-core multiprocessor systems (defined in Section 2.1). As discussed in Section 3, RMR complexity is a standard complexity measure for RME algorithms [17]. Although published RME algorithms have comparable RMR complexity, asymptotic bounds are not accurately reflective of an algorithm's performance. The reason is that the difference in constant coefficients, which may affect performance in practical settings, is ignored in asymptotic notations. This observation motivated a recent study, by Xiao, Zhang, and Golab [73], to benchmark RME algorithms by comparing GH and JJJ locks in a real-world testbed that utilizes an existing persistent main memory component, Intel Optane Persistent Memory. The empirical performance evaluation was accomplished on a single multi-core processor. According to the study, the experiments were limited to a specific number of cores as cross-socket traffic causes dramatic performance degradation. Modern computer systems are *non-uniform memory access* (NUMA) multi-socket multi-core systems. However, the existing recoverable mutex locks are unaware of the underlying NUMA architecture.

In this section, I propose a black-box approach that transforms a NUMA-*oblivious* RME algorithm into a NUMA-aware recoverable lock. In addition, I introduce the first NUMA-aware recoverable locks by transforming two well-known RME algorithms, GH [36] and JJJ [49], into NUMA-aware variants using the proposed technique. I compare the performance of my algorithm to the existing cohort technique [27]. In multi-socket configurations, outcomes demonstrate that my approach increases the performance of the RME locks by up to 45%, and the RME locks transformed using the proposed RF lock are slower than

their non-recoverable cohort variants by up to 9%. The overhead of my algorithm is shown to be minimal when using a single socket. Furthermore, a deeper practical assessment finds that the gap in performance between GH and JJJ is due to the entry section of JJJ, not its exit section. This work has been published recently in SPAA'22 [29].

4.1 Recoverable Filter Lock

NUMA-awareness could be achieved by, intentionally, *handing over* a mutex lock between processes running on the same NUMA node for a specific number of times, after which the lock is handed over between processes on different nodes to establish long-term fairness [27, 13, 14, 24, 54, 58, 64]. This approach is a trade-off between performance and short-term fairness. Based on this idea, I propose the *Recoverable Filter* (RF) lock, which is a NUMA-aware recoverable lock that consists of a base (NUMA-oblivious) RME algorithm and an *outer filter*. The outer filter is an object that reorders processes accessing the base lock, transforming the base lock into a NUMA-aware lock. As opposed to previous works, designing an RF lock is a black-box transformation, that is, it requires no algorithm-specific modifications to the base lock. In contrast, the *top-level* lock must be *thread-oblivious*[1] when designing a cohort lock given a NUMA-oblivious (base) lock, otherwise specific changes are required to make it thread-oblivious (e.g., QNode pooling for MCS locks). This makes the RF lock easier to instantiate than the classic cohort technique.

In addition to being NUMA-aware, my RF lock introduces recoverability, making it the first of its kind. The specific properties of the RF lock are inherited from the underlying base lock. The RF lock guarantees mutual exclusion, deadlock-freedom (DF), and starvation-freedom (SF) only if the underlying base lock upholds these properties. Similarly, when considering recoverability, an RF lock can withstand individual process crashes or system-wide failures only if the underlying base lock is recoverable against these respective failures. In other words, an RF lock built on a non-recoverable mutex will not be recoverable itself, though it still retains the advantage of NUMA-awareness provided by the filter.

Before getting into the algorithm design, I establish the following assumptions: The system consists of M NUMA nodes, each identified by a unique ID, and every process p_i is aware of its corresponding node ID, denoted by n_i. Given processes p_i and p_j, if $n_i = n_j$, then p_i and p_j are running on the same NUMA node; otherwise, they are running

[1] A lock is said to be thread-oblivious if it can be unlocked by a thread that is different from the one who locked it, otherwise, it is said to be thread-aware [27].

on different nodes. The process-node binding is assumed to be static during a single execution of an RME algorithm. Furthermore, I assume that the base RME lock satisfies the *critical section re-entry* (CSR) property. The last assumption can always be satisfied by applying the transformation (Algorithm 3) proposed in [38]. However, a drawback of using the CSR transformation is that it only preserves recoverability against individual crash failures but not system-wide failures. Such a transformation is idempotent. That is, applying it to any correct RME algorithm, whether it satisfies CSR or not, results in an RME algorithm that is correct and satisfies CSR. Hence, the transformation can be applied *blindly* on the base RME lock (i.e., without having prior knowledge about the base mutex under consideration), conserving the RF lock's feature of being a black-box transformation.

4.2 Transformation of RF lock

My RF lock hands over an RME lock between processes on the same node, called the *owner node*, for a specific number of super-passages (HandoffThreshold). After this threshold is reached, ownership of the lock is passed to a different node. The adjustable HandoffThreshold parameter bounds the base lock local handovers, controlling the trade-off between aggregate throughput and short-term fairness. Initially, there is no owner node, and processes race to claim the filter ownership through a *compare-and-swap* (CAS) operation. The process that successfully claims filter ownership is referred to as the *owner process*, and its corresponding node is called the *owner node*. Processes running on the owner node hand over the RF lock and increment a node-local counter during each super-passage. Note that if a process crashes before or after incrementing its node-local counter, it may impact the performance of the RF lock, but it does not compromise its correctness. Once a process increments the node-local counter to HandoffThreshold, it hands over the filter ownership to the *next* process on a different node. If a process p_i does not find a process on a different node n_j ($j \neq i$) when attempting to hand over the filter ownership, p_i *declares* that its node n_i is *alone*. A node n_i is considered alone if it is the only node with processes attempting to acquire the RME lock. When a node is declared alone, processes on other nodes may race to claim the filter ownership through a CAS operation. This is similar to the initial state, where there is no owner process or node. The method for selecting the next process and identifying the alone node is explained in Section 4.3.2.

The two-level hierarchy of my RF lock consists of an inner lock and an outer filter. The inner lock is a NUMA-oblivious RME lock. The outer filter is an object that comprises *atomic* shared variables and associated procedures through which processes coordinate to access the base lock in a NUMA-aware manner. The procedures of a filter object include

FilterRecover, which handles the recovery of the filter object; **Filter**, which enables processes on the owner node to access the base lock; **FilterHandoff**, which manages the transfer of filter ownership among nodes; and **MarkExit**, which marks the end of the current super-passage. The pseudo-code for transforming a base RME lock into an RF lock is shown in Algorithm 4.1.

Procedure Recover

```
    // recover if crashed in FilterHandoff
19  filter.FilterRecover()
    // base RME lock recovery is integrated into Enter
```

Procedure Enter

```
    // pass the filter if n_i is owner node, claim filter ownership if no owner node
20  filter.Filter()
    // call Recover of the base RME lock
21  baseLock.Recover()
    // call Enter of the base RME lock
22  baseLock.Enter()
```

Procedure Exit

```
    // if handoff threshold reached, hand off filter or raise alone flag
23  filter.FilterHandoff()
    // call Exit of the base RME lock
24  baseLock.Exit()
    // mark the end of the current super-passage
25  filter.MarkExit()
```

Algorithm 4.1: Transformation of a base recoverable lock into an RF lock for process p_i.

The recovery routine of the RF lock includes the recovery protocol for the filter, which is expressed as the function **FilterRecover**. The **Recover** subroutine of the base lock is integrated into the entry section of the RF lock. Once a process p_i is (re)started, it enters the **Recover** subroutine. Then, p_i calls **FilterRecover**, where p_i checks whether it crashed during its current super-passage. If crashed, p_i checks the crash location relative to some specific operations. Precisely, if p_i crashed while handing over the filter ownership in **FilterHandoff** (Line 23), then it needs to take actions to ensure the RF lock's recoverability. Otherwise, no recovery is required for the filter and, thus, no action is taken in

FilterRecover.

The recovery section is followed by the entry section, where p_i first calls the **Filter** procedure. When process p_i executes **Filter**, one of the following occurs:

1. It discovers that it has already passed the filter during the current super-passage (and subsequently crashed), allowing it to pass the filter again.
2. It finds that its node n_i is the current owner node, and thus it passes the filter.
3. It detects that the owner node n_j (where $n_j \neq n_i$) is alone, or there is no current owner node, prompting it to set n_i as the new owner node and pass the filter.
4. It finds that the owner node n_j (where $n_j \neq n_i$) is not alone, causing it to spin until n_i becomes the owner node.

The remainder of the RF lock's entry section is simply an execution of the **Recover** and **Enter** procedures of the base lock.

When p_i executes the RF exit section, it calls **FilterHandoff**, in which it checks if the node-local counter reached the HandoffThreshold. If the counter has reached the threshold, then p_i attempts to hand over the filter by changing the owner process and node; Otherwise, it skips the handoff operation. Regardless of the node-local counter value, p_i increments the counter at the end of **FilterHandoff**. Afterward, p_i calls the **Exit** procedure of the base lock and, finally, completes its super-passage by executing the function **MarkExit**, which lets p_i skip the filter recovery routine during its next passage, if any.

4.3 Implementation Details

This section presents the details of each procedure in the filter object. The pseudo-code for the **FilterRecover**, **Filter**, **FilterHandoff**, and **MarkExit** procedures is provided in Algorithms 4.2, 4.3, 4.5, and 4.6, respectively. These filter procedures involve accessing the following atomic shared variables:

- *owner*: A tuple $\langle \textbf{int}\, k, \textbf{int}\, n, \textbf{Boolean}\, alone \rangle$, where k and n represent the IDs of the owner process and owner node, respectively, or $\perp$, with $alone \in \{0, 1\}$, and initially set to $\langle \perp, \perp, 1 \rangle$. The *alone* flag indicates whether the processes running on node n_k

are the only ones attempting to access the RF lock ($alone = 1$), or if there is a process p_i, with $n_i \neq n_k$, waiting for its node n_i to become the owner node ($alone = 0$) so that p_i can pass the filter and access the base lock.

- $B[1...N]$: An array of N integers, where $B[i] \in \{\perp, \text{PASSED}, \text{HANDOFF}, n_i\}$. $B[i] =$ PASSED once process p_i successfully passes the filter, $B[i] =$ HANDOFF when p_i is attempting to hand over the filter, and $B[i] = n_i$ when p_i is locally spinning (busy-waiting) on $B[i]$ while attempting to pass the filter. Initially, $B[i] = \perp$. Note that PASSED and HANDOFF are reserved integer values that are distinct from any possible value of n_i.

- $counter[1...M]$: An array of M integers, where each entry is a counter that is local to a specific NUMA node, initially 0.

- $last[1...M]$: An array of M integers, where each entry is local to a specific NUMA node. $last[n_i] = i$ if p_i is the last process on node n_i to access the base lock, with an initial value of $\perp$.

Procedure FilterRecover

```
26 if B[i] = HANDOFF then
27     ⟨k, n, alone⟩ := owner                          // read current owner
28     Flush(&owner)
29     if alone then
           // if n_i is "alone"
30         Execute lines 73-84 of FilterHandoff(⟨k, n, alone⟩)
31     else if n_i = n then
           // if ownership is not passed yet
32         Execute lines 68-84 of FilterHandoff(⟨k, n, alone⟩)
33     else
           // if ownership is passed and n_i is not alone
34         Execute lines 78-84 of FilterHandoff(⟨k, n, alone⟩)
```

Algorithm 4.2: FilterRecover algorithm for process p_i.

All variables are assumed to reside in PM. A flush instruction, defined in Section 2.2 and denoted by **Flush** in Algorithms 4.2-4.6, is applied immediately after each write to shared variables and after each read of an unflushed value.

Procedure Filter

```
    // skip the filtering operation if passed then crashed
35  if B[i] ≠ PASSED then
36      ⟨_, n, _⟩ := owner
37      if n_i ≠ n then                                  // skip if n_i is the owner node
38          Flush(&owner)
39          B[i] := n_i                                  // p_i wants to access the RME lock
40          Flush(&B[i])
41          ⟨k, n, alone⟩ := owner
42          if alone and n_i ≠ n then
                // take ownership if another node is alone
43              CAS(&owner, ⟨k, n, 1⟩, ⟨i, n_i, 0⟩)
44          ⟨_, n, _⟩ := owner                           // update owner node
45          if n_i ≠ n then
46              await B[i] = PASSED                      // spin
47      Flush(&owner)
48      B[i] := PASSED
49  Flush(&B[i])
50  last[n_i] := i                                       // Set p_i as the last accessing process
51  Flush(&last[n_i])
```

Algorithm 4.3: Filter algorithm for process p_i.

The algorithm consists of three main operations: filtering, filter handoff, and filter recovery. In this section, each operation is discussed in detail.

4.3.1 Filtering Operation

This operation, shown as **Filter** in Algorithm 4.3, provides the main benefit of using the RF lock, NUMA-awareness. When p_i executes **Filter**, it reads the ID of the owner node n from the shared variable *owner* (Line 36). Then, p_i checks if $n_i = n$ (i.e., the owner node) at Line 37. If n_i is the owner node, p_i proceeds to access the base lock. Otherwise, p_i sets $B[i]$ to its node n_i (Line 39), making it *visible* to other processes that p_i wants to access the base lock and that its corresponding node is n_i.

Given an owner node n_k, a process p_k sets the flag *alone* to 1 only when p_k attempts

Procedure FindNextOwner(k)

```
52 h := k
53 repeat
54 |   h := (h  mod N) + 1,
55 |   tmp := B[h]
56 |   Flush(&B[h])
57 until h = k or tmp ∉ {⊥, PASSED, HANDOFF, n_i}
58 if h ≠ k then                                   // if n_k is not alone
59 |   return ⟨h, tmp, 0⟩                          // h ≠ k, tmp = n_h
60 else
61 |   return ⟨h, n_i, 1⟩                          // h = k, n_i = n_k
```

Algorithm 4.4: A helper function used in **FilterHandoff** for process p_i.

to hand over the filter ownership to another node in **FilterHandoff** but finds no process that wants to pass the filter (more details in section 4.3.2). When the *alone* flag is set by p_k, processes on n_k consecutively hand over the base lock between them until a process p_i, where $n_i \neq n_k$, claims the filter ownership in **Filter** as part of the filtering operation. The first process that claims the ownership becomes the owner.

The process of claiming filter ownership in **Filter** follows the *test-and-test-and-set* approach. After p_i updates its *knowledge* of the owner node, the owner process, and the *alone* flag by reading *owner* at Line 41, it performs a *test* at Line 42 to verify whether n_i is not the owner node and the current owner node is alone. If this condition is satisfied, p_i proceeds with a *test-and-set* operation — specifically a CAS at Line 43 — to attempt to establish itself as the owner process, designate its node as the owner node, and reset the *alone* flag to 0. Once a process p_i performs a successful CAS, and n_i becomes the owner node, no process can claim the ownership of the filter until a process p_j, where $n_j = n_i$ (possibly $j = i$), sets *alone* back to 1 (because the tests at Lines 42 and 43 fail as long as $alone = 0$). A final read of *owner* is performed at Line 44 by all processes so that, at that point, all processes agree on one owner node and, accordingly, pass the filter and set their corresponding variable in the array B to PASSED (Line 48) or spin at Line 46.

If a process crashed after passing the filter (Line 48), it will skip the filtering operation in all the following passages during the current super-passage. That is, upon resurrection, if p_i reads $B[i] =$ PASSED at Line 35, then p_i skips Lines 36-48 and access the base lock. At the end of the filtering operation, p_i races other processes on the same node to change

```
Procedure FilterHandoff(⟨k, n, alone⟩ := ⟨⊥, ⊥, 0⟩)          // optional parameters
    // if reached threshold or p_i is last process
62  if counter[n_i] ≥ HandoffThreshold or last[n_i] = i then
63      ⟨k, n, alone⟩ := owner
64      Flush(&owner)
65      if not alone and n_i = n then                  // check if n_i is owner node and not alone
66          Flush(&last[n_i])
67          B[i] := Handoff                                  // mark start of handoff
68          Flush(&B[i])
69          ⟨k, n, alone⟩ := FindNextOwner(k)          // determine the new value of owner
70          owner := ⟨k, n, alone⟩
71          Flush(&owner)
72          if alone then                                                // if n_i is alone
73              ⟨k_2, n_2, alone_2⟩ := FindNextOwner(k)          // perform one more search
74              if not alone_2 then
75                  CAS(&owner, ⟨k, n, alone⟩, ⟨k_2, n_2, alone_2⟩)      // try to pass ownership
76                  Flush(&owner)
77                  ⟨_, n, _⟩ := owner                                          // update n
78          for h ∈ 1...N where B[h] = n do
79              B[h] := Passed                        // set B[h] to Passed for all p_h on n_k
80              Flush(&B[h])
81          counter[n_i] := 0                                    // reset n_i's counter
82          Flush(&counter[n_i])
83          B[i] := Passed                                       // mark end of handoff
84          Flush(&B[i])
85  counter[n_i] := counter[n_i] + 1                             // advance counter
86  Flush(&counter[n_i])
```

Algorithm 4.5: FilterHandoff algorithm for process p_i.

$last[n_i]$ to its ID number at Line 50, marking itself as the last process to leave the operation and access the base lock.

Procedure MarkExit

87 $B[i] := \perp$ // mark end of super-passage

88 **Flush**$(\&B[i])$

Algorithm 4.6: MarkExit algorithm for process p_i.

4.3.2 Filter Handoff Operation

The handoff operation begins when p_i changes $B[i]$ from PASSED to HANDOFF at Line 67, and concludes when p_i resets $B[i]$ to PASSED at Line 83. Thus, process p_i is considered to be performing a handoff operation if and only if $B[i] =$ HANDOFF. Even in the event of a crash during the handoff (i.e., while $B[i] =$ HANDOFF), p_i is still regarded as engaged in the handoff until it successfully resets $B[i]$ to PASSED.

In this operation, which is part of the **FilterHandoff** function (Algorithm 4.5), the ownership of the filter is handed over between different nodes to establish long-term fairness. **FilterHandoff** takes one optional argument, a tuple $\langle k, n, alone \rangle$, where k is the owner process ID, $n = n_k$ is the owner node ID and *alone* is the alone flag. Simply, the tuple $\langle k, n, alone \rangle$ is a local copy of *owner* that is determined at the time **FilterHandoff** is called. The default value for the optional argument is $\langle \perp, \perp, 0 \rangle$. The procedure can be invoked under normal execution at Line 23 or upon recovery from a crash failure at Lines 30, 32, and 34. More details are provided in Section 4.3.3.

Process p_i initiates the handoff operation only if both of the following conditions are satisfied:

1. The node counter reaches HANDOFFTHRESHOLD, or p_i reads $last[n_i] = i$ (Line 62).
2. Node n_i is the current owner and is not marked as alone (Line 65).

If p_i calls **FilterHandoff** and 1 or 2 are not satisfied, then p_i skips the entire operation (Lines 65-84), and calling the procedure is equivalent to incrementing the local counter at Line 85. Note that checking $last[n_i] = i$ to confirm that p_i is the last process on node n_i accessing the base lock is a conservative approach. The algorithm is designed so that if p_i is not alone, is the last process on n_i accessing the base lock, and n_i is the owner node, then p_i is guaranteed to read $last[n_i] = i$ at Line 65. However, the reverse implication does not hold. Such a conservative method might affect performance but prevents deadlock. Refer to Section 4.4 for detailed algorithm correctness.

Once the handoff operation begins, p_i searches for the next owner process (and owner node) by invoking the helper function **FindNextOwner** at Line 69. This function takes the current owner's process ID as an argument and scans $B[1...N]$ in a circular manner, starting from $k+1$ and wrapping around to k, to locate the first process p_h where $n_h \neq n_k$ and $B[h] = n_h$ (indicating p_h is attempting to pass the filter). If such a process p_h is found, **FindNextOwner** returns $\langle h, n_h, 0\rangle$. If no such process is found, it returns $\langle k, n_i, 1\rangle$, indicating that the owner node $n_i = n_k$ remains alone.

If p_i finds a new owner process p_h, then p_i hands over the filter ownership by setting *owner* to $\langle h, n_h, 0\rangle$, at Line 70. Then, p_i sets $B[j], \forall j$ where $n_j = n_h$, to PASSED (Line 78), signaling all p_j to break out of their spin-loop (Line 46) and access the base lock. On the other hand, if p_i returns $\langle k, n_k, 1\rangle$ from its call to **FindNextOwner** at Line 69 (i.e., does not find a new owner process), then p_i sets *owner* (from $\langle k, n_k, 0\rangle$) to $\langle k, n_k, 1\rangle$, making it visible to other processes that $n_k = n_i$ is alone. That is, if any process p_j, where $n_j \neq n_k$, performs the filtering operation, p_j may claim the filter ownership (as discussed in Section 4.3.1).

As mentioned earlier, p_i will never start a handoff operation if n_k is alone. Now, consider the following problem that involves two processes, p_i and p_j: process p_i starts a handoff operation and returns $\langle k, n_k, 1\rangle$ from **FindNextOwner** at Line 69. Then, p_j performs a filtering operation, where it sets $B[j]$ to n_j, reads that n_k is not alone (since p_i did not update *owner* yet) and spins on $B[j]$. Afterward, p_i updates *owner*, such that the alone flag is set to 1. If p_i ended the handoff operation at this point, then p_i would never execute another handoff operation, thus, leaving p_j to spin/starve forever. Lines 72-77 solve this issue. After p_i returns $\langle k, n_k, 1\rangle$ from **FindNextOwner** at Line 69, it performs one more search (by calling **FindNextOwner**(,) at Line 73) and then updates *owner* to a new owner process, if found. A CAS operation is needed to update *owner* at Line 75 since p_i set the alone flag to 1 before starting the second search and, hence, the filter ownership could be already claimed by another process in **Filter**. Similarly to Line 44, in **Filter**, p_i reads *owner* (Line 77) after performing the CAS (even if p_i failed the CAS) to agree on the same value of *owner*. Then, p_i proceeds to set all the spin variables that correspond to the owner node to PASSED, as discussed earlier. Finally, in all cases, p_i resets the node-local counter (Line 81) before it completes the handoff operation (setting $B[i]$ to PASSED, at Line 83).

4.3.3 Filter Recovery Operation

This operation, Lines 27-34 in **FilterRecover** (Algorithm 4.2), represents the recovery protocol of the RF lock's outer filter. This recovery process is initiated by p_i only if it

crashed while performing a handoff operation. If no crash occurred during the handoff, the filter remains in a valid state, and recovery is unnecessary. The filter recovery simply adopts the principle of "continuing where you left off," completing any remaining steps of the interrupted handoff operation during which p_i crashed.

The handoff operation is executed before releasing the base lock (Line 24), meaning it occurs within the CS of the base lock. When p_i initiates the filter recovery operation, it first reads the *owner* variable at Line 27. Next, there are three scenarios based on which p_i determines the necessary steps to restore the filter's state:

1. If n_k is alone, then p_i had set *owner* to $\langle k, n_k, 1\rangle$ before the crash. Therefore, p_i resumes the handoff operation starting from Line 73.
2. If $n_k = n_i$, then p_i did not modify *owner* before crashing. In this case, p_i restarts the handoff operation from the beginning (Line 69).
3. If neither 1 nor 2 applies, then *owner* must be $\langle k, n_k, 0\rangle$, where $n_k \neq n_i$. This indicates that p_i successfully changed the owner process before crashing. Thus, p_i completes the handoff by setting $B[j] = \textsc{Passed}$ for all j where $n_j = n_k$ (Line 78).

It is irrelevant whether p_i crashes before, during, or after updating the spin variables to PASSED, as each spin variable can only be set to PASSED once, during a single handoff operation, regardless of how many times the loop at Line 78 is executed. Although re-executing the loop may degrade performance, it does not impact the correctness of the algorithm; the same holds true for resetting the node-local counter at Line 81.

4.4 Correctness

In this section, I prove that the RF lock satisfies the mutual exclusion (ME), starvation-freedom (SF), and critical section re-entry (CSR) correctness properties, as defined in Section 2.5.3. Additionally, I analyze the remote memory reference (RMR) complexity of the RF lock according to the model presented in Section 2.5.2.

Assumption 4.1. The base RME lock satisfies ME, SF and CSR.

Definition 4.1. A process p_i is said to be *spinning at Line 46* if and only if p_i is at Line 46 and $B[i] = n_i$.

Definition 4.2. Given a history H, a *complete execution of the function* **FindNextOwner** *by* p_i is a sub-history of H that comprises a minimal sequence of consecutive steps, such that, p_i executes Line 52 at the first step, and executes Line 59 or 61 at the last step of the sub-history.

Definition 4.3. Given a history H, let H' be a complete execution of the function **FindNextOwner** by p_i. Let k be the value read by p_i at Line 52. *The* p_i*'s search list* $L_i[1 \dots N] = (k+1, k+2, \dots, N, 1, \dots, k-1, k)$ is a sequence of all process IDs such that, for $m \in \{1 \dots N\}$:

$$L_i[m] = \begin{cases} k+m & \text{if } k+m \leq N \\ k+m-N & \text{otherwise} \end{cases}$$

Definition 4.4. For any valid history H, a *hand-off period* of p_i, denoted H_i^h, is a sub-history of H that comprises a maximal sequence of consecutive steps $(s_1, s_2, \dots)$, such that $B[i] =$ HANDOFF immediately after s_k for each $k \geq 1$, and either H_i^h is infinite, or H_i^h is finite and $B[i] \neq$ HANDOFF immediately after $s_{|H_i^h|}$.

According to Algorithms 4.2-4.6, given a hand-off period $H_i^h = (s_1, s_2, \dots)$, process p_i sets $B[i]$ to HANDOFF at Line 67 at Step s_1. If H_i^h is finite, p_i sets $B[i]$ to PASSED at Line 83 at Step $s_{|H_i^h|}$. Note that hand-off periods are pairwise disjoint. That is, if a process p_i started a hand-off period, H_i^h, no other process can start a hand-off period until after p_i ends H_i^h by setting $B[i]$ to PASSED at Line 83. Please refer to Lemma 4.10 for more details.

Definition 4.5. Given a valid history H, let T_m be the step in H such that T_0 is an auxiliary step that represents system initialization, preceding all other steps in H, and that, for $m > 0$, the value of *owner* is changed for the m-th time at T_m. An *ownership period*, denoted as H_m^o, is the sub-history in H starting with T_m and ending just before T_{m+1}, if such a step exists.

Given an ownership period H_m^o, for some $m \geq 0$, if T_{m+1} does not exist, then either H_m^o is infinite, or H_m^o is finite and it is the last ownership period in H, where the last step in H is also the last step in H_m^o. I refer to the last ownership period in H as the *terminal* one, and all other ownership periods as *non-terminal*. From Algorithms 4.2-4.6, the initial value of *owner* is $\langle \perp, \perp \rangle$ immediately after T_0. For $m > 0$, T_m is defined as the step in which a successful CAS operation occurs at either Line 43 or Line 75, or when a direct

assignment happens at Line 70. Given any history H, the concatenation of all ownership periods forms H, such that:

$$H = H_0^o \circ H_1^o \circ H_2^o \circ \cdots$$

Lemma 4.1. *A process p_i executes the repeat-until-loop in Lines 53-57 in $O(N)$ of p_i's own steps.*

Proof. The function **FindNextOwner**(k) is called at Lines 69 and 73 in the function **FilterHandoff**, where k is passed to **FindNextOwner**. In **FilterHandoff**, k is initialized either at Line 63 by reading the variable *owner*, so that $k = owner_1$, or as a parameter passed to the function **FilterHandoff**. The function **FilterHandoff** is called, with optional parameters passed, in Lines 30, 32 and 34. In all three cases, the passed k is initialized as $owner_1$ at Line 27. Hence, k stores a value previously read from $owner_1$ at Line 27 or 70.

The value of $owner_1$ is set at Lines 43, 70 and 75. At Line 43 $owner_1$ is set to j, for some process ID $j \in 1 \ldots N$. At Lines 70 and 75, $owner_1$ is changed based on some value v returned from an invocation of the function **FindNextOwner** at Lines 69 and 73, respectively. Then, $owner_1$ is set to $v.k$. The value $v.k$ is the value of the private variable h returned at Line 59 or 61. According to the repeat-until-loop in **FindNextOwner**, the returned value of h is determined by executing Line 54. It follows that $(h \mod N) \in \{0 \ldots N-1\}$. As a result, $k = owner_1 = v.k = h = (h \mod N) + 1$ and, therefore, $k \in \{1 \ldots N\}$.

In **FindNextOwner**, process p_i initially sets h to $k \in 1...N$ at Line 52. Then, p_i increments the value of h in a circular fashion at Line 54 in every iteration. Consequently, it is guaranteed that $h = k$ after $< N$ iterations and, thus, the loop is terminated in $O(N)$ of p_i's own steps. □

Lemma 4.2. *For any history H, the base lock sections are executed in the correct order.*

Proof. From Algorithm 4.1, a process executing the algorithm executes the recovery, entry and then exit sections of the RF lock, in that order. In the RF lock's entry section, the base lock's recovery section is executed at Line 21 followed by the base lock's entry section at Line 22. Subsequently, the exit section of the RF lock is executed after the CS of the RF lock. In the exit section of the RF lock, the base lock's exit section at Line 24 is executed, which is the correct execution order of the base lock's sections. □

Lemma 4.3. *For any fair history H, if a process crashes outside the NCS of the base lock, it eventually starts executing the recovery section of the base lock, or crashes infinitely often in the NCS.*

Proof. If a process p_i crashes outside the NCS of the base lock, then p_i crashes in the recovery, entry, critical or exit section of the base lock. By Lemma 4.2, the recovery section of the base lock is the first to be executed by p_i outside the NCS of the base lock. From Algorithm 4.1, process p_i enters the recovery section of the base lock (Line 21) only if it calls the function **Filter**, in which p_i either reads $B[i] = \text{PASSED}$ or $B[i] \neq \text{PASSED}$ at Line 35. If $B[i] = \text{PASSED}$, process p_i skips to Line 50 and then enters the base lock's recovery section. If $B[i] \neq \text{PASSED}$, p_i executes Lines 36-46, setting $B[i]$ to PASSED (Line 46), and finally entering the recovery section of the base lock. In both cases, if process p_i gets outside the NCS (enters the recovery section) of the base lock, then $B[i] = \text{PASSED}$ and flushed at Line 49. From Algorithms 4.2-4.6, if p_i is outside the NCS, the value of $B[i]$ can be set by p_i only to PASSED at Line 83 or to HANDOFF at Line 67, and by a process p_j, for $j \neq i$, only to PASSED at Line 79. Each of the three values are flushed in the following step at Lines 84, 68 and 80, respectively. Therefore, when p_i restarts after it crashes outside the NCS, there are two cases:

CASE A: $B[i] = \text{PASSED}$.

> p_i skips Lines 27-34 of **FilterRecover** and Lines 36-46 of **Filter** because of Lines 26 and 35, respectively. Therefore, either p_i returns from **Filter** and starts executing the base lock's **Recover**, or p_i crashes in the NCS of the base lock.

CASE B: $B[i] = \text{HANDOFF}$.

> By Line 26, p_i executes either Line 30, 32 or 34. The three lines involve setting $B[i]$ to PASSED at Line 83. The only loop in **FilterHandoff** is in **FindNextOwner**, which is bounded (Lemma 4.1). Therefore, p_i returns from **FilterHandoff** and, since $B[i]$ is set to PASSED at Line 83, p_i skips Lines 36-46 of **Filter** (Line 35). Subsequently, either p_i returns from **Filter** and starts executing the base lock's **Recover**, or p_i crashes in the NCS.

In both cases, p_i starts executing the base lock's **Recover**, or p_i crashes in the NCS. If p_i crashes in the NCS, then p_i starts executing the NCS and $B[i] = \text{PASSED}$ or $B[i] = \text{HANDOFF}$, and p_i follows CASE A or B, respectively. Eventually, p_i starts executing the base lock's **Recover**, or crashes in the NCS of the base lock infinitely many times. □

Lemma 4.4. *The base lock is accessed correctly.*

Proof. A process accessing the base lock executes the lock's sections in the correct order by Lemmas 4.2. Moreover, given a fair history of the RF lock, if the process crashes outside the NCS of the base lock, it eventually executes the recovery section of the base lock, or crashes infinitely many times (Lemma 4.3). □

Lemma 4.5. *Given any finite history H, the RF lock satisfies ME.*

Proof. Suppose, for the sake of contradiction, that at the end of a finite history H of an RF lock, there exist processes p_i and p_j, for $i \neq j$, that are both in the CS. In Algorithm 4.1, the CS of the RF lock is inside the CS of the base lock. Therefore, if p_i and p_j are in the CS of the RF lock, then both are also in the CS of the base lock, which contradicts that the base lock satisfies ME (Lemma 4.4). □

Lemma 4.6. *Given any history H, the RF lock satisfies CSR.*

Proof. Assume for contradiction, that in a history H of an RF lock, there exist p_i and p_j, for $i \neq j$, such that p_i crashes inside the CS of the RF lock and p_j enters the CS of the RF lock before p_i re-enters the CS of the RF lock, after the crash failure under consideration. In Algorithm 4.1, the CS of the RF lock is inside the CS of the base lock. Therefore, p_i crashes inside the CS of the RF lock implies that it also crashes inside the CS of the base lock. If p_j enters the CS of the RF lock before p_i re-enters the CS of the RF lock, then p_j enters the CS of the base lock before p_i re-enters the CS of the base lock, after the crash failure under consideration, contradicting that the base lock satisfies CSR (Lemma 4.4). □

Lemma 4.7. *For any history H, if p_i is at Lines 27-34, then p_i is in the CS of the base lock.*

Proof. Process p_i executes Lines 27-34 in **FilterRecover** only if p_i reads $B[i] =$ HANDOFF at Line 26. In Algorithms 4.2-4.6, $B[i]$ can be set to HANDOFF only at Line 67 by p_i at a prior step, which occurs while p_i is in the CS of the base lock. In the absence of failure, p_i should have executed the exit section of the base lock (Line 24) after setting $B[i]$ to HANDOFF at Line 67, before invoking **FilterRecover** (Algorithm 4.1). However, this would imply that p_i would have set $B[i]$ to PASSED at Line 83 before returning from the **FilterHandoff** function and before executing the exit section. This contradicts the fact that p_i reads $B[i] =$ HANDOFF at Line 26 in **FilterRecover**. Therefore, after setting $B[i]$ to HANDOFF at Line 67, p_i must have crashed before reaching Line 83 and before starting the execution of the exit section of the base lock. Consequently, p_i being at Lines 27-34 implies that it has already executed the entry section of the base lock but has not yet executed its exit section. By the definition of the critical section, this means that p_i is currently in the CS of the base lock. □

Corollary 4.8. *If p_i is executing the function* **FilterHandoff**, *then p_i is in the CS of the base lock.*

Proof. The function **FilterHandoff** can be invoked by process p_i at Line 23 in the CS of the base lock (Algorithm 4.1), or at Lines 30, 32, or 34, all of which are also in the CS of the base lock by Lemma 4.7. □

Lemma 4.9. *Given a valid fair history H, and a hand-off period H_i^h of some process p_i, H_i^h is finite, or there are infinitely many crash steps in H.*

Proof. Suppose for contradiction that H_i^h is infinite and that there are finite crash steps in H. Hence, no process can crash an infinite number of times in H. By Definition 4.4, H_i^h starts when $B[i]$ is set to HANDOFF. In Algorithms 4.2-4.6, $B[i]$ is set to HANDOFF only at Line 67 by p_i in the function **FilterHandoff**. There are two loops in **FilterHandoff**: the repeat-until loop at Lines 53-57, which is reached by invoking the **FindNextOwner** function, and the for-loop at Lines 78-80. Both loops are bounded by $O(n)$ of p_i's own steps, as shown in Lemma 4.1 for the repeat-until loop and at Line 78 for the for-loop. Then, the function **FilterHandoff** is also bounded by $O(n)$ of p_i's own steps. Therefore, either p_i sets $B[i]$ to PASSED at Line 83, which proves that H_i^h is finite, or p_i crashes. If p_i crashes, it restarts and either (A) does not crash again or (B) crashes.

CASE A: p_i does not crash.

> First, process p_i executes **FilterRecover**. Since $B[i] =$ HANDOFF (by the contradiction assumption), p_i calls **FilterHandoff** either at Line 30, 32 or 34. In all cases, p_i sets $B[i]$ to PASSED at Line 83, concluding H_i^h.

CASE B: p_i crashes.

> Process p_i restarts after it crashes again. Subsequently, either p_i does not crash again, in which Case A applies, or p_i crashes again, in which Case B applies.

Eventually, p_i sets $B[i]$ to PASSED at Line 83 as explained in Case A, concluding H_i^h, or p_i crashes infinitely many times in H, which is a contradiction. □

Lemma 4.10. *Given a valid fair history H and a hand-off period $H_i^h = (s_1, s_2, \ldots)$ of some process p_i, $\forall k \in 1 \ldots |H_i^h|$ and $\forall j \neq i$, p_j does not set $B[j]$ to* HANDOFF *at s_k.*

Proof. In Algorithms 4.2-4.6, $B[j]$ is set to HANDOFF only if p_j executes Line 67. Assume for contradiction that p_j sets $B[j]$ to HANDOFF at some step s_k at Line 67. By Definition 4.4, $B[i]$ is set to HANDOFF at step s_1 and to a different value at step $s_{|H_i^h|}$. Therefore, p_j does not set $B[j]$ to HANDOFF at s_1 or $s_{|H_i^h|}$.

For $1 < k < |H_i^h|$: Line 67 can only be reached by invoking **FilterHandoff** at Line 23, which is in the CS of the base lock. Therefore, p_j is in the CS of the base lock at s_k.

There are two possibilities regarding p_i: either p_i (A) does not crash after s_1 and before s_k, or (B) crashes after s_1 and before s_k.

CASE A: p_i does not crash after s_1 and before s_k.

If p_i does not crash after s_1 and before s_k, then p_i is at Lines 68-83. As a result, p_i is in the CS of the base lock immediately before s_k, along with p_j, which contradicts the ME property of the base lock (Lemma 4.4).

CASE B: p_i crashes after s_1 and before s_k.

If p_i crashes after s_1 and before s_k, then p_i enters the recovery section (Lemma 4.3) and eventually enters the CS of the base lock by the SF property (Lemma 4.4 and Assumption 1). Accordingly, p_j sets $B[j]$ to HANDOFF at s_k either (B1) before p_i re-enters the CS of the base lock or (B2) after p_i re-enters the CS of the base lock.

CASE B1: s_k occurs before p_i re-enters the CS of the base lock.

Process p_j sets $B[j]$ to HANDOFF before p_i re-enters the CS of the base lock, which implies that p_j enters the CS of the base lock before p_i re-enters the CS of the base lock, which contradicts the CSR property of the base lock (Lemma 4.4).

CASE B2: s_k occurs after p_i re-enters the CS of the base lock.

After the crash failure, p_i calls **FilterRecover**, reads $B[i] =$ HANDOFF (by the definition of H_i^h). Then, by Line 26, p_i executes either Line 30, 32 or 34 before re-entering the CS of the base lock. The three lines involve setting $B[i]$ to PASSED at Line 83, ending H_i^h before p_i enters the CS of the base lock, which contradicts that p_j sets $B[j]$ to HANDOFF at some s_k for $1 < k < |H_i^h|$.

□

Lemma 4.11. *Given a valid fair history H, if $B[i] =$ HANDOFF, then $\forall j \neq i, B[j] \neq$ HANDOFF.*

Proof. Suppose for the sake of contradiction that $B[i] = B[j] =$ HANDOFF. In Algorithms 4.2-4.6, $B[i]$ is set to HANDOFF only if p_i executes Line 67. If p_i sets $B[i]$ to HANDOFF before p_j sets $B[j]$ to HANDOFF, then p_i starts a hand-off period H_i^h, in which p_j sets $B[j]$ to HANDOFF at Step s_k for some $k \in \{1 \dots |H_i^h|\}$, which contradicts Lemma 4.10. Analogous argument if p_j sets $B[j]$ to HANDOFF before p_i sets $B[i]$ to HANDOFF. □

Lemma 4.12. *Given a valid fair history H, if p_i is at Lines 69-83, then $B[i] =$ HANDOFF.*

Proof. If p_i is at Lines 69-83, then p_i must have called **FilterHandoff** (i) at Line 23 in the RF lock's **Exit** or (ii) at Line 30, 32, or 34 in **FilterRecover**. If (i), then p_i has set $B[i]$ to HANDOFF at Line 67, starting a hand-off period H_i^h. If (ii), then p_i has read $B[i] =$ HANDOFF at Line 26, which implies that p_i has started a hand-off period H_i^h in a prior step. In both cases, p_i does not change $B[i]$ until it executes Line 83, ending H_i^h. In addition, no p_j (for $j \neq i$) changes $B[i]$ before p_i executes Line 83 because p_j can change $B[i]$ only at Line 79. Similar to p_i, if p_i is at Lines 69-83, then p_j has started a hand-off period H_j^h, either before or after p_i started H_i^h. Both cases contradict Lemma 4.10. Thus, it is guaranteed that $B[i] =$ HANDOFF while p_i is executing Lines 69-83. □

Lemma 4.13. *For any history H, if p_i is at Lines 40-48, then $B[i] \in \{n_i, \text{PASSED}\}$.*

Proof. Assume for the sake of contradiction that p_i is at Lines 40-48, and $B[i] \notin \{n_i,$ PASSED$\}$. The value of $B[i] \in \{\text{PASSED}, n_i, \text{HANDOFF}, \perp\}$. Since p_i is at Lines 40-48, then p_i sets $B[i]$ to n_i at Line 39 at a prior step. Subsequently, $B[i]$ is changed to some value $\in \{\text{HANDOFF}, \perp\}$ after p_i sets it to n_i. In Algorithms 4.2-4.6, $B[i]$ can be set to HANDOFF at Line 67 and to $\perp$ at Line 87 only by p_i itself. Therefore, setting $B[i]$ to HANDOFF or $\perp$ implies that p_i is not at Lines 40-48, which is a contradiction. □

Lemma 4.14. *Given a valid fair history H, let H' be a subsequence of H such that H' is a complete execution of the function **FindNextOwner**(k) by p_i. Then, p_i either sets k to $owner_1$ or sets $owner_1$ to k at some step $s \in H$, before H', and no process changes owner until the end of H'.*

Proof. If p_i is executing **FindNextOwner**(k), then p_i invoked the function at Line 69 or 73 in **FilterHandoff**. Function **FilterHandoff** is invoked by p_i either in (A) **Exit** at Line 23 or (B) **FilterRecover** at Lines 30, 32, or 34.

CASE A: **FilterHandoff** is invoked in **Exit**.

Since p_i invoked **FilterHandoff** in **Exit**, then p_i is in the CS of the base lock (Algorithm 4.1). Here I have two cases: (A1) **FindNextOwner**(k) is invoked at Line 69 or (A2) Line 73.

CASE A1: **FindNextOwner**(k) is invoked at Line 69.

Process p_i has initialized the variable k with the value of $owner_1$ at Line 63. Now, I show that the value of *owner* is guaranteed not to change until the invocation, and during the execution, of **FindNextOwner**(k) under consideration. Assume for contradiction that the value of *owner* is changed before the end of H' and after p_i reads the value of *owner* at Line 63. The variable *owner* can be set through

a direct assignment at Line 70, or through a CAS at Lines 75 or 43. Since p_i is executing Lines 65-69, p_i reads $owner_3 = 0$ (due to Line 65), and *owner* must have been changed by another process p_j. As $owner_3 = 0$, process p_j could not possibly change *owner* through the CAS at Line 43 in **Filter** due to the CAS condition ($owner_3 = 1$). Therefore, p_j must have changed *owner* either through a direct assignment at Line 70, or through the CAS operation at Line 75. Both Lines imply that p_j is in the function **FilterHandoff** which is in the CS of the base lock (Corollary 4.8), contradicting that p_i is the CS of the base lock (Lemma 4.4).

CASE A2: **FindNextOwner**(k) is invoked at Line 73.

Process p_i has set the variable $owner_1$ with the value of k at Line 70 before calling the function **FindNextOwner**(k). Now, I show that the value of *owner* is guaranteed not to change until the invocation, and during the execution, of **FindNextOwner**(k) under consideration. Assume for contradiction that the value of *owner* is changed before the end of H' and after p_i executes line 70. The variable *owner* can be set through a direct assignment at Line 70, or through a CAS operation at Lines 75 or 43. Since p_i is executing Lines 71-73, then $owner_3 = 0$ (due to Line 72), and *owner* must have been changed by another process p_j. As $owner_3 = 0$, process p_j could not possibly change *owner* through the CAS at Line 43 in **Filter** due to the CAS condition ($owner_3 = 1$). Therefore, p_j must have changed *owner* either through a direct assignment at Line 70, or through the CAS operation at Line 75. Both Lines imply that p_j is in the function **FilterHandoff** which is in the CS of the base lock (Corollary 4.8), contradicting that p_i is the CS of the base lock (Lemma 4.4).

CASE B: **FilterHandoff** is invoked in **FilterRecover**.

Then, according to Corollary 4.7, process p_i is in the CS of the base lock during H'. Again, I have two cases: (B1) **FindNextOwner**(k) is invoked at Line 69 or (B2) 73.

CASE B1: **FindNextOwner**(k) is invoked at Line 69.

Process p_i has initialized the variable k with the value of $owner_1$ at Line 27 in **FilterRecover** before calling **FilterHandoff** at Line 32. Now, I show that the value of *owner* is guaranteed not to change until the invocation, and during the execution, of **FindNextOwner**(k) under consideration. Assume for contradiction that the value of *owner* is changed before the end of H' and after p_i reads the value of *owner* at Line 27. The variable *owner* can be set through a direct assignment at Line 70, or through a CAS at Lines 75 or 43. Since p_i is executing Lines 29-32 and 69, p_i reads $owner_3 = 0$ (due to Line 29), and *owner* must have

been changed by another process p_j. As $owner_3 = 0$, process p_j could not possibly change *owner* through the CAS at Line 43 in **Filter** due to the CAS condition ($owner_3 = 1$). Therefore, p_j must have changed *owner* either through a direct assignment at Line 70, or through the CAS operation at Line 75. Both Lines imply that p_j is in the function **FilterHandoff** which is in the CS of the base lock (Corollary 4.8), contradicting that p_i is the CS of the base lock (Lemma 4.4).

CASE B2: **FindNextOwner**(k) is invoked at Line 73.

Process p_i has initialized the variable k with the value of $owner_1$ at Line 27 in **FilterRecover** before calling **FilterHandoff** at Line 30. Now, I show that the value of *owner* is guaranteed not to change until the invocation, and during the execution, of **FindNextOwner**(k) under consideration. Assume for contradiction that the value of *owner* is changed before the end of H' and after p_i executes line 70. The variable *owner* can be set through a direct assignment at Line 70, or through a CAS operation at Lines 75 or 43. Since p_i is executing Lines 71-69, then $owner_3 = 0$ (due to Line 72), and *owner* must have been changed by another process p_j. As $owner_3 = 0$, process p_j could not possibly change *owner* through the CAS at Line 43 in **Filter** due to the CAS condition ($owner_3 = 1$). Therefore, p_j must have changed *owner* either through a direct assignment at Line 70, or through the CAS operation at Line 75. Both Lines imply that p_j is in the function **FilterHandoff** which is in the CS of the base lock (Corollary 4.8), contradicting that p_i is the CS of the base lock (Lemma 4.4).

□

Lemma 4.15. *Given a valid fair history H, and a complete execution of the function **FindNextOwner**(k) by p_i. If p_i returns $\langle h, tmp, 0\rangle$ at Line 59, then $h \neq k$ and $tmp = n_h \neq n_i$.*

Proof. The proof is straightforward from the conditions of the if-statement and loop at Lines 58 and 57, respectively. By Line 58, $h \neq k$. Since the loop has terminated, then based on Line 57, $tmp \notin \{\perp, \textsc{Passed}, \textsc{Handoff}, n_i\}$. From Line 55, $tmp = B[h]$. In Algorithms 4.2-4.6, due to Lines 39, 48, 87, 67, 79 and 83, $B[h] \in \{\perp, \textsc{Passed}, \textsc{Handoff}, n_h\}$. Then, $tmp = n_h \neq n_i$. □

Lemma 4.16. *Given a valid fair history H, and a complete execution of the function **FindNextOwner**(k) by p_i. If p_i returns $\langle h, n_i, 1\rangle$ at Line 61, then $h = k$ and $owner_2 = n_i$.*

Proof. Directly from the if-condition at Line 58, process p_i returns $h = k$ at Line 61. Now I prove that p_i returns $n_i = owner_2$. I divide the proof into two claims based on the Line at which the function **FindNextOwner** is called.

Claim 4.1. Given a valid fair history H, and a complete execution of the function **FindNextOwner**(k) by p_i invoked at Line 69. If p_i returns $\langle h, n_i, 1\rangle$ at Line 61, then $owner_2 = n_i$.

Proof. (For Claim 4.1) Assume for contradiction p_i returns $\langle h, n_i, 1\rangle$ at Line 61 and $owner_2 \neq n_i$. However, process p_i reads $\langle k, n, alone\rangle = owner$ at Line 63 before calling **FindNextOwner**. By the if-statement at Line 65, $n = n_i$. By the contradiction assumption, $owner_2 \neq n_i = n$, which implies that $owner$ is changed by some process p_j possibly $(i = j)$ after p_i reads $owner$ at Line 63 and before p_i calls **FindNextOwner** at Lines 69, contradicting Lemma 4.14. □

Claim 4.2. Given a valid fair history H, and a complete execution of the function **FindNextOwner**(k) by p_i invoked at Line 73. If p_i returns $\langle h, n_i, 1\rangle$ at Line 61, then $owner_2 = n_i$.

Proof. (For Claim 4.2) Assume for contradiction p_i returns $\langle h, n_i, 1\rangle$ at Line 61 and $owner_2 \neq n_i$. However, process p_i sets $owner = \langle k, n, alone\rangle$ at Line 70 before calling **FindNextOwner** at Line 73. In Algorithms 4.2-4.6, the value $\langle k, n, alone\rangle$ is returned from the invocation of the function **FindNextOwner** at Line 69. By Claim 4.1, p_i returned $n = n_i$ from that invocation. By the contradiction assumption, $owner_2 \neq n_i = n$, which implies that $owner$ is changed by some process p_j (possibly $i = j$) after p_i sets $owner$ at Line 70 and before p_i calls **FindNextOwner** at Lines 73, contradicting Lemma 4.14. □

Therefore, if p_i returns $\langle h, n_i, 1\rangle$ at Line 61 from a complete execution of the function **FindNextOwner**(k) invoked at Line 69 (Claim 4.1) or Line 73 (Claim 4.2), then $h = k$ and $owner_2 = n_i$. □

Lemma 4.17. *Given a valid history H, $\forall m > 0$, $owner_1 = j$, for some $j \in \{1 \dots N\}$, and $owner_2 = n_j$ in the ownership period H^o_m.*

Proof. I proceed by induction on m.

Base Case $(m = 1)$: At the beginning of H^o_0, just after the auxiliary step T_0, the variable $owner = \langle \perp, \perp, 1\rangle$ and all elements of the array B are $\perp$. All active processes begin by executing the recovery section. The steps within this section are finite as processes bypass Lines 27-34, following the if-condition at Line 26. Consider a process p_i, which reads $B[i] \neq$ Passed (since $B[i] = \perp$ initially) at Line 35. Subsequently, p_i finds $owner_2 = \perp$ at Line 37 and $owner_3 = 1$ and again $owner_2 = \perp$ at Line 42, consistent with the initial state of $owner = \langle \perp, \perp, 1\rangle$. Therefore, p_i executes the CAS operation at Line 43. Assuming p_i

is the first process to perform this CAS operation, p_i changes *owner* to $\langle i, n_i, 0\rangle$. This step defines T_1, beginning H_1^o. Hence, $owner_1 = i \in \{1 \dots N\}$ and $owner_2 = n_i$.

Step $(m > 1)$ Assume that, $\forall m \in 1 \dots m'$ for some $m' \geq 1, owner_1 = j$, for some $j \in \{1 \dots N\}$, and $owner_2 = n_j$ in H_m^o. I shall prove that $owner_1 = j'$, where $j' \in \{1 \dots N\}$ (possibly $j = j'$), and $owner_2 = n_{j'}$ in $H_{m'+1}^o$. By Definition 4.5, Step $T_{m'+1}$ is either a successful CAS operation executed at (A) Line 43 or (B) Line 75, or (C) a direct assignment at Line 70. Let p_i be the process that changes *owner* at $T_{m'+1}$. In the subsequent part of this proof, I will demonstrate that $j' \in \{1 \dots N\}$ and $owner_2 = n_{j'}$ in all three scenarios.

CASE A: a successful CAS operation at Line 43.

Upon a successful CAS at Line 43, process p_i changes $owner_1$ to $i \in \{1 \dots N\}$ (since i is a process ID) and $owner_2 = n_i$.

CASE B: a successful CAS operation at Line 75.

At Line 75, *owner* is set to some value $\langle j', n, alone\rangle$ returned from an invocation of the function **FindNextOwner**(k) at Line 73.

The value $j' = h$ which is the private variable defined at Line 52. The value of h is determined by executing Line 54. Since $(h \mod N) \in \{0 \dots N-1\}$, then $j' \in \{1 \dots N\}$.

Now I show that $n = n_{j'}$. Since $T_{m'+1}$ is a a successful CAS operation at Line 75, then by the if-statement at Line 74 ($alone = 0$), process p_i returns $\langle h, tmp, 0\rangle$ at Line 59 where $n = tmp = B[h]$ (Line 55). Let k be the variable passed to the function **FindNextOwner**. According to the loop condition at Line 57, either $h = k$ or $tmp = B[h] \notin \{\perp, \textsc{Passed}, \textsc{Handoff}, n_i\}$. From the if-condition (Line 58), $h \neq k$. Therefore, $B[h] \notin \{\perp, \textsc{Passed}, \textsc{Handoff}, n_i\}$. In Algorithms 4.2-4.6, due to Lines 39, 48, 87, 67, 79 and 83, the value of $B[h] \in \{\perp, \textsc{Passed}, \textsc{Handoff}, n_h\}$. Hence, $n = B[h] = n_h = n_{j'}$ (since $j' = h$).

CASE C: a direct assignment at Line 70.

At Line 70, *owner* is set to some value $\langle j', n, alone\rangle$ returned from an invocation of the function **FindNextOwner**(k) at Line 69.

The value $j' = h$ which is the private variable defined at Line 52. The value of h is determined by executing Line 54. Since $(h \mod N) \in \{0 \dots N-1\}$, then $j' \in \{1 \dots N\}$.

Now I show that $n = n_{j'}$. Either p_i returns (1) at Line 59 or (2) at Line 61. If (1), then p_i returns $\langle h, tmp, 0\rangle$, where $n = tmp = B[h]$ (Line 55). Let k be the variable

passed to the function **FindNextOwner**. According to the loop condition at Line 57, either $h = k$ or $tmp = B[h] \notin \{\perp, \text{PASSED}, \text{HANDOFF}, n_i\}$. From the if-condition (Line 58), $h \neq k$. Therefore, $B[h] \notin \{\perp, \text{PASSED}, \text{HANDOFF}, n_i\}$. In Algorithms 4.2-4.6, due to Lines 39, 48, 87, 67, 79 and 83, the value of $B[h] \in \{\perp, \text{PASSED}, \text{HANDOFF}, n_h\}$. Hence, $n = tmp = B[h] = n_h = n_{j'}$ (since $j' = h$).

If (2), then p_i returns $\langle h, n_i, 1\rangle$ at Line 61 from **FindNextOwner**(k). From the if-condition at Line 58, $j' = h = k$. Since p_i called **FindNextOwner**(k) at Line 69, then p_i read $k = owner_1 = j$ at Line 63, and *owner* is unchanged until p_i returns from the invocation of **FindNextOwner**(k) under consideration (by Lemma 4.14). Therefore, $j = j'$. In addition, when p_i returns at Line 61 (before $T_{m'+1}$), $owner_2 = n_j$ by the induction assumption and $n(= n_i) = n_j$ by Lemma 4.16. Hence, $n = n_{j'}$ since $j = j'$.

□

Lemma 4.18. *Given a valid fair history H, let H' be a complete execution of the function* **FindNextOwner** *by p_i and m be the iteration number of the repeat-until-loop (Lines 53-57). If p_i at Line 55 in the m-th iteration, then $h = L_i[m]$ (Definition 4.3).*

Proof. By Lemma 4.1, $m \in \{1 \dots N\}$. Let h^m be the value of the variable h immediately before p_i executes Line 55 in the m-th iteration of the repeat-until-loop. I shall prove by induction on m that, $\forall m \in \{1 \dots N\}$, $h^m = L_i[m]$.
BASE CASE ($m = 1$): At the start of the first iteration, $h = k$ (Line 52). By Lemma 4.14, $k = owner_1$ in H', and by Lemma 4.17, $k = owner_1 \in \{1 \dots N\}$. Process p_i sets $h = h^1 = (k \bmod N) + 1$ at Line 54. Then, there are two cases: (A) $k + m \leq N$ or (B) $k + m > N$.

CASE A: $k + m \leq N$.

$$\begin{aligned} k + m &\leq N \\ k &\leq N - 1 < N && \because m = 1 \end{aligned}$$

Thus,

$$\begin{aligned} h^1 &= (k \bmod N) + 1 \\ &= k + 1 && \because k < N \\ &= L_i[1] \end{aligned}$$

CASE B: $k + m > N$.

$$\begin{aligned} k + m &> N \\ k &> N - 1 && \because m = 1 \\ k &\geq N \end{aligned}$$

Therefore,

$$\begin{aligned} h^1 &= (k \bmod N) + 1 \\ &= k - N + 1 && \because k \geq N \\ &= L_i[1] \end{aligned}$$

As a result, $h^1 = L_i[1]$.
STEP $(m > 1)$: Let $m_0 \in 1 \ldots N - 1$ be some iteration number. Suppose that, $\forall m \in \{1 \ldots m_0\}$, $h^m = L_i[m]$. I show that $h^{m_0+1} = L_i[m_0 + 1]$. In the $(m_0 + 1)$th iteration, p_i sets $h^{m_0+1} = (h^{m_0} \bmod N) + 1$ at Line 54. Then, I have two cases: (A) $k + m_0 + 1 \leq N$ and (B) $k + m_0 + 1 > N$.

CASE A: $k + m_0 + 1 \leq N$.

$$\begin{aligned} k + m_0 + 1 &\leq N \\ k + m_0 &\leq N - 1 < N \end{aligned}$$

Therefore,

$$\begin{aligned} h^{m_0+1} &= (h^{m_0} \bmod N) + 1 \\ &= (L_i[m_0] \bmod N) + 1 && \text{(Induction Assumption)} \\ &= ((k + m_0) \bmod N) + 1 && \because k + m_0 < N \\ &= (k + m_0) + 1 && \because k + m_0 < N \\ &= L_i[m_0 + 1] \end{aligned}$$

CASE B: $k + m_0 + 1 > N$.

$$\begin{aligned} k + m_0 + 1 &> N \\ k + m_0 &> N - 1 \\ &\geq N \end{aligned}$$

Accordingly, there are two sub-cases: (B1) $k + m_0 = N$ and (B2) $k + m_0 > N$.

CASE B1: $k + m_0 = N$.

$$\begin{aligned} h^{m_0+1} &= (h^{m_0} \bmod N) + 1 \\ &= (L_i[m_0] \bmod N) + 1 && \text{(Induction Assumption)} \\ &= ((k + m_0) \bmod N) + 1 && \because k + m_0 = N \\ &= (0) + 1 = 1 && \because k + m_0 = N \end{aligned}$$

And,

$$\begin{aligned} L_i[m_0 + 1] &= k + (m_0 + 1) - N && \because k + m_0 + 1 > N \\ &= N + 1 - N && \because k + m_0 = N \\ &= 1 = h^{m_0+1} \end{aligned}$$

CASE B2: $k + m_0 > N$.

Since $m_0 \leq N - 1$ and $k \leq N$, then:

$$\begin{aligned} k + m_0 - N &\leq N + (N - 1) - N \\ &\leq N - 1 \\ &< N \end{aligned}$$

Therefore,

$$\begin{aligned} h^{m_0+1} &= (h^{m_0} \bmod N) + 1 \\ &= (L_i[m_0] \bmod N) + 1 && \text{(Induction Assumption)} \\ &= ((k + m_0 - N) \bmod N) + 1 && \because k + m_0 > N \\ &= (k + m_0 - N) + 1 && \because k + m_0 - N < N \\ &= L_i[m_0 + 1] \end{aligned}$$

□

Lemma 4.19. *Given a valid history H, let H' be a complete execution of the function* **FindNextOwner** *by p_i. If p_i returns at Line 61, then $\forall j \in \{1 \dots N\}$, where $n_j \neq n_i$, p_i reads $B[j] \neq n_j$ at Line 55.*

Proof. Assume for the sake of contradiction that given H and H', process p_i returns at Line 61, and there exists a process p_j such that $n_j \neq n_i$ and p_i reads $B[j] = n_j$. Let $m \in \{1 \dots m_0\}$ be the iteration number of the repeat-until-loop (Lines 53-57), where $m_0 \in \{1 \dots N\}$ is the total number of iterations performed by p_i in H'. Let h^m be the

value of the private variable h (defined at Line 52) just before p_i executes Line 55 in the m-th iteration of the repeat-until-loop.

Since p_i returns at Line 61, p_i reads $h = k$ at Line 58, which implies that $h^{m_0} = k$. By Lemma 4.18, $h^{m_0} = L_i[m_0]$, and by Definition 4.3, $k = L_i[N]$. Since $h^{m_0} = k = L_i[N] = L_i[m_0]$ and the elements of L_i is distinct, then $m_0 = N$. Hence, either $j \in L_i[1 \dots m_0 - 1]$ or $j = L_i[m_0]$. If $j \in L_i[1 \dots m_0 - 1]$, then from Line 57, I know that $\forall m \in \{1 \dots m_0 - 1\}, p_i$ reads $h^m \neq k$ and $B[h^m] \in \{\perp, \textsc{Passed}, \textsc{Handoff}, n_i\}$, which implies that p_i reads $B[j] \neq n_j$ (since $n_j \neq n_i$), which is a contradiction. If $j = L_i[m_0] = L_i[N] = k$, then $n_j = n_k$. By Lemma 4.14, $k = owner_1$ during H'. By Lemma 4.17, $owner_2 = n_k$. By Lemma 4.16, $owner_2 = n_i$. Therefore, $n_j = n_k = owner_2 = n_i$, which is a contradiction. □

Lemma 4.20. *Given a valid history H, let H' be a complete execution of the function* **FindNextOwner** *by p_i. If p_i returns at Line 61, then $\forall j \in \{1 \dots N\}$ where $n_j \neq n_i, p_j$ is not spinning at Line 46 just before p_i reads $B[j]$ in H' at Line 55.*

Proof. Since p_i returns at Line 61 and $n_j \neq n_i$, then p_i reads $B[j] \neq n_j$ at Line 55 (by Lemma 4.19), which implies that $B[j] \neq n_j$ just before p_i reads $B[j]$. Therefore, (by Definition 4.1), p_j is not spinning at Line 46 just before p_i reads $B[j]$. □

Lemma 4.21. *Given a valid fair history H, consider a non-terminal ownership period H^o_m, for some $m \geq 0$. $\forall i \in \{1 \dots N\}$, where $n_i = owner_2$ in H^o_m, p_i is not spinning at Line 46 just before Step T_{m+1}, or there are infinitely many crash steps in H.*

Proof. I proceed by induction on m. Let $S(m)$ be the following statement: $\forall i \in \{1 \dots N\}$, where $n_i = owner$ in H^o_m, p_i is not spinning at Line 46 just before Step T_{m+1}, or there are infinitely many crash steps in H. I prove that $S(m)$ holds for $m \geq 0$.

BASE CASES ($m = 0$ and $m = 1$):

At the beginning of H^o_0, just after the auxiliary step T_0, the value of $owner = \langle \perp, \perp, 1 \rangle$ and all elements of the array B are $\perp$, indicating that no node holds the ownership of the lock. All active processes begin by executing the recovery section. The steps within this section are finite as processes bypass Lines 27-34, following the if-condition at Line 26. Consider a process p_i, which reads $B[i] \neq \textsc{Passed}$ (since $B[i] = \perp$ initially) at Line 35. Subsequently, p_i finds $owner_2 = \perp$ at Line 37 and $owner_3 = 1$ and again $owner_2 = \perp$ at Line 42, consistent with the initial state of $owner = \langle \perp, \perp, 1 \rangle$. Therefore, p_i executes the CAS operation at Line 43. Without loss of generality, assume p_i is the first process to perform this CAS operation, p_i changes $owner$ to $\langle i, n_i, 0 \rangle$. This action marks the transition from H^o_0 to H^o_1 and is defined as step T_1,

ending H_0^o and beginning H_1^o. Hence, $S(0)$ is true. Since p_i is the first process to execute the CAS at Line 43, any process p_j where $n_j = n_i$ (possibly $j = i$) reads $owner_2 = n_j$ at Line 44 and skips the loop at Line 46 due to the if-condition at Line 45. Therefore, $S(1)$ holds.

STEP $(m > 1)$:

Assume that $S(m)$ is true $\forall m \in \{0 \dots m' - 1\}$, for some integer $m' > 1$. I prove that $S(m')$ is also true. Assume for contradiction that $S(m')$ is not true. That is, there exists a process p_i, where $n_i = owner_2$ in $H_{m'}^o$, that is spinning at Line 46 just before Step $T_{m'+1}$, and there are finite crash steps in H. Hence, no process can crash an infinite number of times in H. By Definition 4.5, Step $T_{m'}$ is either a successful CAS operation executed at (A) Line 43 or (B) Line 75, or (C) a direct assignment at Line 70. Let p_j, possibly $j = i$, be the process that changes *owner* at $T_{m'}$. In the subsequent part of this proof, I will demonstrate that $S(m')$ is true in all three scenarios.

CASE A: $T_{m'}$ is a successful CAS at Line 43.

Since p_i is spinning at Line 46 just before Step $T_{m'+1}$, then, by the if-condition at Line 45, process p_i reads $owner_2 \neq n_i$ at Line 44 before Step $T_{m'+1}$ at a step Q. Therefore, either Q is (A1) in $H_{m'}^o$, or (A2) in an ownership period $H_{m''}^o$ such that $m'' < m'$.

CASE A1: Q is in $H_{m'}^o$.

By the contradiction assumption, $owner_2 = n_i$ in $H_{m'}^o$. Accordingly, p_i reads $owner_2 = n_i$ at Line 44, which contradicts that Q occurs in $H_{m'}^o$.

CASE A2: Q is in $H_{m''}^o$.

Since $T_{m'}$ is a successful CAS at Line 43, from the CAS condition and the if-condition (Line 42), I know that $owner = \langle h, n, 1\rangle$, where $j \neq h$ and $n \neq n_j = n_i$, in $H_{m'-1}^o$. Let p_l be the process that executed Step $T_{m'-1}$. In Algorithms 4.2-4.6, the value of $owner_3$ can be set to 1 only at Line 70 by returning $\langle h, n, 1\rangle$ at Line 61 from an invocation of the function **FindNextOwner** at Line 69. By Line 65, p_l reads $owner_2 = n_l \neq n_i$, which is unchanged until the end of the invocation of **FindNextOwner** under consideration (Lemma 4.14). Now, p_l returns $\langle h, n_l, 1\rangle$ from the function call at Line 69, to which p_l sets *owner* at Line 70. Then, based on the if-statement at Line 72, p_l invokes **FindNextOwner** at Line 73, and reads the value of $B[i]$ at Line 55 at a step Q', or p_l crashes. I first proceed with the proof assuming a crash-free execution of p_l, then I show that the proof of this case holds in the existence of failures.

Since Q is before $T_{m'}$, then p_i has set $B[i]$ to n_i at a prior step at Line 39 either (1) before Q' or (2) after Q'. If (1), then p_l reads $B[i] = n_i$, and thus returns at Line 59. By the if-statement at Line 74, p_i executes the CAS at Line 75. If the CAS is successful, then $T_{m'}$ is a successful CAS at Line 75, which is a contradiction. If the CAS fails, then another process has changed *owner* at $T_{m'}$. I know, by the contradiction assumption, that $owner_2 = n_i$ in $H^o_{m'}$. Therefore, p_l reads *owner* in $H^o_{m'}$ and eventually sets $B[i]$ to PASSED at Line 79, which contradicts that p_i spins just before $T_{m'+1}$. If (2), then p_i reads $owner = \langle h, n_l, 1\rangle$ at Line 41. Since $n_l \neq n_i$, p_i executes the CAS at Line 43. If p_i succeeds, then $j = i$. If p_i fails, then $j \neq i$. In both cases, I know that p_j changes *owner* at $T_{m'}$, starting $H^o_{m'}$, before Step Q, which contradicts that Q is in $H^o_{m''}$.

Now, I show that if p_l crashes before Q', the proof is still correct. I know by the contradiction assumption that p_l crashes finitely many times. Consider the last crash of p_l before Q'. If p_l crashes before Q', p_l restarts by calling **FilterRecover** at Line 19. Since p_l was executing Lines 69-83, then $B[l]$ = HANDOFF by Lemma 4.12. Hence, either *owner* is unchanged since p_l set it to $\langle h, n_l, 1\rangle$, or *owner* is changed by another process p_j at $T_{m'}$ by the contradiction assumption. If the value of *owner* is unchanged since p_l set it to $\langle h, n_l, 1\rangle$, then p_l executes Line 73-84 of **FilterHandoff** due to the if statement at Line 29, which is equivalent to the crash-free execution that leads to contradictions. If p_j, where $n_j = n_i$, executed $T_{m'}$ before p_l reads *owner* at Line 27, then p_l executes Lines 78-84, setting $B[i]$ to PASSED at Line 79, which contradicts that p_i spins just before $T_{m'+1}$.

CASE B: $T_{m'}$ is a successful CAS at Line 75.

Since $T_{m'}$ is a successful CAS at Line 75, from the CAS condition and the if-condition (Line 74), I know that p_j sets *owner* to $\langle h, n, 0\rangle$, for some h and n, at $T_{m'}$. Therefore, process p_j returns $\langle h, n, 0\rangle$ at Line 59 from an invocation of the function **FindNextOwner** at Line 73. By the contradiction assumption, $n = n_i$ (since $owner_2 = n_i$ in $H^o_{m'}$). By Lemma 4.15, $n \neq n_j$. Hence, $n_i \neq n_j$.

Since p_i is spinning at Line 46 at Step $T_{m'+1}$, then, by the if-condition at Line 45, process p_i reads $owner_2 \neq n_i$ at Line 44 before Step $T_{m'+1}$ at some step Q. Therefore, either Q is (B1) in $H^o_{m'}$, or (B2) in an ownership period $H^o_{m''}$ such that $m'' < m'$.

CASE B1: Q is in $H^o_{m'}$.

By the contradiction assumption, $owner_2 = n_i$ in $H^o_{m'}$. Accordingly, p_i reads $owner_2 = n_i$ at Line 44, which contradicts that Q occurs in $H^o_{m'}$.

CASE B2: Q is in $H^o_{m''}$.

If Step Q is in $H^o_{m''}$, then Step Q is before $T_{m'}$. After p_j sets *owner* at Line 75, p_j eventually sets $B[i]$ to PASSED at Line 79, or p_j crashes. Since $B[j] =$ HANDOFF (Lemma 4.12) and $owner_2 = n_i \neq n_j$ in $H^o_{m'}$, if p_j crashes, p_j enters the recovery section, calls **FilterRecover**, and then executes Lines 78-84. By the contradiction assumption, p_j does not crash infinitely many times, eventually setting $B[i]$ to PASSED at Line 79 in $H^o_{m'}$. Therefore, p_i does not spin at Line 46, which is a contradiction.

CASE C: $T_{m'}$ is a direct assignment at Line 70.

Since p_i is spinning at Line 46 at Step $T_{m'+1}$, then, by the if-condition at Line 45, process p_i reads $owner_2 \neq n_i$ at Line 44 before Step $T_{m'+1}$ at some step Q. Therefore, either Q is (C1) in $H^o_{m'}$, or (C2) in an ownership period $H^o_{m''}$ such that $m'' < m'$.

CASE C1: Q is in $H^o_{m'}$.

By the contradiction assumption, $owner_2 = n_i$ in $H^o_{m'}$. Accordingly, p_i reads $owner_2 = n_i$ at Line 44, which contradicts that Q occurs in $H^o_{m'}$.

CASE C2: Q is in $H^o_{m''}$.

If Step Q is in $H^o_{m''}$, then Step Q is before $T_{m'}$. Process p_j sets *owner* to $\langle h, n, a\rangle$, for some h, n and a, at $T_{m'}$ at Line 70. From the if-condition at Line 65, $owner_2 = n_j$ in $H^o_{m'-1}$. By the contradiction assumption, $n = n_i$, which is the value of $owner_2$ in $H^o_{m'}$. Subsequently, process p_j returns $\langle h, n_i, a\rangle$ at Line 59 from an invocation of the function **FindNextOwner** at Line 69. Either p_j (1) returns at Line 61 or (2) at Line 59. If (1), then $n_i = n_j$. Since $n_i = n_j$ and $n_j = owner_2$ in $H^o_{m'-1}$, then $n_i = owner_2$ in $H^o_{m'-1}$. Therefore, in $H^o_{m'-1}$, $owner_2 = n_i$, which contradicts the induction hypothesis that if $owner_2 = n_i$ in $H^o_{m'-1}$, then p_i is not spinning at Line 46 just before Step $T_{m'}$. If (2), process p_j skips Lines 73-77 (due to the if-statement at Line 72), and eventually sets $B[i]$ to PASSED at Line 79, or p_j crashes. Since $B[j] =$ HANDOFF (Lemma 4.12) and $owner_2 = n_i \neq n_j$ (Lemma 4.15) in $H^o_{m'}$, if p_j crashes, p_j enters the recovery section, calls **FilterRecover** at Line 19, and then executes Lines 78-84. By the contradiction assumption, p_j does not crash infinitely many times and eventually sets $B[i]$ to PASSED at Line 79 in $H^o_{m'}$. Therefore, p_i does not spin at Line 46, which is a contradiction.

□

Lemma 4.22. *Consider a valid fair history H. For each ownership period H^o_m, where $m \geq 0$, one of the following conditions holds:*

1. *H^o_m is finite,*
2. *H^o_m is infinite, and within this period, either:*
   (a) *every process is not active,*
   (b) *every process halts while not executing a super-passage,*
   (c) *or there are infinitely many crash steps.*

Proof. Let $S(m)$ be the following statement: If H^o_m exists in H, then H^o_m is finite, or H^o_m is infinite and within this period, either every process is not active, every process halts while not executing a super-passage, or there are infinitely many crash steps. I prove that $S(m)$ holds for $m \geq 0$ by induction on m.

BASE CASE ($m = 0$):

At the beginning of H^o_0, just after the auxiliary step T_0, the value of $owner = \langle \perp, \perp, 1 \rangle$ and all elements of the array B are $\perp$, indicating that no node holds the ownership of the lock. All active processes begin by executing the recovery section. The steps within this section are finite as processes bypass Lines 27-34, following the if-condition at Line 26. Consider a process p_i, which reads $B[i] \neq$ PASSED (since $B[i] = \perp$ initially) at Line 35. Subsequently, p_i finds $owner_2 = \perp$ at Line 37 and $owner_3 = 1$ and again $owner_2 = \perp$ at Line 42, consistent with the initial state of $owner = \langle \perp, \perp, 1 \rangle$. Therefore, p_i executes the CAS operation at Line 43. Without loss of generality, assume p_i is the first process to perform this CAS operation, p_i changes $owner$ to $\langle i, n_i, 0 \rangle$. This action marks the transition from H^o_0 to H^o_1 and is defined as step T_1, ending H^o_0 and beginning H^o_1. Therefore, H^o_0 is finite and $S(0)$ holds

STEP ($m > 0$):

Assume that $S(m)$ is true $\forall m \in \{0 \ldots m' - 1\}$, for some integer $m' > 0$. I prove that $S(m')$ is also true. If any H^o_m is infinite, then $H^o_{m'}$ does not exist, and $S(m)$ holds $\forall m \geq 0$. If H^o_m is finite $\forall m \in \{0 \ldots m' - 1\}$, then either $H^o_{m'-1}$ is terminal and $H^o_{m'}$ does not exist, thus, $S(m)$ holds $\forall m \geq 0$ by the induction hypothesis, or $H^o_{m'-1}$ is non-terminal and Step $T_{m'}$ exists by which $H^o_{m'}$ starts. According to Definition 4.5, the value of $owner$ is changed at $T_{m'}$. From Algorithms 4.2-4.6, I know that the value

of *owner* can be altered at some step where a successful CAS operation is executed at (A) Line 43 or (B) Line 75, or (C) when a direct assignment is made at Line 70. In the subsequent part of this proof, I will demonstrate that $S(m')$ is true in all three scenarios.

CASE A: $T_{m'}$ is a successful CAS at Line 43.

I aim to prove the following claim: If some process p_i changes *owner* to $\langle i, n_i, 0\rangle$ at $T_{m'}$, then either there are infinitely many crash steps in H and $S(m')$ is true, or there exists a process p_j (possibly $j = i$) that will eventually change *owner* at Line 70 or 75, which implies that Step $T_{m'+1}$ exists and that $H^o_{m'}$ is finite, proving that $S(m')$ is true.

Assume, for contradiction, that there are finite crash steps in H and that no process changes *owner* at Line 70 or 75 after $T_{m'}$. Hence, no process can crash an infinite number of times in H. Given p_i already updated *owner* to $\langle i, n_i, 0\rangle$ at $T_{m'}$, the conditions for a CAS at Line 43 are not met again, preventing further changes to *owner* through this CAS.

If any process p_j, where $n_j = n_i$, crashes before entering the CS in $H^o_{m'}$, then it restarts from the recovery section, calls **FilterRecover** and skips Lines 27-34 ($B[j] \neq$ HANDOFF, since $B[j]$ can only be set to HANDOFF in the CS by p_j at Line 67). Then, p_j calls **Filter** and skips the spin-loop at Line 46 (since $owner_2 = n_i$ in $H^o_{m'}$), eventually entering the recovery section. As a result, I shall omit sub-cases where a process p_j crashes in the entry section and recovery section.

After p_i sets *owner* to $\langle i, n_i, 0\rangle$ at $T_{m'}$, it bypasses the loop at Line 46 (due to the if-condition at Line 45), sets $B[i]$ to PASSED and $last[n_i]$ to i at Lines 48 and 50, respectively. Then, process p_i enters the recovery section of the base lock and, by the SF property of the base lock (Lemma 4.4), p_i eventually enters the CS, or there are infinitely many crashes in H. By the contradiction assumption, there are finite crash steps in H. Therefore, p_i enters CS, invokes **FilterHandoff** (Line 23) and evaluates the if-condition at Line 62. Now, either (A1) $last[n_i] = i$, or (A2) $last[n_i] \neq i$.

CASE A1: $last[n_i] = i$.

When p_i verifies $last[n_i] = i$ at Line 62, it reads $owner = \langle i, n_i, 0\rangle$ at Line 63. The if-condition at Line 65 prompts p_i to execute steps from Line 67 to 86. Following this, p_i calls **FindNextOwner** at Line 69. This results in p_i either returning $\langle h, n, 0\rangle$ at Line 59, for some h and n where $n \neq n_i$ (Lemma 4.15), or returning $\langle h, n, 1\rangle$ at Line 61. In both scenarios, p_i updates *owner* at Line 70 at Step $T_{m'+1}$, which is a contradiction.

CASE A2: $last[n_i] \neq i$.

When p_i reaches Line 62, it finds $last[n_i] \neq i$. There are two scenarios:

(i) $counter[n_i] \geq$ HandoffThreshold: p_i reads $owner = \langle i, n_i, 0 \rangle$ at Line 63. The condition at Line 65 prompts p_i to execute steps from Line 67 to 86. The remainder of this case follows CASE A1, which leads to a contradiction.

(ii) $counter[n_i] <$ HandoffThreshold: Before entering the CS of the base lock, p_i sets $last[n_i]$ to i at Line 50. Since p_i reads $last[n_i] \neq i$ at Line 62, then another process p_j (where $j \neq i$ and $n_j = n_i$) must have updated $last[n_i]$ to j after p_i executed Line 50. Subsequently, both p_i and p_j enter the recovery section of the base lock. According to the SF property of the base lock (Lemma 4.4), both processes eventually enter the CS of the base lock, or there are infinitely many crashes in $H^o_{m'}$. By the contradiction assumption, no process crashes infinitely many times in H. Therefore, both p_j and p_i enter the CS of the base lock. If p_j enters the CS of the base lock before p_i, it invokes **FilterHandoff** and reads $last[n_j] = j$ at Line 62, where CASE A1 applies. Consequently, p_j changes *owner* at Line 70, leading to a contradiction. If, p_i enters the CS of the base lock before p_j, then p_i skips Lines 63-84 (since $last[n_i] \neq i$), increments $counter[n_i]$ at Line 85, and executes the exit section of the base lock. Following this, p_j enters the CS of the base lock, calls **FilterHandoff** at 23, and based on $last[n_j]$ and $counter[n_i]$, follows either:

- CASE A1 if $last[n_j] = j$,
- CASE A2(i) if $counter[n_i] \geq$ HandoffThreshold, or
- CASE A2(ii) if $last[n_j] \neq j$ and $counter[n_i] <$ HandoffThreshold.

CASE A1 and CASE A2(i) lead to contradictions. As **FilterHandoff**, which is in the CS of the base lock (Corollary 4.8), increments $counter[n_i]$ at Line 85, the number of consecutive applications of A2(ii) is bounded by HandoffThreshold. Beyond this limit, A2(i) applies, leading to contradiction.

Eventually, all scenarios in CASE A lead to a contradiction by showing that *owner* is changed at Line 70 which implies that Step $T_{m'+1}$ exists and that $H^o_{m'}$ is finite, proving that $S(m')$ is true.

CASE B: $T_{m'}$ is a successful CAS at Line 75.

Assume that p_i is the process that executed the successful CAS at Line 75 at Step $T_{m'}$. From Line 74, p_i sets *owner* to $\langle h, n, 0\rangle$, for some integer $h \in \{1 \dots N\}$ and $n = n_h$ (Lemma 4.17). I aim to prove the following claim: If p_i changes *owner* to $\langle h, n_h, 0\rangle$ at $T_{m'}$ at Line 75, then either there exists a process p_j (possibly $j = h$) that will eventually change *owner* at Line 70 or 75, which implies that Step $T_{m'+1}$ exists and that $H^o_{m'}$ is finite, or $H^o_{m'}$ is infinite and there are infinitely many crashes in $H^o_{m'}$, proving that $S(m')$ is true.

Assume, for contradiction, that no process changes *owner* at Line 70 or 75 after $T_{m'}$ and there are finite crash steps in H. Hence, no process can crash infinitely many times in H. Given p_i already updated *owner* to $\langle h, n_h, 0\rangle$ at $T_{m'}$ at Line 75, the conditions for a CAS at Line 43 are not met, preventing further changes to *owner* through this CAS.

By Lemma 4.4, p_i eventually executes the exit section of the base lock after $T_{m'}$. Let H' be the sub-history that starts with Step $T_{m'}$ and ends just before p_i invokes the exit function of the base lock at Line 24. In H', process p_i changes $B[h]$ to PASSED at Line 79, after Step $T_{m'}$, or crashes. If p_i crashes, it starts from the NCS and executes **FilterRecover**. By Lemma 4.12, $B[i]$ = HANDOFF just before the crash step of p_i under consideration, and I know (by the contradiction assumption) that *owner* is unchanged during both $H^o_{m'}$, and H' (since $H' \sqsubseteq H^o_{m'}$). As a result, p_i calls **FilterHandoff** at Line 34 and executes Lines 78-84, setting $B[h]$ to PASSED at Line 79. In all cases, process p_i changes $B[h]$ to PASSED in H'.

Since p_i sets *owner* to $\langle h, n_h, 0\rangle$ at Step $T_{m'}$, then p_i returns $\langle h, n_h, 0\rangle$ at Line 59 from the invocation of **FindNextOwner** at Line 73 in a prior step, which implies that $n_h \neq n_i$ (by Lemma 4.15). In **FindNextOwner**, p_i returns n_h only if it reads $B[h] = n_h$ at Line 55 (by the loop-condition at Line 57). The value of $B[h]$ can be set to n_h only by a process p_h at Line 39. By Lemmas 4.5 and 4.6, process p_h does not enter the CS in H'. Therefore, p_h is in the NCS, recovery section, or entry section during H'.

Now I show that after H', process p_h is guaranteed to eventually enter the CS. All steps in **FilterRecover** and **Filter** are bounded by p_h's own steps except the spin loop at Line 46. Since p_i changes $B[h]$ to PASSED, p_h skips the loop at Line 46 unless $B[h]$ is changed. The value of $B[h]$ can only be changed from PASSED to HANDOFF at Line 67, $\perp$ at Line 87, or n_h at Line 39, only by p_h. Lines 67 and 87 imply that p_h entered the CS. As for Line 39, either p_h reads $n_h = owner_2$ at Line 45, or p_h crashes, restarts from NCS, executes Line 26, skips Lines 27-34, and then reads $n_h = owner_2$ at Line 36. Since *owner* remains

unchanged during $H^o_{m'}$ (contradiction assumption), p_h reads $n_h = owner_2$ at Line 36, even if p_h crashes multiple times. Therefore, eventually p_h reads $n_h = owner_2$ at Line 45 or 37 and skips the loop at Line 46, sets $B[h]$ to PASSED (Line 48) and $last[h]$ to h (Line 50), and enters the recovery section of the base lock. By the SF property of the base lock (Lemma 4.4), p_h eventually enters the CS of the base lock. Subsequently, p_h calls **FilterHandoff**, and either reads (B1) $last[h] = h$ or (B2) $last[h] \neq h$ at Line 62. Since $owner = \langle h, n_h, 0\rangle$ and $B[h] =$ PASSED, then CASEs B1 and B2 of this particular invocation of **FilterHandoff** follow CASEs A1 and A2, respectively, which both show that *owner* is changed at Line 70 which implies that Step $T_{m'+1}$ exists and that $H^o_{m'}$ is finite, or $H^o_{m'}$ is infinite and there are infinitely many crashes in $H^o_{m'}$, leading to a contradiction.

CASE C: $T_{m'}$ is a direct assignment at Line 70.

Assume p_i is the process that executes Line 70 at $T_{m'}$. Process p_i sets *owner* to some value returned from the invocation of **FindNextOwner** at Line 69 at a prior step by p_i. In **FindNextOwner**, process p_i either returns at (C1) Line 59 or (C2) Line 61.

CASE C1: p_i returns at Line 59.

If p_i returns at Line 59, then, p_i sets *owner* to $\langle h, n, 0\rangle$, for some integer $h \in \{1 \dots N\}$ and $n = n_h$ (Lemma 4.17). I aim to prove the following claim: If p_i changes *owner* to $\langle h, n_h, 0\rangle$ at $T_{m'}$ at Line 70, then either there exists a process p_j (possibly $j = h$) that will eventually change *owner* at Line 70 or 75, which implies that Step $T_{m'+1}$ exists and that $H^o_{m'}$ is finite, or $H^o_{m'}$ is infinite and there are infinitely many crashes in $H^o_{m'}$, proving that $S(m')$ is true.

Assume, for contradiction, that no process changes *owner* at Line 70 or 75 after $T_{m'}$ and there are finite crash steps in H. Hence, no process crashes infinitely many times in H. Given that p_i already updated *owner* to $\langle h, n_h, 0\rangle$ at $T_{m'}$ at Line 70, the conditions for a CAS at Line 43 are not met, preventing further changes to *owner* through this CAS.

Either p_i skips Lines 73-77 (due to the if-condition at Line 72), or p_i crashes. If p_i crashes, it starts from the NCS and executes **FilterRecover**. By Lemma 4.12, $B[i] =$ HANDOFF just before the crash step of p_i, and I know that *owner* is unchanged during $H^o_{m'}$. As a result, p_i calls **FilterHandoff** at Line 34 and executes Lines 78-84, skipping Lines 73-77. In all cases, process p_i skips Lines 73-77. Therefore, this call to **FilterHandoff** follows CASE B, which leads to a contradiction by showing that *owner* is changed at Line 70

which implies that Step $T_{m'+1}$ exists and that $H^o_{m'}$ is finite, or $H^o_{m'}$ is infinite and there are infinitely many crashes in $H^o_{m'}$, proving that $S(m')$ is true.

CASE C2: p_i returns at Line 61.

If p_i returns at Line 61, then, p_i sets *owner* to $\langle h, n_i, 1\rangle$, for some integer $h \in \{1 \dots N\}$. The ownership period $H^o_{m'}$ is either finite, which implies that $S(m')$ is true, or infinite. I aim to prove the following claim: If p_i changes *owner* to $\langle h, n_h, 1\rangle$ at $T_{m'}$ at Line 70 and $H^o_{m'}$ is infinite, then $\forall j \in \{1 \dots N\}$, either p_j is not active, p_j halts while not executing a super-passage, or there are infinitely many crash steps in $H^o_{m'}$.

Suppose for contradiction that $H^o_{m'}$ is infinite and there exists a process p_j (possibly $j = i$) that is active, does not halt while executing a super-passage, and there are finite crash steps in H. Hence, no process crashes infinitely many times in H, including p_j. Therefore, p_j is executing infinitely many ordinary steps in $H^o_{m'}$. There are two loops in Algorithms 4.2-4.6: The repeat-until-loop at Lines 53-57 and the spin loop at Line 46. Since the repeat-until-loop is bounded by p_j's own steps (Lemma 4.1), then p_j must be infinitely executing the loop at Line 46, which implies that p_j is spinning at Line 46 infinitely.

Process p_i executes Line 70 at Step $T_{m'}$. Then, either p_i completes its super-passage (the rest of the exit section) or crashes. If p_i crashes in Lines 71-84 (i.e., before setting $B[i]$ to PASSED), then p_i restarts in the NCS and $B[i] =$ HANDOFF (Lemma 4.12). As a result, p_i enters the recovery section, calls **FilterRecover** at Line 19, calls **FilterHandoff** at Line 30, and executes Lines 73-84, including setting $B[i] =$ PASSED. Then, p_i returns from **FilterHandoff**, calls **Filter**, and skips Lines 36-46 (due to the if-condition at Line 35), including the spin loop at Line 46. If p_i crashes after setting $B[i]$ to PASSED at Line 84, then p_i restarts, enters recovery section, calls **FilterRecover**, skips Lines 27-34 (since $B[i] =$ PASSED $\neq$ HANDOFF), calls **Filter**, and skips Lines 36-46 (due to the if-condition at Line 35), including the spin loop at Line 46. Therefore, in all cases p_i does not spin at Line 46 during $H^o_{m'}$, which implies that $j \neq i$.

After p_i sets *owner* to $\langle h, n_i, 1\rangle$ at Line 70, it invokes **FindNextOwner** at Line 73 (due to the if-condition at Line 72) and returns some value v at Line 59 or 61. If p_i returns at Line 59, then $v_3 = 0$, and p_i executes the CAS at Line 75 due to the if-statement at Line 74. If p_i performs a successful CAS, then $H^o_{m'}$ is finite, which is a contradiction. If p_i fails the CAS, then some

other process changes *owner* after $T_{m'}$, which implies that $H^o_{m'}$ is finite, hence another contradiction. Therefore, p_i returns at Line 61 from the invocation of **FindNextOwner** at Line 73.

Since p_j is spinning at Line 46, then p_j reads $owner_2 \neq n_j$ at Line 44 at a step, call it Q, either before or after $T_{m'}$. If Q occurs after $T_{m'}$, then by Line 45, $n_j \neq n_i$. If Q occurs before $T_{m'}$, by the contra-positive of Lemma 4.21, since p_j is spinning at Line 46 just before $T_{m'}$, then $owner_2 \neq n_j$ in $H^o_{m'-1}$. By the if-condition at Line 65, p_i reads $owner_2 = n_i$ and $owner_3 = 0$ at Line 63, which implies that $owner_2 = n_i$ in $H^o_{m'-1}$. Therefore, $n_j \neq n_i$.

Given that p_j is spinning at Line 46, process p_j must have set $B[i]$ to n_j at Line 39 at some prior step Q_j. Since p_i returns at Line 61 from the invocation of **FindNextOwner** at Line 73, then p_i reads $B[j]$ at Line 55 at some step Q_i. By Lemma 4.19, since p_i returns at Line 61 and $n_i \neq n_j$, then p_i reads $B[j] \neq n_j$ at Step Q_i. Therefore, Q_i occurs before p_j sets $B[i]$ to n_j at Line 39 (Step Q_j). Then, p_j reads $owner = \langle h, n_i, 1\rangle$ at Line 41 after Q_j, and then performs a successful CAS at Line 43, contradicting that $H^o_{m'}$ is infinite.

Accordingly, I prove the claim that if p_i changes *owner* to $\langle h, n_h, 1\rangle$ at $T_{m'}$ at Line 70 and $H^o_{m'}$ is infinite, then $\forall j \in \{1 \ldots N\}$, either p_j is not active, p_j halts while not executing a super-passage, or there are infinitely many crash steps in $H^o_{m'}$. Therefore, by this claim, $S(m')$ holds if $H^o_{m'}$ is infinite, and it also holds directly when $H^o_{m'}$ is finite.

□

Lemma 4.23. *Given a valid fair history H and two consecutive ownership periods H^o_m and H^o_{m+1}, for some $m > 0$, either T_m or T_{m+1} is a direct assignment to owner at Line 70.*

Proof. In Algorithms 4.2-4.6, the CAS operations at Lines 43 and 75 change $owner_2$ from 1 to 0. If T_m is a successful CAS at Line 43 or 75, then $owner_2 = 0$ in H^o_m. Therefore, Step T_{m+1} cannot possibly be a successful CAS at Line 43 or 75, which implies that T_{m+1} is a direct assignment at Line 70. □

Lemma 4.24. *Given a valid fair history H and a non-terminal ownership period H^o_m, for some $m \geq 0$, if a process p_i sets owner at Line 70 at Step T_{m+1}, then p_i completes an execution of the function* **FindNextOwner** *in H^o_m.*

Proof. In Algorithms 4.2-4.6, if p_i sets *owner* at Line 70 at Step T_{m+1}, then p_i invokes **FindNextOwner** at Line 69 and returns (since the function steps are bounded by p_i's

own steps) at a step before T_{m+1}. Let H' be the last complete execution of the function **FindNextOwner** by p_i before T_{m+1} occurs. I prove that $H' \sqsubset H^o_m$. Line 70 can be reached by calling the function **FilterHandoff** at Lines 23 or 32. Both scenarios involve reading $k = owner_1$ at Lines 27 and 63, respectively. By Lemma 4.14, *owner* is unchanged until the end of H', which implies that $H' \sqsubset H^o_m$. □

Lemma 4.25. *Given an infinite valid fair history H, if p_i is spinning at Line 46, then p_i eventually enters the CS, or there are infinitely many crash steps in H.*

Proof. Assume for contradiction that p_i is spinning at Line 46, p_i never enters the CS, and there are finite crash steps in H. Hence, no process crashes infinitely many times in H, including p_i. Let the ownership period in which p_i starts spinning at Line 46 be $H^o_{m'}$, for some $m' \geq 0$. Since p_i is active, executing a super passage, and no process crashes infinitely many times in H, then $\forall m \geq m'$, H^o_m is finite (Lemma 4.22).

My proof strategy is to show that there exists a step T_{m_0}, for some $m_0 > m'$, where a process p_j eventually invokes the function **FindNextOwner**, reads $B[i] = n_i$ at Line 55, returns at Line 59 (where $tmp = n_i$), and sets $owner_2$ to n_i at Line 70 during Step T_{m_0}. I then show that, following T_{m_0}, p_j eventually sets $B[i]$ to PASSED at Line 79, allowing p_i to break out of the spin-loop at Line 46 and eventually enter the CS.

If p_i crashes while spinning at Line 46, it restarts and either eventually enters the CS, which would contradict my assumption, or continues spinning on $B[i]$ at Line 46. Since, by the contradiction assumption, p_i does not crash infinitely many times, it must eventually begin spinning at Line 46 without crashing again. In this proof, I consider the last time p_i starts spinning at Line 46.

First, I prove that there is an ownership period after which each T_m is a direct assignment of *owner* at Line 70. Precisely, I prove the following claim: $\forall m \geq m' + 3, T_m$ is a direct assignment at Line 70. Suppose for contradiction that there exists $m'' \geq m' + 3$ such that $T_{m''}$ is a successful CAS at Line 43 or 75. By the CAS condition (at Line 43 or 75), $owner_3 = 1$ just before Step $T_{m''}$. Therefore, *owner* is set by some process p_j to $\langle h, n, 1\rangle$, for some h and n, at $T_{m''-1}$.

I know that p_i is spinning at Line 46 in $H^o_{m'}$ and that $m' < m'' - 2$. Therefore, $\forall m > m'$, process p_i is spinning at Line 46 during the whole period of H^o_m, including the time just before Step $T_{m''-1}$. By Lemma 4.23, since $T_{m''}$ is a successful CAS operation at Line 43 or 75, then $T_{m''-1}$ is a direct assignment at Line 70. By Lemma 4.24, p_j completes an execution of the function **FindNextOwner** in $H^o_{m''-2}$. Since p_j sets $owner_3$ to 1 at Line 70 at Step $T_{m'-1}$, then p_j returns from the invocation of **FindNextOwner** under consideration at Line 61. By Line 61, $n = n_j$. By Lemma 4.20, as p_j returns at Line 61

and p_i is spinning at Line 46 during the whole period of $H^o_{m''-2}$, including the time just before p_j reads $B[i]$ at Line 55 in the invocation of the function **FindNextOwner** under consideration, then $n_i = n_j$. By Lemma 4.16, since p_j returns at Line 61, then $owner_2 = n_j$ in $H^o_{m''-2}$. As a result, $n_i = owner_2$ in $H^o_{m''-2}$.

However, by Lemma 4.21, since $n_i = owner_2$ in $H^o_{m''-2}$, p_i is not spinning at Line 46 just before Step $T_{m''-1}$, which is a contradiction, or there are infinitely many crash steps in H, contradicting my assumption. As a result, I prove that $\forall m \geq m' + 3, T_m$ is a direct assignment at Line 70.

Let k^m be the value assigned to $owner_1$ at Step T_m. Let p^m be the process executed Step T_m. By Lemma 4.24, p^m completes an execution of the function **FindNextOwner** in H^o_{m-1}, denoted as H'_{m-1}, before T_m. Let $L^m[1 \dots N]$ be a sub-sequence of p^m's search list (Definition 4.3), such that, for some integers $h, x_m \in \{1 \dots N\}$, $h \in L^m[1 \dots x_m]$ if and only if p^m reads $B[h]$ at Line 55 in H'_{m-1}. I prove that there exists $m_0 \geq m' + 3$ such that $i \in L^{m_0}[1 \dots x_{m_0}]$. Assume for contradiction that such m_0 does not exist. Therefore, $\forall m \geq m' + 3, p^m$ does not read $B[i]$ at Line 55 in H'_{m-1}. By Lemma 4.14, the variable k is set to $owner_1 = k^{m-1}$ and $owner$ is unchanged until the end of in H'_{m-1}. Accordingly, in H'_{m-1}, p^m is guaranteed to read $B[(k^{m-1} \mod N) + 1]$ at Line 55 in the first iteration of the repeat-until-loop (Lines 52-57). Therefore, the minimum number of iterations is 1, such that $L^m[1] = (k^{m-1} \mod N) + 1$, as per Line 54. Assume that $\forall m \geq m' + 3, x_m = 1$. Then, p^m returns from **FindNextOwner** and sets $owner_1$ to $k^m = (k^{m-1} \mod N) + 1$ at Line 70 at Step T_m. As a result,

$$
\begin{aligned}
L^{m'+3}[1] &= (k^{m'+2} \mod N) + 1 \\
&= k^{m'+3} \\
L^{m'+4}[1] &= (k^{m'+3} \mod N) + 1 \\
&= (L^{m'+3}[1] \mod N) + 1 \\
&= L^{m'+3}[2] \\
L^{m'+5}[1] &= (k^{m'+4} \mod N) + 1 \\
&= (L^{m'+3}[2] \mod N) + 1 \\
&= L^{m'+3}[3] \\
&\vdots
\end{aligned}
$$

$$\begin{aligned} L^{m'+3+y}[1] &= (L^{m'+3}[y] \mod N) + 1 \\ &= L^{m'+3}[y+1] \end{aligned}$$

Since, $\forall m > m'$, process p_i is spinning at Line 46 during the whole period of H^o_m, by Lemma 4.22, $\forall m > m, H^o_m$ is finite. Now, consider $y = N - 1$, such that

$$\begin{aligned} & L^{m'+3}[1] \circ L^{m'+4}[1] \circ \cdots \circ L^{m'+3+y}[1] \\ &= L^{m'+3}[1] \circ L^{m'+4}[1] \circ \cdots \circ L^{m'+3+N-1}[1] \\ &= L^{m'+3}[1] \circ L^{m'+3}[2] \circ \cdots \circ L^{m'+3}[N] \\ &= L^{m'+3}[1 \ldots N] \end{aligned}$$

By Definition 4.3, $i \in L^{m'+3}[1 \ldots N]$. Therefore, there exists an integer m_0, where $m' + 3 \leq m_0 \leq m' + 2 + N$ such that $L^{m_0}[1] = i$, which contradicts that such m_0 does not exist.

Since, $\forall m > m'$, process p_i is spinning at Line 46 during the whole period of H^o_m, including the time just before T_{m+1}, by the contra-positive of Lemma 4.21, $n_i \neq owner_2$ in all H^o_m, including $H^o_{m_0-1}$. Now, assume that p_j is the process executed Step T_{m_0} (i.e., p^{m_0}). In Algorithms 4.2-4.6, process p_j reads *owner* before H'_{m_0} at either Line 27 or 63, and *owner* is unchanged until the end of H'_{m_0} (Lemma 4.14). Therefore, by the if-statements at Lines 31 and 65, $n_j = owner_2$ in $H^o_{m_0-1}$. Accordingly, $n_j \neq n_i$.

Since $i \in L^{m_0}[1 \ldots x_{m_0}]$, for some $m_0 \geq m' + 3$, then p_j reads $B[i] = n_i$ at Line 55. Since $n_i \neq n_j$, then p_j terminates the loop (by the condition at Line 57), returns $\langle k^{m_0}, n_i, 0 \rangle$ at Line 59, and then sets *owner* to $\langle k^{m_0}, n_i, 0 \rangle$ at Step T_{m_0}. Subsequently, p_j skips Lines 73-77 (if-condition at Line 72), and eventually sets $B[i]$ to PASSED at Line 79, or p_j crashes. Since $B[j] =$ HANDOFF (Lemma 4.12) and $owner_2 = n_i \neq n_j$, if p_j crashes, p_j enters the recovery section, calls **FilterRecover**, and then executes Lines 78-84, setting $B[i]$ to PASSED at Line 79. By the contradiction assumption, p_j does not crash infinitely many times and eventually sets $B[i]$ to PASSED at Line 79 in $H^o_{m_0}$. Therefore, p_i does not spin at Line 46 and enters the recovery section of the base lock. By the SF property of the base lock (Lemma 4.4), p_i eventually enters the CS, which is a contradiction. □

Lemma 4.26. *Given a valid fair history H, the RF lock satisfies SF in H.*

Proof. Assume for contradiction that the RF lock does not satisfy the SF property. That is, given an infinite fair history H, there exists a process p_i that is in the recovery section or

entry section after some finite prefix of H, and p_i never enters the CS, and no process crashes infinitely many times in H. Since p_i is in the **Recover** or **Enter** functions (Algorithm 4.1), then either (A) p_i is in the recovery section or entry section of the base lock (Line 21 or 22, respectively), or p_i is executing the (B) **Filter** or (C) **FilterRecover** functions (Line 20 or 19, respectively).

CASE A: p_i is in the recovery section or entry section of the base lock.

By the SF property of the base lock (Lemma 4.4), p_i eventually enters the CS, which is a contradiction.

CASE B: p_i is executing the **Filter**.

All steps in **Filter** are bounded by p_i's own steps except when p_i is spinning at Line 46 (Definition 4.1). By the contradiction assumption, process p_i does not crash infinitely many times in H. Consider the sub-history H' which is the suffix of H after the last crash of p_i. Then, either p_i does not spin on $B[i]$ at Line 46 in H' and eventually enters the CS, or p_i spins on $B[i]$ at Line 46 in H' and, by Lemma 4.25 eventually enters the CS. Both cases lead to contradiction.

CASE C: p_i is executing the **FilterRecover**.

By Lemma 4.1, the function **FilterRecover** is bounded by p_i's own steps, and therefore, p_i eventually invokes the function **Filter** at Line 20, which is CASE B.

□

Lemma 4.27. *Given a valid history H, if the base lock satisfies the ME, CSR and SF properties in H, the RF lock satisfies the ME, CSR and SF properties in H.*

Proof. Given a valid history H, by Lemmas 4.5, 4.6 and 4.26, the RF lock satisfies the ME, CSR and SF properties, respectively. □

Lemma 4.28. *Given a fair history H, if a process p_i reads some value v of some variable, then v is flushed to PM before p_i writes into any other variable.*

Proof. I carry out the proof by showing that for every possible read in Algorithms 4.2-4.6, the read value is explicitly flushed before modifying any other variable by the same process. If a spontaneous flush occurred, then it directly follows that the read value is flushed before modifying any other variable by the same process.

CASE A: $B[i]$ at Line 26.

Process p_i reads $B[i] = \text{HANDOFF}$ at Line 26. $B[i]$ can be set to HANDOFF only by p_i at Line 67 in the **FilterHandoff** function. Since $B[i] = \text{HANDOFF}$ at the time p_i executes Line 26, then p_i crashes in **FilterHandoff** before executing Line 83. By Corollary 4.8, p_i crashed in the CS. By the CSR property (Lemma 4.4), no other process can invoke **FilterHandoff**. Therefore, no other process can possibly change $B[i]$ (Line 79). Now, either p_i crashed (A1) before or (A2) after flushing $B[i]$ at Line 68 in **FilterHandoff**.

CASE A1 p_i crashed before flushing $B[i]$ at Line 68 in **FilterHandoff**.

By the if-statement at Line 65, $owner_2 = n_i$ and $owner_3 = 0$ at the time p_i reads *owner* at Line 63 in **FilterHandoff**. Since p_i crashed immediately before flushing $B[i]$ at Line 68, then no other process could possibly modify *owner* in **FilterHandoff** (due to ME and CSR), or at Line 43 in **Filter** because of the condition ($owner_3 = 1$) at Line 42. Therefore, p_i reads $owner_2 = n_i$ and $owner_3 = 0$ and, accordingly, invokes **FilterHandoff** at Line 32, which includes flushing $B[i]$ at Line 49.

CASE A2 p_i crashed after flushing $B[i]$ at Line 68 in **FilterHandoff**.

Since no process can possibly change $B[i]$ (Line 79) due to the CSR property, then p_i reads a flushed value of $B[i]$ at Line 26.

CASE B: *owner* at Line 27.

The value is flushed at p_i's next step at Line 28.

CASE C: $B[i]$ at Line 35.

The value is either flushed or overwritten by p_i at Line 40 or 48, respectively, before p_i modifies any other variable.

CASE D: *owner* at Line 36.

The value is flushed by p_i at Line 38 before modifying any other variable.

CASE E: *owner* at Line 41.

The value is either overwritten or flushed by p_i at Line 43 or 47, respectively, before p_i modifies any other variable.

CASE F: *owner* at Line 44.

The value is flushed by p_i at Line 47 before modifying any other variable.

CASE G: $B[i]$ at Line 46.

The value is overwritten by p_i at Line 48, before p_i modifies any other variable.

CASE H: $B[i]$ at Line 55.

The value is flushed at p_i's next step at Line 56.

CASE I: $counter[n_i]$ at Line 62.

The value of $counter[n_i]$ can only be set at Line 85 and flushed at Line 86 by some process p_j (possibly $i = j$). Since p_i at Line 62, by the ME property (Lemma 4.4), p_j must have set the value of $counter[n_i]$ and flushed it at Lines 85 and 86, respectively, before p_i reads the value at Line 62.

CASE J: $last[n_i]$ at Line 62.

The value is flushed by p_i at Line 66 before modifying any other variable.

CASE K: $owner$ at Line 63.

The value is flushed at p_i's next step at Line 64.

□

Lemma 4.29. *Given any fair history H, if the base lock satisfies the ME, CSR and SF properties in the presence of system-wide failures, the RF lock satisfies the ME, CSR and SF properties in the presence of system-wide failures.*

Proof. Let H be a fair execution history of the RF lock algorithm. Let system-wide crash failures be denoted as $C_1, C_2, C_3, \ldots$. For $m \geq 0$, let H_m be the sub-history that is followed by the crash C_{m+1}, if C_{m+1} exists. If C_{m+1} does not exist, then either there are no system-wide crashes in H, or, for some $m' < m + 1$, $C_{m'}$ is the last system-wide failure in H. Therefore,

$$H = H_0 \circ C_1 \circ H_1 \circ C_2 \circ \cdots$$

Let $H(m)$ be the prefix of H, such that

$$H(m) = \begin{cases} H_0 & \text{if } m = 0, \\ H_0 \circ C_1 \circ \ldots \circ C_m \circ H_m & \text{if } m > 0. \end{cases}$$

First, I prove that, for all $m \geq 0$, the prefix history $H(m)$ can be simulated by ordinary and independent crash steps. The proof will be carried out by induction on m.

BASE CASE: $m = 0$.

The proof is straightforward since there is no system-wide crash failure in $H(0) = H_0$.

STEP: $m > 0$.

Assume for induction that, for some $m' > 0$, the prefix history $H(m'-1) = H_0 \circ C_1 \circ \ldots \circ C_{m'-1} \circ H_{m'-1}$ can be simulated by ordinary and independent crash steps. Then, I show that the prefix history $H(m') = H_0 \circ C_1 \circ \ldots \circ C_{m'-1} \circ H_{m'-1} \circ C_{m'} \circ H_{m'}$ can also be simulated by ordinary and independent crash steps.

By the induction assumption, $H(m'-1)$ can be simulated by ordinary and independent crash steps, and so is its prefix $H_0 \circ C_1 \circ \ldots \circ C_{m'-1}$. In addition, I know that there exist no system-wide failures in $H_{m'}$. Now, I show that $H_{m'-1} \circ C_{m'}$ can be simulated by a sub-history H' of sequence of individual process ordinary and crash steps. I construct H' that replaces $H_{m'-1} \circ C_{m'}$ such that:

1. The system crash $C_{m'}$ is replaced by a sequence of consecutive crash steps of all processes. Let that sequence be C.
2. For every process, remove all its ordinary steps in H', starting from the first read or write of an unflushed value by that process.

Step 1 resets all processes to their initial states, and removes the system-wide failure in H'. In Algorithms 4.2-4.6, all writes are atomic, and processes ensure that any variable they read is flushed before modifying other variables (Lemma 4.28). Therefore, Step 2 makes H' equivalent to a modified version of the sub-history $H_{m'-1} \circ C_{m'}$, such that, instead of the system-wide failure, each process crashes right before the first read or write of an unflushed value in $H_{m'-1}$. Therefore, all removed steps in H' are consecutive steps of reads or unflushed writes, which is identical to the effect of $C_{m'}$. Hence, the states of the system at the end of $C_{m'}$ and H' are identical. As a result, $H(m') = H_0 \circ C_1 \circ \ldots \circ C_{m'-1} \circ H' \circ H_{m'}$ can be simulated by ordinary and independent crash steps.

Assume for contradiction that, in the presence of system-wide failures, the RF lock does not satisfy the ME, CSR and SF properties in H. First, I construct a history H' by recursively simulating all prefixes of H, i.e., $H(0), H(1), ...$ by equivalent counterparts that only consist of individual process ordinary and crash steps. Since the RF lock does not satisfy the ME, CSR and SF properties in H, then the RF lock does not satisfy the ME, CSR and SF properties in H'. Since H' only consists of individual process ordinary and crash steps, then, by Lemma 4.27, the RF lock satisfies the ME, CSR and SF properties in H', which is a contradiction. □

Lemma 4.30. *If the base lock's worst-case RMR complexity per passage is O(h), then the worst-case RMR complexity of the RF lock algorithm is O(h + N) per passage in the CC model.*

Proof. Algorithms 4.2-4.6 have the following three loops:

1. The spin-loop at Line 46. In Algorithms 4.2-4.6, $B[i]$ can be changed to a value $v \neq$ Passed at Lines 48, 87 and 83 only by p_i. Therefore, if $B[i]$ is changed to Passed while p_i is spinning on $B[i]$ at Line 46, then p_i is guaranteed to reads $B[i] =$ Passed and break out of the loop. As a result, Process p_i incurs a maximum of 2 RMRs in this loop: one when reading $B[i] \neq$ Passed and another when $B[i] =$ Passed.

2. The repeat-until loop at Lines 53-57. Process p_i incurs one RMR per iteration at Line 55. Since this loop is bounded by $O(N)$ iterations (Lemma 4.1), p_i incurs $O(N)$ RMRs while executing this loop.

3. The for-loop at Lines 78-80. Process p_i incurs one RMR per iteration at Line 79. Since this loop is also bounded by $O(N)$ iterations (Line 78), p_i incurs $O(N)$ RMRs while executing this loop.

Let X be the maximum number of steps a process takes per passage, excluding the steps taken in the three loops of Algorithms 4.2-4.6. Since any step $x \in X$ occurs outside of a loop, the number of such steps, X, is bounded by a constant C, which represents all the steps in Algorithms 4.2-4.6 that are not part of any loop.

Consequently, the worst-case RMRs per passage for p_i consist of the RMRs incurred within the three loops, along with those incurred in the maximum number of steps outside the loops, and those involved when accessing the base lock. This results in a total of $O(2 + N + N + C + h) = O(h + N)$. □

Theorem 4.31. *If the base RME lock satisfies mutual exclusion, starvation-freedom and critical section re-entry properties and has a worst-case RMR complexity $O(h)$ per passage in the CC model, then the RF lock satisfies mutual exclusion, deadlock-freedom, starvation-freedom and critical section re-entry properties and has a worst-case RMR complexity $O(h+N)$ per passage in the CC model.*

Proof. By Lemma 4.27, if the base lock satisfies the ME, CSR and SF properties in the individual model, then the RF lock satisfies the ME, CSR and SF properties in the individual failure model. By Lemma 4.29, if the base lock satisfies the ME, CSR and SF properties in the system-wide failure model, then the RF lock satisfies the ME, CSR and SF properties in the system-wide failure model. Moreover, by Lemma 4.30, the worst-case RMR complexity of the RF lock is $O(h + N)$ per passage. □

4.5 Experimental Evaluation

In this section, the proposed RF algorithm is evaluated against Dice et al.'s well-known cohort lock [27]. Two RME algorithms, namely GH and JJJ, are used as the base locks for the RF algorithm and the local and global locks for the cohort technique. The cohort locks, built with GH and JJJ, are called *cohort-GH* and *cohort-JJJ*, respectively. In addition, I compare the NUMA-aware locks, RF and cohort, to a *baseline* algorithm that simulates the desired behavior of a NUMA-aware lock (i.e., handing over the lock between nodes every some number of lock acquisitions). The algorithm consists of M Boolean variables, each corresponding to a NUMA node, and an additional thread, called the scheduler, that sets only one of the Boolean variables and switches between them in a round-robin fashion. A process p_i accesses the RME lock only if the Boolean variable that corresponds to n_i is set to `True`, otherwise, p_i spins on that variable. Switching between nodes occurs every time interval $t = \frac{\text{number of threads on } n_i}{N} \times \textsc{TimeThreshold}$, where TIMETHRESHOLD is a predefined time in microseconds. I call this algorithm the Dedicated Scheduler-Based Filter (DSF) lock. Although DSF is impractical as it requires an additional thread, it reflects the desired behaviour of some NUMA-aware locks, including the cohort and RF. In addition, since DSF has a dedicated process that handles the lock hand-offs between nodes, I expect DSF to be faster than both the RF and cohort algorithms. As a result, I use DSF as an approximate upper bound when comparing the performance of various NUMA-aware locks.

The following list summarizes the locks implemented and used in my experiments:

- Recoverable (All variables reside on PM):
  - **GH**: An implementation of the GH lock [36].
  - **JJJ**: An implementation of the JJJ lock [51].
  - **RF-GH**: RF lock with GH as the base lock.
  - **RF-JJJ**: RF lock with JJJ as the base lock.
  - **DSF-GH**: A dedicated scheduler that hands over a GH lock between nodes.
  - **DSF-JJJ**: A dedicated scheduler that hands over a JJJ lock between nodes.
- Non-recoverable (Some variables reside on PM while others reside on DRAM):
  - **Cohort-GH**: Cohort lock with GH local locks, one per node, and a GH global lock. Variables associated with the cohorting logic are stored on DRAM for performance, while the GH is stored on PM for fair comparison.

– **Cohort-JJJ**: Cohort lock with JJJ local locks, one per node, and a JJJ global lock. Variables associated with the cohorting logic are stored on DRAM for performance, while the JJJ is stored on PM for fair comparison.

The RF and cohort locks have the value HandoffThreshold set to 1024, while DSF locks have a **TimeThreshold** of 10 ms. All locks are implemented in C++ with `-O2` optimization level using gcc 9.3.0 on Ubuntu 20.04.1. I added flush instructions when implementing GH and JJJ to ensure correctness. A flush is implemented as a CLWB (`_mm_clwb`) compiler intrinsic followed by a memory store fence (`_mm_sfence`). PM is accessed using the PMDK 1.7 library through memory-mapped files.

The server used in my experiments is a multi-socket multi-core server with 4 NUMA nodes. Each node comprises a 2.1 GHz Intel Xeon 6230 processor with 20 physical cores (40 logical cores with hyper-threading). Moreover, the server has six 128GB Intel Optane modules per socket. In my experiments, one region of the PM is used in App-Direct mode. The memory-mapped file resides in one NUMA node, and I ensure that at least one thread is running on that node at any given time. Cache line write-back instructions are supported. Turbo boost is disabled to obtain consistent evaluation results.

4.5.1 Scalability

In practice, applications are expected to run free of failures most of the time. Hence, I consider failure-free runs of the locks under consideration[2] in my scalability tests. Flush instructions are used in my failure-free experiments to reflect the locks' practical performance. Each run involves multiple processes that continuously acquire and release a single lock for a specific time period. The performance metric used for evaluation is the total number of super-passages per second over a varying number of processes. Experiments engage up to 160 processes in multi-socket settings. Processes, or threads, are one-to-one mapped to processor cores. In multi-socket settings, I consider two ways of mapping threads upon initiation. The first mapping is (i) *round-robin*, where for every newly initiated thread, a node on which a core is allocated to that thread is picked in a round-robin fashion. The second mapping I adopt is (ii) *node-by-node*, where threads are mapped to cores that are located on the same NUMA node (initially from a starting node) until all the cores of that node are utilized, and then I map new threads to cores that are located on the next node,

[2]I tested recoverability by injecting crash failure code into my C++ implementation of GH, JJJ, RF-GH and RF-JJJ. I observed that the program reaches a stable state after performing crash failures in different locations, including the recovery, entry and exit sections.

and so on. Both mapping techniques map threads to all of the 80 physical cores first, then to the rest of the 80 hyper-threaded (logical) cores. Empirical results that show scalability are represented as graphs in which each point denotes the median of three consecutive runs of five seconds.

The first experiment considers a multi-socket configuration, where round-robin mapping of threads to sockets is used. Such settings reflect the performance under the case of utilizing cores from different nodes uniformly. From the outcomes illustrated in Figure 4.1, I observe that RF-GH and RF-JJJ boost the performance of GH and JJJ by 43% and 13% on average, respectively. Moreover, RF is lower than the non-recoverable cohort

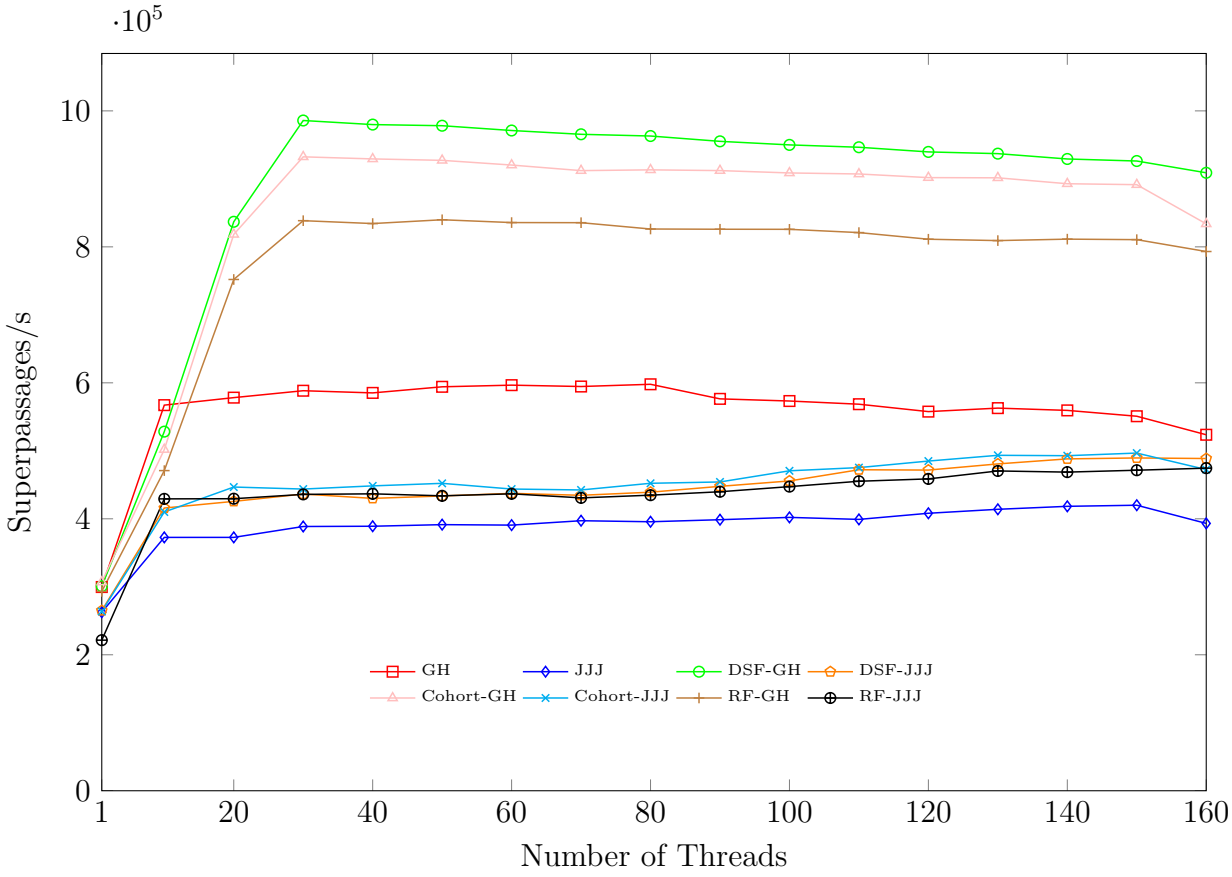


Figure 4.1: Throughput in multi-socket configuration using round-robin mapping.

in throughput by 9% and 3% on average when considering GH and JJJ, respectively. I believe there are two reasons for the cohort algorithm to result in higher throughput than the RF:

- Cohort locks are not recoverable, as opposed to RF locks. The variables associated with the cohort algorithm reside on the DRAM, while the RF variables reside on the Optane persistent memory.
- Flushing the caches in the RF algorithm, which is necessary to preserve recoverability, introduces overhead that does not exist in non-recoverable locks, like the cohort.

Comparing RF to DSF, RF-GH and RF-JJJ are lower in performance than their DSF variants by 13% and 1% on average, respectively. Moreover, when using GH as the base RME lock, the difference between the NUMA-aware algorithms is more significant than when using JJJ. The reason is that the slower entry and exit sections of the JJJ base lock become the performance bottleneck instead of the NUMA-aware algorithms. It is worth mentioning that DSF outperforms both the RF and cohort algorithms, at least when considering GH, making DSF performance a suitable upper bound for my micro-benchmark. This is expected as threads in the RF and cohort algorithms coordinate to hand over the lock between nodes, adding synchronization overhead. On the other hand, this operation is offloaded to a dedicated scheduler process in DSF.

My second experiment considers a multi-socket configuration, where node-by-node mapping of threads to nodes is used. The results, presented in Figure 4.2, show that using the RF algorithm boosts the performance of GH by 22%, 36% and 40% when using all the physical cores on two, three and four NUMA nodes, respectively, and by 45% when using all the 160 logical cores. Similarly, the RF algorithm increases the throughput of JJJ by 4%, 6% and 9% when using all the physical cores on two, three and four NUMA nodes, respectively, and by 21%[3] when using all the 160 logical cores. I observe that the performance benefit of using the RF lock increases as the number of utilized NUMA nodes increases. Such results show the scalability of my proposed black-box transformation technique. As for the other two NUMA-aware algorithms, the gap in throughput between the DSF and RF, and the Cohort and RF, algorithms when using node-by-node mapping (Figure 4.2) is similar to the gap of the same when using round-robin mapping (Figure 4.1). This suggests that the RF algorithm performs at least as well when randomly picking a node as when enforcing some specific mapping (i.e., node-by-node).

Another observation is the variation in performance of each algorithm when using a single socket, i.e., 1, 10 and 20 threads in Figure 4.2. DSF yields a minimal overhead, followed by the cohort algorithm and then the RF algorithm. It is important to assess the overhead of the proposed algorithm in the absence of cross-socket communication.

[3]I believe this could be an outlier as the performance boost gained from using RF-JJJ over JJJ is 11% when considering 150 logical cores.

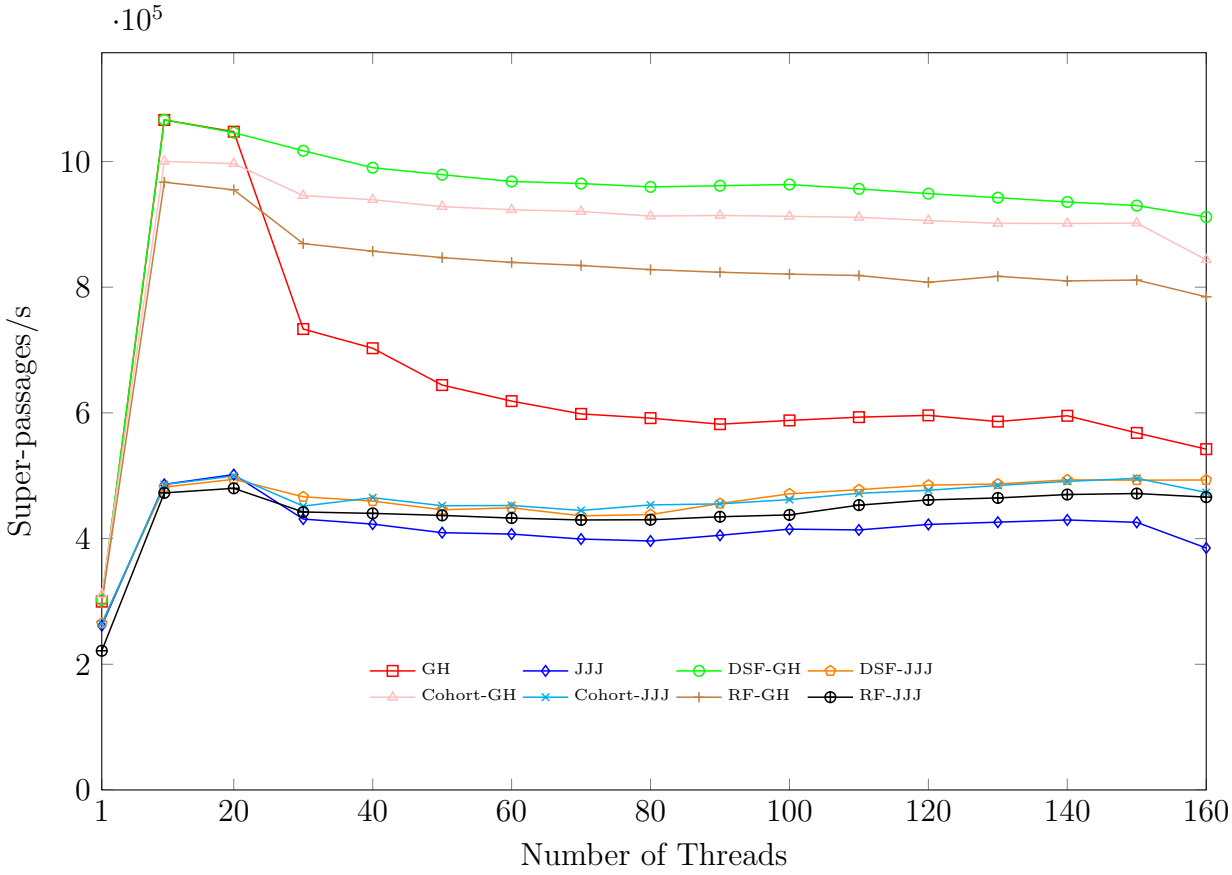


Figure 4.2: Throughput in multi-socket configuration using node-by-node mapping.

4.5.2 Single Socket Overhead

My next experiment is a comparison among the algorithms under consideration in a single-socket setting. In this experiment, all threads simultaneously run on the same NUMA node with no hyper-threading. Such an experiment investigates the overhead associated with adding NUMA-awareness to the NUMA-oblivious RME algorithms, GH and JJJ. The results, presented in Figure 4.3, show that DSF has minimal performance degradation[4] when considering both GH and JJJ. This suggests that DSF is closer than the RF and cohort algorithms to the ideal behavior of a NUMA-aware lock, supporting my claim that DSF is a reasonable upper bound estimate. In addition, from Figures 4.2 and 4.3, RF-GH

[4]In this experiment, the code is instrumented with time measuring functions (`std::chrono`) for further analysis, hence the slight difference in performance between the results in Figures 4.2 and 4.3.

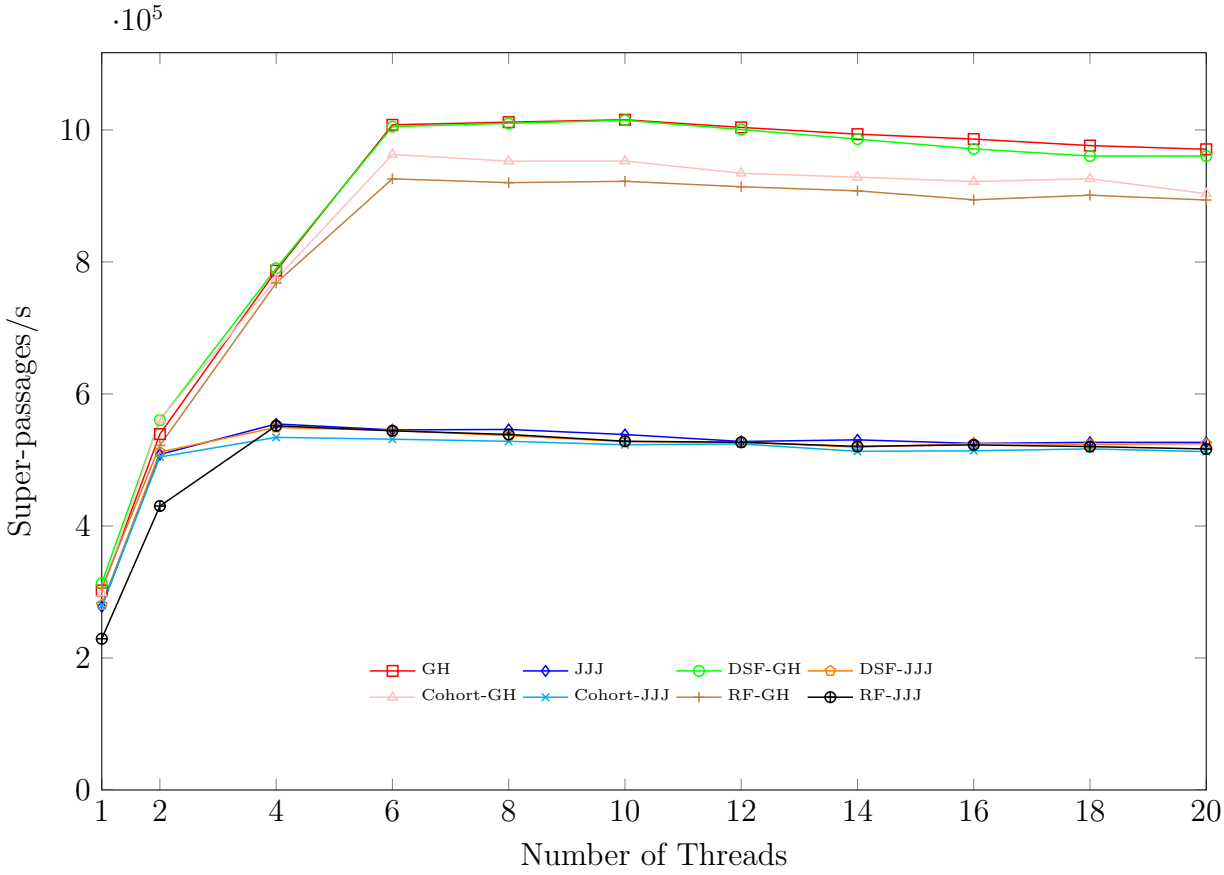


Figure 4.3: Throughput when using a single socket.

and RF-JJJ are shown to have a lower average overhead in single-socket settings when compared to GH and JJJ, respectively. That is, the performance gap between the RF locks and their potential upper bounds when using a single-socket (base lock or DSF) is lower than the performance gap (DSF) when using multi-socket configurations. The reason is that, in **FilterHandoff**, the first time the counter reaches HandoffThreshold, the *alone* flag is set. After that, all the subsequent calls of the same procedure are guaranteed to be a simple counter increment until a process in a different node, which does not exist when using a single NUMA node, performs the CAS in **Filter**. As a result, the overhead (of the RF transformation) associated with increasing the number of threads in a single-socket setting is less than the overhead in a multi-socket configuration.

4.5.3 Closer Look at Single Socket Overhead

I wanted to take a closer look at the result in Figure 4.3 to further examine the overhead of the proposed algorithm. Precisely, I believe that it is helpful to understand the impact of adding NUMA-awareness on each of the entry and exit sections of each algorithm. Accordingly, for each algorithm under study, I measured the average execution time per super-passage per thread in nanoseconds by using the high-resolution clock library `std::chrono`. The results for the GH and JJJ variants are presented in Figure 4.4. From the figure, I observe the following:

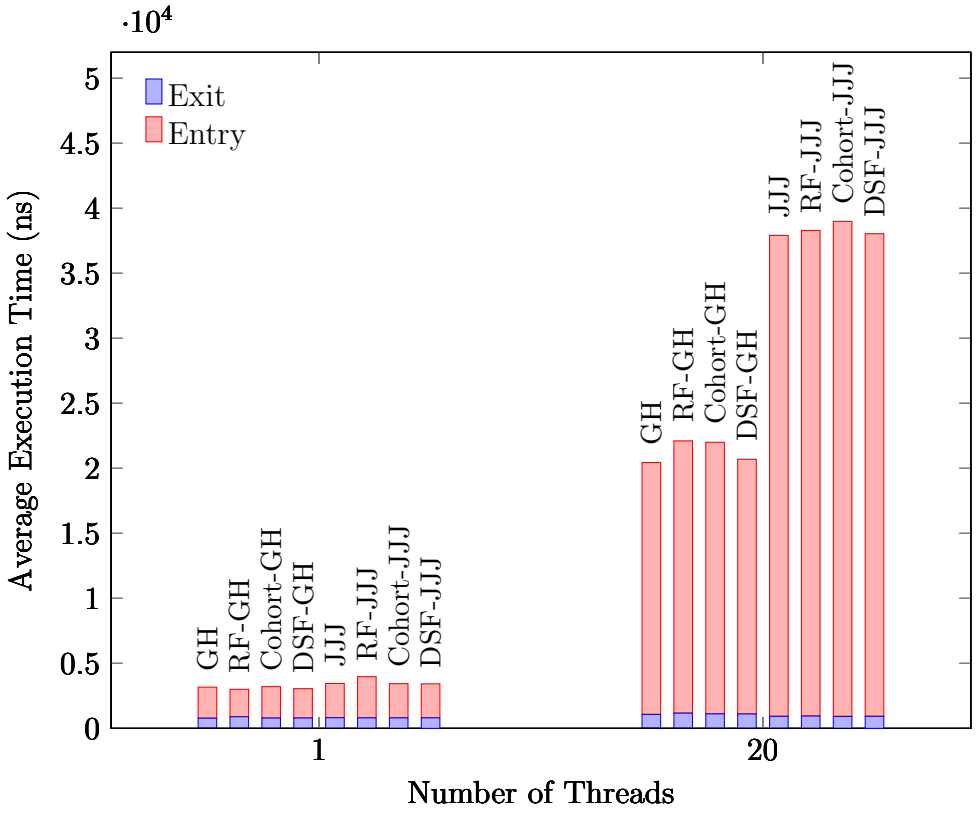


Figure 4.4: Average execution time per (super-)passage per thread of the entry and exit sections using a single socket.

- For all the locks, including the RF locks, increasing the number of threads does not significantly increase the execution time of the exit section.

- The execution time of the JJJ algorithms' entry section is significantly more sensitive to contention than the GH algorithms' entry section. As a result, the entry section of JJJ is the main reason for the dominance of GH in performance. That said, the exit section of JJJ is not a factor in its lower throughput when compared to GH.
- As opposed to GH, the exit section of JJJ is wait-free. However, I observe that the average execution time of the exit section of the two locks is not significantly different. This shows that a wait-free exit section is not necessarily faster on average.
- Considering 20 threads, the overhead of the NUMA-aware locks is more observable with GH (up to 8%) than with JJJ (up to 3%). Such an observation supports my hypothesis that the entry section of JJJ is the performance bottleneck that conceals the overhead of the NUMA-aware transformations.

In conclusion, this chapter introduced the(RF) lock as a novel solution to enhance the performance of RME locks in NUMA architectures. By transforming NUMA-oblivious RME algorithms into NUMA-aware versions, the proposed approach effectively leverage memory locality to achieve significant performance gains, with improvements of up to 45% in multi-socket configurations. Moreover, the RF lock preserves essential correctness properties, such as mutual exclusion and starvation-freedom, while minimizing overhead in single-socket setups. These results provide a robust answer to the first research question, offering a scalable and efficient method to improve the performance of RME locks on modern multicore systems.

Chapter 5

DULL: A Fast Scalable Detectable Unrolled Lock-Based Linked List

In this section of the thesis, I explore Q2 and Q3 as follows: First, I consider a fast lock-based data structure, the Platz' unrolled linked list [62], described in Section 2.7.1. Then, I exploit the mutual exclusion property of the underlying locks to achieve detectability by exclusively manipulating persistent shared variables. The result is DULL: a fast, scalable and Detectable Unrolled Lock-based Linked list. DULL is a concurrent implementation of the detectable sequential unordered set defined in Section 2.6.6. To the best of my knowledge, DULL is the fastest detectable lock-based, and the first detectable strictly-linearizable, linked list. The work presented in this section has been published in SPAA'24 [30] and in OPODIS'24 [31].

5.1 Persistent Objects

Before getting into the algorithmic details of DULL, three persistent objects are defined to be used as building blocks: *PMptr* is a pointer designed to work with *persistent memory mapped files*, *PMpair* is a tuple to store a key-value pair atomically in PM, and *descriptor* is an object that stores information related to the last operation completed by a specific process in a way that facilitates list detectability.

5.1.1 Persistent Pointers

The persistent pointer object is defined in Algorithm 5.1. A persistent pointer consists of

Variables:

- *offset*: **tuple** $\langle$**pointer**, **Boolean**$\rangle$, **init** $\langle 0, 0 \rangle$

Procedure SaP(*ptr*)

```
89 value := ptr − BaseAddress                          // Get the offset
90 offset := ⟨value, 1⟩          // Mark not persisted and store offset atomically
91 Flush(&offset)                                      // Persist to PM
92 CAS(&offset, ⟨value, 1⟩, ⟨value, 0⟩)                // Mark persisted
```

Procedure GaP

```
93 ⟨value, flag⟩ := offset                  // Read the offset atomically
94 if flag = 1 then                  // Check if offset is not persisted
95     Flush(&offset)                                  // Persist to PM
96     CAS(&offset, ⟨value, 1⟩, ⟨value, 0⟩)            // Mark persisted
97 return BaseAddress + value                          // Return the address
```

Algorithm 5.1: Variables and functions of a *PMptr* object

the variable *offset*, which is a tuple $\langle value, flag \rangle$ where *value* is an address offset from PM base address, and *flag* is 1 if *offset* is persisted and 0 otherwise.

A persistent pointer, PMptr, handles two problems: *address space layout randomization* (ASLR) [44] and efficiently persisting pointers. As a result of ASLR, de-referencing a pointer on PM that stores the address of a persistent object after recovering from a system failure might result in accessing an invalid memory location. The PMptr compensates for ASLR by storing each pointer as an offset from the base address of the persistent memory-mapped file and calculating the pointer value on every access. The offset is the location of the persistent object, referenced by the pointer, relative to the base address of the memory-mapped file. BaseAddress is initialized every time the program is (re)started to ensure valid referencing when using PMptr pointers.

Algorithm 5.1 also handles persisting PMptr pointers. In some scenarios, values read from or written to persistent variables must be flushed before being used. Consider the

following example: Assume there exists a concurrent singly-linked list that realizes an unordered set, and that the set only contains the value 5 at the head node. Consider two concurrent operations **Remove**(5) and **Contains**(5). Suppose that the former changed the head pointer of the list to $\perp$, then the latter dereferenced the head pointer and returned false, indicating that 5 is removed from the list. Now, if the system crashes before **Remove**(5) persists the value ($\perp$) of the head pointer, then, after recovery, a subsequent **Contains**(5) would return True which contradicts that 5 has been removed from the set. A possible solution is to persist the head pointer after reading it in **Contains**(5) before returning.

Persisting the offset upon every read and write is a conservative approach to ensure correctness. However, this might be inefficient since each update needs to be flushed only once to PM for a subsequent recovery. PMptr persists the offset more efficiently. The idea is borrowed from the work of Wang et al. in [71]. The *offset* variable is a tuple of the offset value intended to be persisted and a "dirty bit" that indicates whether the current offset value has ever been persisted (0) or not (1). As shown in Algorithm 5.1, a PMptr object can be accessed through two operations: *Set and Persist* (**SaP**) and *Get and Persist* (**GaP**).

The function **SaP** takes a pointer to a persistent object (*ptr*) and calculates the offset value (*value*) at Line 89. Then, *offset* is set to $\langle value, 1\rangle$, setting its value to the calculated offset and marking the offset as not persisted. The **SaP** operation flushes *offset* (Line 91) and then attempts to mark it as persisted by resetting the dirty bit to 0 through a **CAS** operation (Line 92). The **CAS** at Line 92 is required to avoid erroneously overwriting a value set by another concurrent **SaP** operation.

The **GaP** operation reads the tuple *offset* (Line 93) and checks whether it was flushed or not by examining the dirty bit (Line 94). If the dirty bit is set to 1, then *offset* is persisted at Line 95, and the dirty bit is reset to 0 through a **CAS** operation at Line 96. Similar to **SaP**, the **CAS** at Line 96 is needed to prevent potential incorrect changes to updates occurred by a concurrent **SaP** operation.

5.1.2 Persistent Pair

The persistent key-value pair object is defined in Algorithm 5.2. A persistent key-value pair, PMpair, comprises a single variable and three associated functions. The variable *pair* is a tuple $\langle key, value, flag\rangle$, where *key* and *value* are variables that store a key and its corresponding data, respectively, and *flag* is a dirty bit set to 1 if *pair* is not persisted and 0 otherwise.

Variables:

- *pair*: **tuple** $\langle key, value, \textbf{Boolean} \rangle$, **init** $\langle \perp, \perp, 0 \rangle$.

Procedure Set$(\langle key, value \rangle)$

```
98  pair := ⟨key, value, 1⟩        // Store the key-value pair and mark not persisted
```

Procedure Get

```
99  ⟨key, value, _⟩ := pair        // Read key-value pair
100 return ⟨key, value⟩
```

Procedure Persist

```
101 ⟨key, value, flag⟩ := pair      // Read the pair atomically
102 if flag = 1 then                // Check if the pair is not persisted
103     Flush (&pair)               // Persist to PM
104     CAS (&pair, ⟨key, value, 1⟩, ⟨key, value, 0⟩)   // Mark
```

Algorithm 5.2: Variables and functions of a *PMpair* object

Accessing *pair* is governed by three functions: **Set**, **Get** and **Persist**. The function **Set** stores a key-value pair, given as an argument, into *pair* and marks the tuple as not persisted. The key and value stored in *pair* are retrieved by the procedure **Get**, ignoring the dirty bit state. Neither function flushes the key-value pair object. Persisting *pair* is delegated to a separate procedure, **Persist**, for performance purposes (more details in Section 5.3). This function reads the tuple *pair* (Line 101) and checks the dirty bit (Line 102). If the dirty bit is set to 1, then *pair* is not persisted, and the function flushes *pair* to PM at Line 103. Finally, *pair* is marked as persisted by resetting the dirty bit to 0 through a **CAS** at Line 104. The **CAS** primitive at Line 104 is used for the same reason as at Lines 92 and 96.

5.1.3 Descriptor

A descriptor object is defined in Algorithm 5.3. To achieve detectability, each process accessing an instance of *DULL* is associated with a descriptor object. This descriptor is exclusively accessible by the process during normal operation and by a single process during the recovery phase after a system crash. At the beginning of a detectable operation, a process shall store some operation-related information in its descriptor, i.e., *prepare* the

Define Unit:

- *op*: **int** $\in \{\text{INSERT}, \text{REMOVE}, \perp\}$, **init** $\perp$
- *args*: **tuple**$\langle key, value\rangle$, **init** $\langle\perp, \perp\rangle$
- *tag*: **int**, **init** $\perp$
- *result*: **int** $\in \{$**true**, **false**, $\perp\}$, **init** $\perp$
- *offset*: **pointer** of a node, **init** 0

Variables:

- *units*$[0, 1]$: **array** of **struct** Unit
- *curr*: **Boolean**, **init** 0

Procedure SetResult(*res*)

```
105 units[curr].result := res                          // Set result of current op
106 Flush(&units[curr].result)                         // Persist to PM
```

Procedure SetNode(*ptr*)

```
107 units[curr].offset := ptr − BaseAddress
108 Flush(&units[curr].offset)                         // Persist to PM
```

Procedure Prep-*op*(*key*, *value*, *tag*)

```
109 next := 1 − curr
110 units[next].op := op                               // initialize next unit
111 units[next].args := ⟨key, value⟩
112 units[next].tag := tag
113 units[next].result := ⊥
114 units[next].offset := 0
115 Flush(&units[next])                                // Persist next unit to PM
116 curr := next                                       // make next unit the current one
117 Flush(&curr)                                       // Persist curr to PM
```

Procedure Resolve

```
118 op := units[curr].op
119 ⟨key, value⟩ := units[curr].args
120 tag := units[curr].tag
121 result := units[curr].result
122 pred := units[curr].offset + BaseAddress
123 return (op, key, value, tag, result, offset)      // return resolved state
```

Algorithm 5.3: Variables and functions of a *descriptor* object

operation, through the function **Prep-***op*. Preparing an operation is designed such that all changes to the descriptor variables *appear to be atomic*. That is, either all changes take effect or none at all. To achieve atomicity, each descriptor object consists of two instances of its variables, called *units*, and a Boolean variable, *curr*, that points to the *current unit*, where the other instance is called the *next unit*.

In **Prep-***op*, the next unit variables are initialized with the appropriate parameters (Lines 110-114), then *curr* is flipped to switch the next and current units (Line 116). Note that, all the descriptor variables in the next unit are persisted to PM at Lines 115 and 117 before the next unit becomes the current one. The preparation process is completed only if *curr* is flushed. If a crash occurs before persisting *curr*, then re-invoking the function **Prep-***op* upon recovery repeats the whole procedure, making it failure-atomic. The function **SetNode** is used for check pointing (more in Section 5.3.3), and **SetResult** marks the end of the prepared operation (*op*) by storing its response in the descriptor. Furthermore, a process can resolve its last successful operation at any point in time by invoking the idempotent **Resolve** procedure of its descriptor object. If the system fails before completing *op*, i.e., setting and flushing *result*, the incomplete operation will be discarded by changing the Boolean *curr* to point to the (next) unit of the last completed operation (more details in Section 5.3.6).

5.2 Linked List Structure

In designing DULL, persistent unrolled nodes are used as building blocks. Node unrolling, initially proposed by Shao et al. [67], involves the storage of multiple elements in an array within each node. My detectable implementation is based on Platz et al. unrolled linked list [62], presented in Section 2.7.1. Each node comprises the following variables:

- *count*: the number of key-value pairs stored in the node.
- *marked*: a Boolean indicating whether the node has been logically removed from the list.
- *m*: a volatile mutex lock used only during insertions and removals in the list.
- *anchor*: the key that uniquely characterizes a node such that if a key exists in a node, then the key $\geq$ the *anchor* of that node and $<$ the *anchor* of the successor node. Nodes are thus sorted in ascending order by their anchor keys.
- *next*: a persistent pointer (PMptr) to the successor node.

- *pairs*$[0..K-1]$: an array of up to K unordered variables of type PMpair, where K is the maximum number of pairs in a node.

In addition to persistent nodes, a DULL instance has an array $D[1..N]$ of N descriptor objects, where N is the number of processes. Moreover, a DULL object has at least three persistent nodes at all time: the head and tail sentinel nodes and one non-sentinel node. Sentinel nodes do not store key-value pairs. The head and the non-sentinel node anchor keys are $-\infty$, and the tail anchor key is ∞, where $-\infty$ is some key $<$ the minimum key in the key space, and ∞ is some key $>$ the maximum key in the key space.

The procedure **PersistNode**, in Algorithm 5.4, defines how a node is persisted to PM. Persisting a variable can be achieved by writing its cache line back to memory, followed by a store fence. Minimizing the number of memory fences to one per procedure call reduces synchronization and improves performance. Moreover, Algorithm 5.4 only persists the *anchor* and array *pairs* to PM. After a crash failure, some variable values can be inferred from other persisted objects upon recovery. For example, the value of *count* is derived by counting valid pairs (i.e., $\neq \langle \perp, \perp \rangle$) in the recovered array *pairs*. Other variables are reset, such as m (more details in Section 5.3.6). Futhermore, persisting node variables through Algorithm 5.4 only occurs locally by a thread upon initializing a node prior to insertion. Therefore, failure-atomicity is not necessary for the **PersistNode** procedure.

Procedure PersistNode

```
124 CLWB(&anchor)                          // Cache-line write-back anchor key
125 for i := 0 to K − 1 do                 // Cache-line write-back each kv pair
126     ⟨key, value⟩ := pairs[i].Get()
127     pairs[i].pair := ⟨key, value, 0⟩
128     CLWB(&pairs[i].pair)
129 SFENCE
```

Algorithm 5.4: Persist node to PM

5.3 Implementation Details

This section presents the implementation details of *DULL*. For simplicity, the operations **Detectable-Insert** and **Detectable-Remove** (Figure 2.4) will be referred to as **Insert** and **Remove** throughout this section. A full description of the DULL algorithm, including

the overlap with the algorithm in [62] is presented for completeness. In search, insertion, and removal operations, I highlight the lines of added or modified steps in red and underline corresponding line numbers to distinguish my contribution from the original work. The remaining operations form essential components of my contribution.

5.3.1 Memory Layout

The memory mapped file is organized as follows: The first 8 bytes are reserved for NIL pointer. Subsequently, N descriptor objects—one per process—are allocated. The next 2x node-size bytes are used for the head and tail nodes. The rest of the memory-mapped file is utilized as a pool for node allocation. The memory layout is illustrated in Figure 5.1. I implement and use a simple recoverable bitmap fixed-size memory allocator.

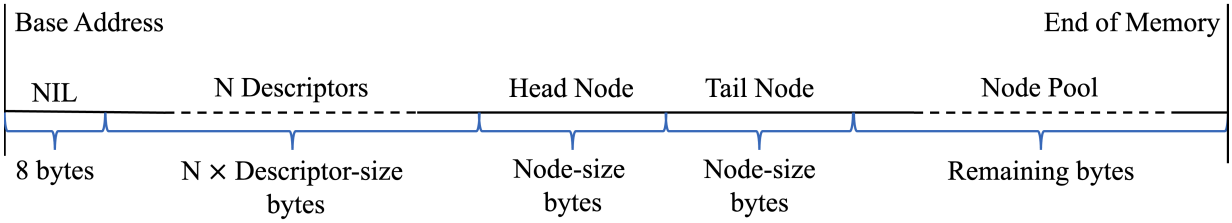


Figure 5.1: Memory layout for DULL.

5.3.2 Traversal and Search

Traversing in DULL is separated into two functions: **Scan** and **Seek**. Both procedures are presented in Algorithm 5.5. The function **Scan** searches the list for a target *key* starting from the first non-sentinel node (Line 131), while tracking the predecessor, *pred*, and successor, *succ*, of the current, *curr*, node. The search ends by reaching a node, *curr*, where the target key might exist, i.e., the successor's anchor key is greater than the target key (Line 133). Finally, the function **Scan** returns the three tracked pointers at Line 137. The while-loop at Line 133 is guaranteed to terminate since the tail's anchor key is greater than all keys in the key space. The procedure **Seek** takes a pointer to some node *curr* and a target key, and it linearly searches for the key in the array *pairs* of the node *curr* (Line 138). If the target key is found, the function returns the index of its key-value pair at Line 142; Otherwise, it returns $\perp$ at Line 143, indicating that the key is absent in *curr*.

Procedure Scan(*key*)

```
130 pred := head.GaP()                                  // Skip head
131 curr := pred.next.GaP()
132 succ := curr.next.GaP()
133 while succ.anchor ≤ key do                          // find suc's anchor > key
134     pred := curr
135     curr := succ
136     succ := succ.next.GaP()
137 return (pred, curr, succ)
```

Procedure Seek(*curr*, *key*)

```
138 for j := 0 to K − 1 do                               // For each pair
139     ⟨k, v⟩ := curr.pairs[j].Get()                   // Read pair atomically
140     curr.pairs[j].Persist()                         // Persist pair to PM
141     if k = key then
142         return j                                    // Return pair index if found
143 return ⊥                                            // Otherwise return ⊥
```

Algorithm 5.5: List traversal

A **Contains** operation, presented in Algorithm 5.6, searches for the node, *curr*, that potentially contains the argument target key by invoking the procedure **Scan** at Line 144. The array *pairs* of the node *curr* is searched for the target key by calling the function **Seek** at Line 145. If the key is found, **Seek** returns a valid index (Line 146) of the corresponding pair, which is atomically read and then persisted at Lines 147 and 148, respectively. The key of the read pair is checked at Line 149 before returning its value at Line 150. If **Seek** does not find *key* and returns $\perp$, or the check at Line 149 fails, the operation returns $\perp$ at Line 151, indicating that the list does not contain the target key. Note that all traversed pointers are flushed at least once through the function **GaP** at Line 136, and each pair read is persisted through the **Persist** function at Line 140. Additionally, the function **Contains** persists the found pair (Line 148) after the atomic read at Line 147. All persistence operations occur before returning from the corresponding procedures to preserve strict linearizability (more details in Section 5.4).

```
Procedure Contains(key)
144 (_, curr, _) := Scan(key)                          // Get the containing node
145 j := Seek(curr, key)                               // Get the index of key in curr
146 if j ≠ ⊥ then                                      // If key is found
147     ⟨k, v⟩ := curr.pairs[j].Get()                  // Read pair atomically
148     curr.pairs[j].Persist()                        // Persist pair to PM
149     if k = key then                                // Return value if key still in pairs
150         return v
151 return ⊥                                           // Otherwise return ⊥
```

Algorithm 5.6: Searches the node that contains a target key

5.3.3 Insertion

Given a pair, *key* and *value*, and some *tag*, an **Insert** operation, shown in Algorithm 5.7, calls **Prep-Insert** of the executing process' descriptor to initialize it at Line 152. Subsequently, the operation finds the appropriate node (*curr*) to store the pair, and its predecessor (*pred*), by calling the function **Scan** at Line 154. The predecessor lock is acquired at Line 155 to prevent any simultaneous update to *curr*, *pred* and the linking pointer, *pred.next*. The validation in [40] is then applied to check that *pred.next* still points to *curr* and that neither node is marked as removed (Line 156). If the validation fails, then the predecessor's mutex is released and the insertion operation is restarted. Otherwise, Line 159 calls the function **Seek**, searching for *key* in *curr*'s array. If the returned index (j) from **Seek** is valid, then the function marks the insert operation as completed, unlocks the predecessor's lock and returns false (Lines 161-163), indicating that the key already exists.[1] Otherwise, an empty slot in the array *pairs* is searched for by the function **Seek** at Line 165. Consequently, there are two possible scenarios:

1. There is an empty slot in *pairs* and **Seek** returns its index, or
2. the array is full and **Seek** returns $\perp$.

If 1, then the empty slot in *pairs* is set to *key* and *value* and is persisted at Lines 167 and 168, respectively, and the counter of *curr* is incremented at Line 169. If 2, then *curr* is split into two nodes, new_1 and new_2, using the helper procedure **Split**, called at Line 172.

[1] Instead, the value in a found pair can be updated and the algorithm remains correct.

Subsequently, the new pair is inserted into one of the two new replacement nodes. Note that if the target key is not found at Line 159, it will eventually be inserted regardless of whether case 1 or 2 occurs. To maintain consistency in the presence of a crash failure, the pointer *pred* is stored in the descriptor $D[i]$ as a checkpoint marker at Line 164. That is, if a process p_i crashes after persisting $D[i].offset$ (Line 108), the insertion operation can be *cleaned up* during the recovery phase (Section 5.3.6).

Algorithm 5.8 defines the subroutine **Split**. This function creates two new nodes to replace *curr*, where each stores roughly half of the pairs in *curr*. The procedure starts by creating a local copy, *tmp*, of the *pairs* array in *curr* at Line 187 and sorting it by key at Line 188. New nodes, new_1 and new_2, are allocated (Lines 189 and 190), and the smallest share ($\lfloor K/2 \rfloor$) of the pairs in *tmp* are copied to new_1, and the rest of *tmp* is copied to new_2 (Lines 191-196). The anchor key of new_1 is set to the anchor key of *curr* at Line 197. The anchor key of new_2 is set to the key of the first pair (Line 200), which has the smallest key in new_2 since the node's array is sorted by key. The counter of each new node is set to the corresponding number of copied pairs from *curr* at Lines 198 and 201. Finally, the next pointers of new_1 and new_2 are set to new_2 and the successor of *curr* at Lines 202 and 203, respectively, before returning to **Insert**.

Subsequently, the given pair is inserted into new_1 if the pair's key is strictly less than the anchor key of new_2. Otherwise, the pair is inserted into new_2. In both cases, the counter of the node where the pair is inserted is incremented. Finally, *curr* is marked to prevent erroneous subsequent updates to the removed node (Line 182), the locks in *curr* and *pred* are released (Lines 183 and 185), and the value **true** is returned (Line 186) to indicate that the pair is successfully inserted into the list. Before the procedure **Insert** returns, p_i marks its operation as completed by setting the result field of its descriptor object through calling **SetResult** at Line 184.

To ensure recoverability, the next pointers of the new replacement nodes are persisted in the function **Split** (Lines 202 and 203). The other variables of the new nodes are persisted by invoking the function **PersistNode**, Algorithm 5.4, for both nodes (Lines 179 and 180) before they are inserted and become reachable by other threads (Line 181). Strict linearizability is maintained in the following manner: If the system fails before flushing the value written in $curr.pairs[j]$.**Set** at Line 167, or before replacing *curr* with the new nodes (Line 181) during the node split, the incomplete operation will be cleaned up upon recovery, as it has not yet taken effect. However, if the failure occurs after Line 168 or 181, then the operation is considered to have taken effect before the crash, and the inserted pair will be recovered (more in Section 5.3.6).

```
Procedure Insert(key, value, tag)
152 D[i].Prep-Insert(key, value, tag)
153 while true do
154     (pred, curr, _) := Scan(key)                          // Get the containing node
155     pred.m.Lock()                                         // Lock the predecessor node
        // Validate curr is the pred of next and that both are not deleted
156     if pred.marked ∨ curr.marked ∨ pred.next.GaP() ≠ curr then   // If not valid
157         pred.m.Unlock()                                   // Unlock predecessor and retry
158         continue
159     j := Seek(curr, key)                                  // Get the index of key in curr
160     if j ≠ ⊥ then                                         // If the key is found
161         D[i].SetResult(false)                             // Mark op as completed
162         pred.m.Unlock()                                   // unlock predecessor and return
163         return false
164     D[i].SetNode(pred)                                    // Set pred in descriptor
165     j := Seek(curr, ⊥)                                    // Get the index of empty slot in curr
166     if j ≠ ⊥ then                                         // If found
167         curr.pairs[j].Set(key, value)                     // Set key-value pair
168         curr.pairs[j].Persist()                           // Persist key-value pair
169         curr.count := curr.count + 1                      // Advance counter
170     else
171         curr.m.Lock()                                     // If curr is full, lock curr
172         (new1, new2) := Split(curr)                       // Split into new1, new2
            // Insert key-value pair into the appropriate node
173         if key < new2.anchor then
174             new1.pairs[new1.count].Set(key, value)
175             new1.count := new1.count + 1
176         else
177             new2.pairs[new2.count].Set(key, value)
178             new2.count := new2.count + 1
179         new1.PersistNode()                                // Persist new nodes
180         new2.PersistNode()
181         pred.next.SaP(new1)                               // Logically insert new nodes
182         curr.marked := true                               // Mark curr as removed
183         curr.m.Unlock()
184     D[i].SetResult(true)                                  // Mark op as completed
185     pred.m.Unlock()
186     return true
```

Algorithm 5.7: Key-value pair insertion.

Procedure Split($curr$)

```
187 tmp := curr.pairs                                   // Get local copy of the array pairs
188 Sort(tmp)                                           // Sort pairs by key
189 new1 := new Node                                    // Create two new nodes
190 new2 := new Node
    // Copy half pairs to new1 and the rest to new2
191 for j := 0 to K − 1 do
192     ⟨k, v⟩ := tmp[j].Get()
193     if j ≤ ⌊K/2⌋ then
194         new1.pairs[j].Set(k, v)
195     else
196         new2.pairs[j].Set(k, v)
197 new1.anchor := curr.anchor                          // Set other node variables
198 new1.count := ⌈curr.count/2⌉
199 ⟨kmin, _⟩ := new2.pairs[0].Get()
200 new2.anchor := kmin
201 new2.count := ⌊curr.count/2⌋
202 new1.next.SaP(new2)                                // Link and persist the next pointers
203 new2.next.SaP(curr.next.GaP())
204 return (new1, new2)
```

Algorithm 5.8: Helper function **Split**

5.3.4 Removal

Algorithm 5.9 shows the procedure of removing a key-value pair from a DULL instance. First, the descriptor of the executing process is initialized at Line 205. Given a target *key*, invoking **Remove** finds the node (*curr*), and its predecessor (*pred*), that potentially stores the key via the function **Scan** at Line 207. The lock in *pred* is acquired at Line 208 to prevent any simultaneous update to the *curr* and *pred* nodes. The algorithm validates that the next pointer or *pred* still points to *curr* and that both nodes are not marked as removed (Line 209). If the validation fails, then the mutex of *pred* is unlocked and the removal operation is restarted. If the validation succeeds, the function **Seek** searches for the target key at Line 212. If the key is not found, then the function marks the operation as completed, unlocks the lock of *pred* and returns **false** (Lines 214-216). Otherwise, p_i stores the offset of *pred* in its descriptor object at Line 217 to detect and optionally resume the operation

Procedure Remove $(key,\ tag)$

```
205 D[i].Prep-Remove(key, ⊥, tag)
206 while true do
207     (pred, curr, _) := Scan(key)
208     pred.m.Lock()
209     if pred.marked ∨ curr.marked ∨ pred.next.GaP() ≠ curr then
210         pred.m.Unlock()                          // Validate curr is pred next and are not deleted
211         continue
212     j := Seek(curr, key)
213     if j = ⊥ then
214         D[i].SetResult(false)                                  // mark op as completed
215         pred.m.Unlock()
216         return false
217     D[i].SetNode(pred)                                         // Set pred in descriptor
218     curr.pairs[j].Set(⊥, ⊥)                                    // Remove pair and persist
219     curr.pairs[j].Persist
220     curr.count := curr.count − 1
221     if curr.count < MinFull then
222         curr.m.Lock()
223         succ := curr.next.GaP()
224         if curr.count = 0 ∧ curr.anchor ≠ −∞ then
225             pred.next.SaP(succ)          // Remove current node if empty and not last non-sentinel
226             curr.marked := true
227         else if succ.anchor ≠ +∞ then
228             succ.m.Lock()                                  // otherwise, merge or redistribute
229             if curr.count + succ.count < MaxMerge then
230                 new1 := Merge(curr, succ)
231             else
232                 (new1, new2) := Redistribute(curr, succ)
233                 new2.PersistNode()
234             new1.PersistNode()
235             pred.next.SaP(new1)                                // Logically insert new node
236             curr.marked := true                          // Mark curr and succ as removed
237             succ.marked := true
238             succ.m.Unlock()
239         curr.m.Unlock()
240     D[i].SetResult(true)                                       // mark op as completed
241     pred.m.Unlock()
242     return true
```

Algorithm 5.9: Key-value pair removal.

upon recovery (Section 5.3.6). Then, the pair containing the target key is removed and persisted at Lines 218 and 219, respectively, and the node's counter is decremented at Line 220. If the counter is less than a predefined constant (MinFull $= \lfloor K/4 \rfloor$), some actions are taken to reduce the variation in the number of pairs across all nodes. In the general case where *curr* is not the only non-sentinel node and its successor (*succ*) is not the tail, there are three possible scenarios:

1. *curr* is empty,
2. *curr* is not empty, and the sum of the counters of *curr* and *succ* is less than some predefined constant (MaxMerge $= \lfloor 3 \times K/4 \rfloor$), or
3. *curr* is not empty, and the sum of the counters of *curr* and *succ* is greater than or equal to MaxMerge.

If 1, then *curr* is logically removed (Line 225) from the list and marked to prevent invalid subsequent changes to the node. If 2, then *curr* and *succ* are merged into a new node by invoking the helper function **Merge** at Line 230. If 3, then the pairs stored in *curr* and *succ* are redistributed into two new replacement nodes, by calling the helper procedure **Redistribute** at Line 232, such that both nodes contain roughly the same number of pairs. The steps for the helper subroutines **Merge** and **Redistribute** are presented in Algorithms 5.10 and 5.11, respectively.

In cases 2 and 3, the new nodes are persisted and logically inserted into the list (Lines 233-235), and *curr* and *succ* are marked at Lines 236 and 237, respectively. Finally, at Lines 238-242, all acquired locks are released, the operation is marked as completed, and the **Remove** procedure returns true, indicating that the pair was successfully deleted.

In Algorithm 5.10, merging *curr* and *succ* involves creating a new node at Line 243 and copying all valid pairs from the *curr* and *succ* nodes to the newly created node, new_1, at Lines 245-253. The *anchor* of new_1 is set to *curr*'s anchor key (Line 254) as it is less than or equal to all keys in the *curr* and *succ* nodes. The counter of the new node is the sum of counters of the *curr* and *succ* (Line 255). The *next* pointer of new_1 is set to *succ.next* at Line 256 to replace both merged nodes, *curr* and *succ*, when *pred*'s next pointer is set to the new node, new_1, at Line 235 of **Remove**.

If merging the *curr* and *succ* nodes is not possible (Line 229), the pairs stored in these two nodes are roughly evenly redistributed across them by calling the helper function **Redistribute**, shown in Algorithm 5.11. The helper subroutine replaces a *curr* and *succ* with two new nodes, new_1 and new_2, respectively. To evenly redistribute pairs among

Procedure Merge ($curr$, $succ$)

```
243 new_1 := new Node                                          // Create a new node
244 j := 0                                                     // Copy valid pairs from curr and succ to new_1
245 for i := 0 to K − 1 do
246     ⟨k_curr, v_curr⟩ := curr.pairs[i].Get()
247     ⟨k_succ, v_succ⟩ := succ.pairs[i].Get()
248     if k_curr ≠ ⊥ then
249         new_1.pairs[j].Set(k_curr, v_curr)
250         j := j + 1
251     if k_succ ≠ ⊥ then
252         new_1.pairs[j].Set(k_succ, v_succ)
253         j := j + 1
254 new_1.anchor := curr.anchor                                // Set other node variables
255 new_1.count := curr.count + succ.count
256 new_1.next.SaP(succ.GaP())
257 return new_1
```

Algorithm 5.10: Helper function **Merge**

the two nodes, each node should store roughly $(curr.count + succ.count)/2$ pairs. From Line 221, the pair count in $curr$ is less than MinFull. Additionally, from Line 229, the combined pair count of $curr$ and $succ$ is greater than or equal to MaxMerge $= 3 \times$ MinFull, which implies that $succ.count \geq 2 \times$ MinFull $> curr.count$. Since $curr$ has fewer pairs, $M = (\lceil curr.count + succ.count/2 \rceil - curr.count)$ pairs should be moved from $succ$ to $curr$ (Line 265). The procedure **Redistribute** copies all pairs stored in $curr$ to new_1 in Lines 260-264. Subsequently, the function copies the smallest M pairs from $succ$ to new_1 and the rest of pairs to new_2 at Lines 265-273. The anchor key of new_1 is set to $curr$'s anchor key (Line 274) as it is less than or equal to all keys in $curr$ and $succ$. The keys in new_2 are sorted since they are copied in order from the sorted array tmp. Hence, the anchor key of new_2 is set to the key of its first pair, which is the smallest key in new_2, at Line 277. The counter of new_1 is set to the pair count of $curr$ plus the number of pairs copied from $succ$ (M) at Line 275, and the counter of new_2 is set to the number of the remaining pairs ($succ.count - M$) at Line 278. Finally, the function **Redistribute** links the new nodes together by setting the new_1's next pointer to new_2 (Line 279), and connects the two nodes to the list by setting the new_2's next pointer to the next node of $succ$ (Line 280). The next pointers of the two new replacing nodes are persisted by the **SaP** function,

Procedure Redistribute $(curr,\ succ)$

```
258 new_1 := new Node                                          // Create two new nodes
259 new_2 := new Node
260 for i := 0 to K − 1 do                                     // Copy valid pairs from curr to new_1
261     ⟨k, v⟩ := curr.pairs[i].Get()
262     if k ≠ ⊥ then
263         new_1.pairs[new_1.count].Set(k, v)
264         new_1.count := new_1.count + 1
    // Number of pairs to move from succ to curr
265 M := ⌈(curr.count + succ.count)/2⌉ − curr.count
266 tmp := succ.pairs                                         // Get local copy of the array pairs
267 Sort (tmp)                                                // Sort pairs by key
268 for i := 0 to M − 1 do
        // Copy the first M pairs in tmp to new_1 and the rest to new_2
269     ⟨k, v⟩ := tmp[i].Get()
270     new_1.pairs[curr.count + i].Set(k, v)
271 for i := M to succ.count − 1 do
272     ⟨k, v⟩ := tmp[i].Get()
273     new_2.pairs[i − M].Set(k, v)
274 new_1.anchor := curr.anchor                                // Set other node variables
275 new_1.count := curr.count + M
276 ⟨k_min, _⟩ := new_2.pairs[0].Get()
277 new_2.anchor := k_min
278 new_2.count := succ.count − M
279 new_1.next.SaP(new_2)                                     // Link and persist the next pointers
280 new_2.next.SaP(curr.GaP())
281 return (new_1, new_2)
```

Algorithm 5.11: Helper function **Redistribute**

before they are returned to and used in the **Remove** procedure.

To guarantee the recoverability of the **Remove** operation, the next pointers of the newly created replacement nodes are persisted within the **Merge** and **Redistribute** functions (see Lines 256 and 279-280). The remaining variables of these new nodes are persisted by invoking the **PersistNode** function (Algorithm 5.4) for each new node (Lines 234 and

233) before they are inserted and made accessible to other threads (Line 235). Strict linearizability is maintained as follows: If the system crashes before flushing the value written by *curr.pairs*[*j*].**Set** (Line 218), the incomplete operation will be cleaned up upon recovery, as it has not yet taken effect. If the crash occurs after Line 219, the pair removal has already taken effect before the failure. In this case, the recovery protocol (Section 5.3.6) will handle any subsequent node merging, redistribution, or removal as needed.

5.3.5 Resolve

The resolve function, presented in Algorithm 5.12, simply returns the last successful operation's signature, tag and result from the descriptor object. A process can resolve its last completed operation at any point in time by invoking the idempotent **Resolve**.

Procedure Resolve(i)

282 $\langle op, key, value, tag, result, _ \rangle := D[i].\textbf{Resolve}()$
283 **return** ($op, key, value, tag, result$)

Algorithm 5.12: Resolution of the last successful operation.

5.3.6 Recovery

The recovery protocol, defined in Algorithm 5.13, is a procedure that is invoked, by a single thread, after a crash failure (for more details regarding the crash-recovery failure model, refer to Section 2.6.3). First, D, *head* and *tail* are initialized to the addresses of the descriptors array, head node and tail node (Lines 284-286), respectively (refer to Section 5.3.1 for details about the memory mapped file layout). Then, the recovered head and tail are used to traverse the list (Line 288) and recover the variables of each node.

Adding persistent nodes to the list—which may occur when inserting or removing a key—comprises creating new nodes and persisting their *anchor*, *pairs* and *next* variables before the new nodes are logically inserted into the list. Throughout a node life-cycle, *anchor* does not change after it is first initialized. As for the *next* pointers, traversing the list (Section 5.3.2) is designed such that the value of a *next* pointer is persisted by the **GaP** function upon reading. In addition, each element in the array *pairs* is persisted on update. As a result, a node is simply recovered by resetting its mutex (Line 294) and deriving the

Procedure Recover

```
284 D := BaseAddress + 8                                              // Recover descriptors
285 head := D + N × SizeOf(Descriptor)                                 // Recover head
286 tail := head + SizeOf(Node)                                        // Recover tail
287 ptr := head                                                        // Recover all nodes from head to tail
288 while ptr ≠ BaseAddress do                                         // While ptr ≠ ⊥
289     count := 0                                                     // Recover the current node counter
290     for i := 0 to K − 1 do
291         if ptr.pairs[i].Get() ≠ ⟨⊥, ⊥⟩ then
292             count := count + 1
293     ptr.count := count
294     ptr.m := new Mutex                                             // Reset the node lock
295     ptr = ptr.next.GaP()
296 incompleteOp := new Map                                            // Map node pointers to PIDs
297 for i := 1 to N do                                                 // For every process i
298     ⟨op, _, _, _, res, pred⟩ := D[i].Resolve()                     // Resolve for Pi
299     if res = ⊥ ∧ op ≠ ⊥ ∧ pred ≠ BaseAddress then                   // If op is incomplete
300         incompleteOp[pred] = i                                     // Map pred pointer to PID
301     else if res = ⊥ ∧ op ≠ ⊥ then                                   // no effect
302         D[i].curr := 1 − D[i].curr                                 // clean up descriptor
303 foreach pred ∈ list of nodes from tail to head do
304     if pred ∈ incompleteOp then
305         curr := pred.next.GaP()
306         succ := curr.next.GaP()
307         i := incompleteOp[pred]                                    // Get PID
308         ⟨op, key, _, _, _⟩ := D[i].Resolve()
309         j := Seek(curr, key)                                       // Search for key in curr
310         if op = Insert ∧ (succ.anchor ≤ key ∨ j ≠ ⊥) then          // If split done or key in curr
311             D[i].SetResult(true)                                   // No need to insert
312         else if op = Remove ∧ j = ⊥ then                           // Key removed, merge or redistribute if needed
313             Execute Lines 221-242 of Remove except for (un)locking mutexes
314         else                                                       // no effect
315             D[i].curr := 1 − D[i].curr                             // clean up descriptor
```

Algorithm 5.13: Recovery protocol

value of the variable *count*. A node's *count* is inferred by traversing the array *pairs* and counting all valid key-value pairs (Lines 289-293).

After initializing the list variables from PM, the remainder of the recovery protocol uses the recovered descriptors to resolve and clean up interrupted operations. The lock properties must not be violated during recovery, as cleaning up incomplete operations in an arbitrary order can break strict linearizability. For instance, consider the following problematic scenario: suppose there are two processes, p_i and p_j, each performing a removal on two consecutive nodes, where the *curr* node of p_i is the *pred* node of p_j. Assume p_i's *curr* node contains only one pair, while p_j's *curr* node holds MinFull pairs. In a failure-free execution, p_i would only remove its *curr* node (Line 225) after p_j finishes merging its *curr* node and releases the lock on its *pred* node (Line 241), which corresponds to p_i's *curr.m* (Line 222). This sequential order of execution, enforced by the mutex locks, prevents the system from reaching invalid states. Now, imagine a crash failure occurs after both processes execute Line 219. Upon recovery, if p_i removes its *curr* node from the linked list first—thereby removing p_j's *pred* node—then p_j would incorrectly create a new node, new_1, while merging its *curr* and *succ* nodes, and set the next pointer of the removed *pred* node (previously p_i's *curr*) to new_1 (Line 235). This would lead to an inconsistent state in the linked list.

To prevent such inconsistencies, the sequential order enforced by the locks must be preserved. A key observation is that locks on consecutive nodes are always acquired in one direction (from head to tail) and released in the opposite direction. Therefore, resuming incomplete operations in reverse order—relative to the positions of their *pred* nodes in the linked list (Lines 297-313), from tail to head—ensures that all locks are released before being re-acquired by other processes.

The for-loop at Line 297 iterates through each descriptor object, reading its current unit. For each descriptor, the last operation of each process is resolved (Line 298) and checked to determine if it requires clean-up (Line 299). Incomplete operations (where $result = \perp$ in the corresponding descriptor) with a valid node address stored in the *offset* variable must be cleaned up. These operations are logged in an associative array, mapping each predecessor node address to the ID of the process that was executing the operation when the crash occurred. The remaining interrupted operations are discarded by changing the Boolean *curr* in the descriptor to point to the unit of the last completed operation (Line 302).

Next, the for-each-loop at Line 303 resolves (Line 308) and fixes the interrupted operations (Lines 310-313) in reverse order, based on the positions of their predecessor nodes in the linked list. For each **Insert** operation, if the anchor key of the successor node is

smaller than the key to be inserted, it indicates that both the node split and the key insertion occurred prior to the crash. If the key is found in *curr*, it confirms that the insertion took place without splitting the node. In both cases, the recovery protocol simply marks the operation as complete (Line 311). If neither of these conditions holds, the **Insert** procedure did not take effect before the crash. In this case, the **Recover** procedure discards the interrupted insertion by flipping the Boolean *curr* in the descriptor to point to the unit of the last completed operation at Line 315.

For interrupted **Remove** operations, if the key was removed prior to the failure (Line 312), Lines 221-242 of the **Remove** function are executed to merge, redistribute, or remove the node as necessary, while skipping the lock acquisition and release steps. Note that executing Lines 221-242 is idempotent. The checks at Lines 221 and 229 ensure that nodes are redistributed, merged, or removed at most once. If the key removal (Line 312) was not flushed prior to the system crash, the **Remove** procedure did not take effect before the crash. In this scenario, the **Recover** procedure cancels the interrupted operation by toggling the Boolean *curr* in the descriptor to point back to the unit corresponding to the last successfully completed operation (315).

5.4 Correctness

In this section, I present the proof for the safety and liveness property for DULL. I choose strict linearizability (Definition 2.9) as the safety property for DULL, and recoverable deadlock-freedom (Definition 2.10) as the liveness property.

5.4.1 Safety

My approach follows the proof technique established by Herlihy and Wing [41]. Let *DSet* denote the abstract type of the detectable unordered set defined in Section 2.6.6. A shared object of type *DSet* is implemented using a *representation type*. Let the representation type that implements *DSet* be the detectable unrolled linked list, denoted by *DULL*, introduced in this chapter. *DULL* consists of the following base objects: the *head* and *tail* pointers, descriptor variables, and node variables (e.g., key-value pairs and pointers), excluding mutex locks (m). I assume that each base object fits in one cache line.[2] Accordingly,

[2]This assumption can be satisfied by using indirection, where a pointer to the key-value pair object is stored if the size of the pair exceeds that of a cache line.

a history H consists of events involving both *DSet* and *DULL* objects, structured as follows: Each operation *op* on a *DSet* object consists of two events: invocation and response (Section 2.6.1), and is implemented by a sequence of operations on the base objects of *DULL* that occur within *op*. Each operation on a base object of *DULL* is an atomic read, write, or compare-and-swap (CAS) operation on a shared variable in persistent memory. Consequently, each operation on a base object of *DULL* has its invocation and response occurring simultaneously, generating only a single event in H. The sub-history of *DSet* events in H is denoted by $H|DSet$, while the sub-history of *DULL* events is denoted by $H|DULL$.

For a given history H, the value of an object in H at the end of a strict linearization of H is called a *strictly linearized value* of H. Let $SL(H|DSet)$ and $SL(H|DULL)$ denote the set of all strictly linearized values of H for *DSet* and *DULL* objects, respectively. Since the operations on a base object of *DULL* is atomic, there is only one strictly linearized value for $H|DULL$, which is denoted as r and called the representation value. Let $I(r)$ be a predicate, referred to as the *representation invariant*, which defines the allowed values of r, and let $A(r)$ be an abstraction function that maps r to a non-empty set of abstract values. I demonstrate that *DULL* implements *DSet* by proving that for every history H, the condition $s(H)$ holds such that [41]:

$$s(H) \iff I(r) \wedge A(r) \subseteq SL(H|DSet) \tag{5.1}$$

I will now proceed with the proof as follows:

1. I define the linearization points for the set operations.
2. For any given history H, I construct a strictly linearized history L based on the linearization points defined in 1, where the corresponding representation value is r.
3. I define a representation invariant I and an abstraction function A.
4. I prove that $I(r)$ holds and that $A(r) = SL(L)$.
5. Since L is a strictly linearized history of H induced by the linearization points defined in 1, it follows that $SL(L) \subseteq SL(H|DSet)$. Consequently, $A(r) = SL(L)$ implies $A(r) \subseteq SL(H|DSet)$. Combined with $I(r)$, this implies that $s(H)$ holds.

Linearization points

For each of the operations **Insert**, **Remove**, **Contains**, and **Resolve**, I define the linearization points, as follows:

Linearization Point 1. The function **Insert** is linearized as follows:

1. If $D[i].result$ is written at Line 105 in the function $D[i]$.**SetResult** to some value, called at Line 161, and the value is persisted to PM at Line 106, then the function is linearized at the earliest persistence, either at Line 106 or by another flush event.

2. Else if the condition at Line 166 is evaluated as **true** and the function sets $curr.pairs[j]$ $.pair$ at Line 167 to some value, and if that value is persisted to PM, which occurs either in $curr.pairs[j]$.**Persist** at Line 168 or before by another flush event, then linearize at line 167.

3. Else if the condition at Line 166 is evaluated as **false**, $pred.next$ is set at Line 90 to some value in the function $pred.next$.**SaP**, called at Line 181, and that value is persisted to PM, which occurs either at Line 91, or before by another flush event, then linearize at line 90.

4. Otherwise, **Insert** is a pending operation and no linearization point defined.

Linearization Point 2. The function **Remove** is linearized as follows:

1. If $D[i].result$ is written at Line 105 in the function $D[i]$.**SetResult** to some value, called at Line 214, and the value is persisted to PM at Line 106, then the function is linearized at the earliest persistence, either at Line 106 or by another flush event.

2. If $curr.pairs[j].pair$ is written at Line 218 to some value and that value is persisted to PM, which occurs either in $curr.pairs[j]$.**Persist** at Line 219, or before by another flush event, then the function is linearized at Line 218.

3. Otherwise, **Remove** is a pending operation and no linearization point defined.

Linearization Point 3. The function **Contains** is linearized as follows:

1. If the operation returns at Line 151, then there exists a position of step in H during the execution of **Contains** where the target pair is not in PM (shown in CASE A4 of the proof for Lemma 5.1), I linearize at this step.

2. If the operation returns at Line 150, and the value read at Line 99 in the function *curr.pairs*[*j*].**Get** (Line 147) is persisted to PM, either at Line 148 or before by another flush event, then linearize at Line 99.

3. Otherwise, **Contains** is a pending operation and no linearization point defined.

Linearization Point 4. The function **Resolve** is linearized at the response of the operation at Line 283.

Note that for each operation performed, its linearization point, if it exists, corresponds to an event between the operation's invocation and response. Thus, all linearization points are unique.

Candidate linearization history

For every history H consisting of **Insert**, **Remove**, **Contains**, **Resolve** operations, and crash events, I define a candidate linearization history L, which is constructed as follows:

1. For each pending operation in H, if its linearization point exists based on the defined Linearization Points 1-4, a matching response is added prior to the next crash event, if one occurs, or at the end if no crash event is present as follows:

   (a) **Resolve**: The linearization point occurs at Line 283, where the response is returned, which is then added to L.

   (b) **Insert**: If the linearization point at Line 161 is reached, the corresponding response event returns **false**, which is the only possible outcome (Line 163). If the linearization point at Line 167 or 181 is reached, the response event returns **true**, which is the sole possible outcome (Line 186).

   (c) **Remove**: If the operation hits the linearization point at Line 214 or 218, the matching response event returns either **false** or **true**, which are the only possible outcomes at Line 216 or Line 242, respectively.

   (d) **Contains**: According to Linearization Point 3, a pending **Contains** operation has no linearization points. Therefore, no response is added.

2. Pending operations in H that lack linearization points are removed.

3. Construct the sequential history L such that all the remaining operations in H occur in the order of their linearization point.

Representation invariant

Given a representation value r of *DULL*, let **pval**(r) be the value of r in the persistent memory, and **cval**(r) be the value of r in the cache if it exists; otherwise it is **pval**(r). The representation value r contains the linked list variables: all nodes (except their mutex locks), head and tail pointers, and the array of descriptors. In addition, r also includes private variables for each active process p_i, such as program counters, denoted $r.PC[i]$. The rep invariant $I(r)$ is illustrated in Figure 5.2.

$$
\begin{aligned}
I(r) = {} & r.head \neq \bot \\
& \wedge\, r.tail \neq \bot \\
& \wedge\, r.tail.next = \bot \\
& \wedge\, r.head.anchor = -\infty \\
& \wedge\, r.head.next.anchor = -\infty \\
& \wedge\, r.tail.anchor = \infty \\
& \wedge\, \textbf{isConnected}(r) \\
& \wedge\, \textbf{hasNoDuplicate}(r) \\
& \wedge\, \textbf{isAnchorOrdered}(r) \\
& \wedge\, \textbf{isWeaklyOrdered}(r) \\
& \wedge\, \textbf{noMarkedReachable}(r)
\end{aligned}
$$

Figure 5.2: The representation invariant $I(r)$ of *DULL*.

Both the head and tail pointers must not be $\bot$. The next pointer of the *tail* must be $\bot$. The *anchor* values of the head, first non-sentinel (*head*'s *next*), and tail nodes must always be $-\infty$, $-\infty$, and ∞, respectively. The **isConnected** function returns **true** if the node referenced by the tail pointer can be reached from the node referenced by the head pointer by traversing the next pointers of each node. Otherwise, it returns **false**. I define the **hasNoDuplicate** function to count each key in r and returns **true** if all key counts are ≤ 1, and returns **false** otherwise. The function **isAnchorOrdered** is defined as follows: While traversing the list in r through the *next* pointer of each node n, if $n.next \neq \bot \wedge n.anchor \geq n.next.anchor$ return **false**. Otherwise, **isAnchorOrdered** returns **true**. The function **isWeaklyOrdered** returns **true** only if every key in r is greater than or equal

to its containing node's anchor and less than its successor's; otherwise, it returns **false**. Finally, the **noMarkedReachable** function is defined as iterating through r's list and returning **false** if any node in the list $marked = \textbf{false}$. Otherwise, **noMarkedReachable** returns **true**.

Abstraction function

The abstract value of *DSet* in L is denoted as v. The value v contains the items in the set $\textbf{items}(v)$, the last operation's signature, tag and response executed by each process, denoted $v.\mathcal{A}[p_i]$, $v.\mathcal{T}[p_i]$ and $v.\mathcal{R}[p_i]$ for process p_i. For the abstraction function $A(r)$, I define the functions **nodeItems**, **getSig**, **getTag**, and **getRes**. The function $\textbf{nodeItems}(r)$ is defined as follows:

1. For each node in the list r from head to tail:
   (a) For each item $t \neq \langle \perp, \perp \rangle$ in the array *pairs*, if t is in PM, add t to a set s
2. Return the set s.

I define the following notations:

$$\text{let } r \xrightarrow[i]{curr} x \text{ denote } \textbf{pval}(r.D[i].units[\textbf{pval}(r.D[i].curr)].x)$$
$$\text{and } r \xrightarrow[i]{next} x \text{ denote } \textbf{pval}(r.D[i].units[\textbf{pval}(1 - r.D[i].curr)].x)$$

where x is some variable name in *units*.

For the **getSig**, **getTag** and **getRes** functions, I define the following conditions:

X1: p_i is at Lines 153-167 in **Insert** or 206-218 in **Remove**, and $r \xrightarrow[i]{curr} result = \perp$.

X2: p_i is at Lines 168-169 in **Insert** or 219-239 in **Remove**, and:

a) $\textbf{cval}(r[p_i].curr.pairs[j].pair) \neq \textbf{pval}(r[p_i].curr.pairs[j].pair)$.
b) $\textbf{cval}(r[p_i].curr.pairs[j].pair) = \textbf{pval}(r[p_i].curr.pairs[j].pair)$.

X3: p_i is at Lines 171-180 in **Insert**.

X4: p_i is at Lines 181-183 in **Insert** and:

a) $\mathbf{pval}(r[p_i].pred.next) \neq r[p_i].new_1$.

b) $\mathbf{pval}(r[p_i].pred.next) = r[p_i].new_1$.

X5: p_i is at Line 184-186 in **Insert** or 240-242 in **Remove**.

X6: Otherwise.

Accordingly, the **getSig**, **getTag** and **getRes** functions are defined in Figure 5.3.

$$\mathbf{getSig}(r,i) = \begin{cases} \langle r \xrightarrow[i]{next} op, r \xrightarrow[i]{next} args\rangle & \textbf{if } X1 \vee X2a \vee X3 \vee X4a \\ \langle r \xrightarrow[i]{curr} op, r \xrightarrow[i]{curr} args\rangle & \textbf{otherwise} \end{cases}$$

$$\mathbf{getTag}(r,i) = \begin{cases} r \xrightarrow[i]{next} tag & \textbf{if } X1 \vee X2a \vee X3 \vee X4a \\ r \xrightarrow[i]{curr} tag & \textbf{otherwise} \end{cases}$$

$$\mathbf{getRes}(r,i) = \begin{cases} r \xrightarrow[i]{next} result & \textbf{if } X1 \vee X2a \vee X3 \vee X4a \\ \textbf{true} & \textbf{if } X2b \vee X4b \vee X5 \\ r \xrightarrow[i]{curr} result & \textbf{otherwise} \end{cases}$$

Figure 5.3: The definition of the **getSig**, **getTag** and **getRes** functions in $A(r)$

The abstraction function $A(r)$, which maps r to v, is shown in Figure 5.4.

$$\begin{aligned} A(r) = \{v \mid \mathbf{items}(v) &= \mathbf{nodeItems}(r) \\ \wedge \forall i \in \{1,...,N\}, v.\mathcal{A}[p_i] &= \mathbf{getSig}(r,i) \\ \wedge v.\mathcal{T}[p_i] &= \mathbf{getTag}(r,i) \\ \wedge v.\mathcal{R}[p_i] &= \mathbf{getRes}(r,i)\} \end{aligned}$$

Figure 5.4: The abstraction function $A(r)$ that maps a representation value (r) to a set of abstract values of a *DSet*'s object (v).

Proving $s(H)$

I will now prove that, for any given history H, $I(r)$ holds and $A(r) = SL(L)$, where L is a strictly linearized history of H induced by the previously defined linearization points. Since $SL(L) \subseteq SL(H|DSet)$, it follows that $A(r) = SL(L)$ implies $A(r) \subseteq SL(H|DSet)$. Together with $I(r)$, this establishes that $s(H)$ holds.

Lemma 5.1. *Given any finite history H, the candidate linearization history L derived from H is legal, and $A(r) = SL(L)$. As a result, $s(H)$ holds.*

Proof. I will use induction on the length of the history H.

BASE CASE ($|H| = 0$):

Since H is empty, L is identical to H. At the start (Section 5.2), both the head and tail pointers of $DULL$ ($r.head$ and $r.tail$) are set to point to sentinel nodes. The *next* pointer of the head sentinel node points to the first non-sentinel node, and the *next* pointer of the non-sentinel node points to the tail node. This ensures that **isConnected** returns **true**. The *anchor* values of the head, the first non-sentinel node, and the tail node are initialized to $-\infty$, $-\infty$, and ∞, respectively. Thus, the function **isAnchorOrdered** returns **true**. Additionally, for all three nodes, the *pairs* array is empty (each entry is $\langle \perp, \perp \rangle$) and the *marked* variable is set to **false**. As a result, **hasNoDuplicate**, **isWeaklyOrdered** and **noMarkedReachable** functions return **true**. Therefore, the invariant $I(r)$ holds.

Regarding $A(r)$, the **nodeItems** function returns an empty sequence since the *pairs* arrays in all nodes are empty (each entry is $\langle \perp, \perp \rangle$), which matches the initial value of $DSet$. At the initial state, all variables in the descriptors array D are set to $\perp$ (except for $offset = 0$). Therefore, $r \xrightarrow[i]{curr} op$ is $\perp$. Also, all processes are not executing any operation. Hence, Case X6 applies, and **getSig**, **getTag**, and **getRes** all return $\perp$. According to the $DSet$ specifications (Section 2.6.6), $v.\mathcal{A}[p_i]$, $v.\mathcal{T}[p_i]$, and $v.\mathcal{R}[p_i]$ initially map all processes p_i to $\perp$. Thus, the only element of $A(r)$ is a strict linearized value of L. The statement $s(H)$ holds, and $A(r) = SL(L)$.

STEP:

For any $i \geq 0$, assume that $s(H)$ holds when $|H| = i$, and consider $|H'| = i + 1$ such that the history H' consists of H along with an additional event e as the $(i+1)$-th event. The representation value for H' is denoted as r', and the corresponding candidate linearization history of H' is denoted as L'. The event e must either be (A) an event of $DULL$ represented by a specific line in Algorithms 5.1-5.13, (B) an invocation or response event of a $DSet$ operation, or a (C) crash event.

CASE A: e is an event of *DULL*.

Assume that e is executed by process p_i. By the induction hypothesis, if $r = r'$ and $L = L'$, then $s(H')$ holds. Therefore, I analyze each line that modifies r.

CASE A1: operation **Insert**:

Lines 109-116 in **Prep-Insert** *at Line 152: prepare for inserting a key-value pair.* In **Insert**(*key*, *value*, *tag*), the function **Prep-Insert**(*key*, *value*, *tag*) is called at Line 152, which involves changing r as follows:

- $r \xrightarrow[i]{next}$ *op* is set to **Insert** at Line 110.
- $r \xrightarrow[i]{next}$ *args* is set to $\langle key, value \rangle$ at Line 111.
- $r \xrightarrow[i]{next}$ *tag* is set to *tag* at Line 112.
- $r \xrightarrow[i]{next}$ *result* is set to $\perp$ at Line 113.
- $r \xrightarrow[i]{next}$ *offset* is set to 0 at Line 114.
- **cval**$(r.D[i].curr)$ is set to $1 - r.D[i].curr$ at Line 116.

Only descriptor variables change from r to r'. According to Figure 5.2, such variables do not affect the representation invariant. Therefore, $I(r) = I(r')$. Since $I(r)$ holds, by the induction hypothesis, then $I(r')$ also holds.

For the abstraction function $A(r)$, no keys have been inserted or removed from the list r'. Therefore, **nodeItems**(r') = **nodeItems**(r). Since **Insert** is not linearized at this point, L' and $SL(L')$ have no changes from L and $SL(L)$, respectively. It follows that L' is legal.

Furthermore, Lines 109-116, do not change $r \xrightarrow[i]{curr}$ *op*, $r \xrightarrow[i]{curr}$ *args*, $r \xrightarrow[i]{curr}$ *tag*, $r \xrightarrow[i]{curr}$ *result* or **pval**$(r'.D[i].curr)$. Therefore, Case X6 applies to both r and r', and **getSig**(r', i), **getTag**(r', i) and **getRes**(r', i) return the same value as in r. Accordingly, $A(r) = A(r')$ and, by the induction hypothesis, $A(r) = SL(L)$. Since L is equivalent to L', then $A(r') = SL(L')$. As a result $S(H')$ holds.

Line 117 in **Prep-Insert** *at Line 152: flush curr.* Only descriptor variables change from r to r'. According to Figure 5.2, such variables do not affect the representation invariant. Therefore, $I(r) = I(r')$. Since $I(r)$ holds, by the induction hypothesis, then $I(r')$ also holds. By the induction hypothesis, $I(r')$ holds. By the event e, no keys have been inserted or removed from the list r'. Therefore, **nodeItems**(r') = **nodeItems**(r). Since **Insert** is not linearized at this point, then L' and $SL(L')$ have no changes from L and $SL(L)$, respectively. Hence, L is

legal and, by the induction hypothesis, $\textbf{items}(v') = \textbf{items}(v) = \textbf{nodeItems}(r) = \textbf{nodeItems}(r')$.

When p_i flushes the *curr* to PM at Line 117, it also goes to Line 154, changing Case X6 to X1. Accordingly,

$$\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{next} op, r' \xrightarrow[i]{next} args \rangle && \text{(Case X1 in } r') \\
&= \langle r \xrightarrow[i]{curr} op, r \xrightarrow[i]{curr} args \rangle && \text{(Lines 116-117)} \\
&= \textbf{getSig}(r, i) && \text{(Case X6 in } r) \\
&= v.\mathcal{A}[p_i] && \text{(induction hypothesis)} \\
&= v'.\mathcal{A}[p_i] && (v = v')
\end{aligned}$$

$$\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{next} tag && \text{(Case X1 in } r') \\
&= r \xrightarrow[i]{curr} tag && \text{(Lines 116-117)} \\
&= \textbf{getTag}(r, i) && \text{(Case X6 in } r) \\
&= v.\mathcal{T}[p_i] && \text{(induction hypothesis)} \\
&= v'.\mathcal{T}[p_i] && (v = v')
\end{aligned}$$

$$\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{next} result && \text{(Case X1 in } r') \\
&= r \xrightarrow[i]{curr} result && \text{(Lines 116-117)} \\
&= \textbf{getRes}(r, i) && \text{(Case X6 in } r) \\
&= v.\mathcal{R}[p_i] && \text{(induction hypothesis)} \\
&= v'.\mathcal{R}[p_i] && (v = v')
\end{aligned}$$

Accordingly, $A(r') = SL(L')$, and $S(H')$ holds.

Line 161: Set result to false. At Line 161, p_i invokes the function **SetResult** in which it sets $r \xrightarrow[i]{curr} result$ to **false** (Line 105) from $\perp$ (previously set at Line 152), and then flush that value to PM at Line 106. The operation is linearized when the value written at Line 105 is persisted by the flush at Line 106, thus updating the abstract state.

According to the induction hypothesis, the linked list maintains the representation invariant, specified in Figure 5.2, for all events in H. This guarantees that the **Scan** function returns the correct node where the target *key* would appear,

and **Seek** returns $\perp$ only if *key* is absent in that node—and therefore absent in the entire linked list.

The if-statement at Line 160 indicates that *key* exists in the linked list at some point during **Seek** at Line 159. Since *pred* is locked, *key* also exists in the list when p_i reaches Line 161, including at the moment when the **Insert** function is linearized. Eventually, p_i unlocks *pred.m* (Line 162), then returns **false** (Line 163), which is the correct value to return. Hence, L' is legal.

According to the sequential specification (Figure 2.4), I have $v'.\mathcal{A}[p_i] = \langle$**Insert**, $\langle key, value\rangle\rangle$ (which is the signature of the current invocation of **Insert**), $v'.\mathcal{T}[p_i] = tag$, $v'.\mathcal{R}[p_i] =$ **false** and **items**(v') = **items**(v). As for the representation value, persisting $r \xrightarrow[i]{curr} result$ changes Case X1 in r to X6 in r'. Therefore,

$$\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{curr} op, r' \xrightarrow[i]{curr} args\rangle && \text{(Case X6 in } r'\text{)} \\
&= \langle \textbf{Insert}, \langle key, value\rangle\rangle && \text{(Lines 110-111)} \\
&= v'.\mathcal{A}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{curr} tag && \text{(Case X6 in } r'\text{)} \\
&= tag && \text{(Line 112)} \\
&= v'.\mathcal{T}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{curr} result && \text{(Case X6 in } r'\text{)} \\
&= \textbf{false} && \text{(Line 105 in } \textbf{SetResult} \text{ at Line 161)} \\
&= v'.\mathcal{R}[p_i]
\end{aligned}$$

Accordingly, $A(r') = SL(L')$.

Only descriptor variables change from r to r'. According to Figure 5.2, such variables do not affect the representation invariant. Therefore, $I(r) = I(r')$. Since $I(r)$ holds, by the induction hypothesis, then $I(r')$ also holds. As a result $S(H')$ holds.

Line 167: set the key-value pair. According to the induction hypothesis, the linked list satisfies the representation invariant, specified in Figure 5.2, for all events in H. This ensures that the **Scan** function returns the correct node where the target *key* would appear, and **Seek** returns $\perp$ only if *key* is absent in that node—and therefore absent in the entire linked list. At 167, p_i has previously returned $j = \perp$

from an invocation of **Seek** at Line 165, indicating that *key* does not exist in the linked list at some point during that invocation of **Seek**. Since *pred.m* is locked, no other process could possibly insert *key*, ensuring that *key* is not in the list when p_i inserts *key* at Line 167. Therefore, **hasNoDuplicate**(r') holds. Since no other changes are applied to the representation value from r to r' (pointers or other node variables), then all other conditions in $I(r')$ hold from $I(r)$ by the induction hypothesis. Thus, $I(r')$ holds.

For the abstraction function $A(r)$, the inserted key is not persisted to PM in r'. Therefore, **nodeItems**(r') = **nodeItems**(r). Since **Remove** is not linearized at this point, then L' and $SL(L')$ have no changes from L and $SL(L)$, respectively. Hence, L' is legal and, by the induction hypothesis, **items**(v') = **items**(v) = **nodeItems**(r) = **nodeItems**(r'). Executing Line 167 involves changing p_i's program counter to point at Line 168, which changes Case X1 in r to Case X2a in r'. Functions **getSig**(r', i), **getTag**(r', i) and **getRes**(r', i) return the same value as in r since no changes to the descriptor variables occur, and the behavior of each of the three functions do not change from Case X1 to X2a (Figure 5.3). Accordingly, $A(r) = A(r')$ and, by the induction hypothesis, $A(r) = SL(L)$. Since L is equivalent to L', then $A(r') = SL(L')$. As a result $S(H')$ holds.

Line 168: persist the key value pair. The operation is linearized when the value written at Line 167 is persisted to PM, either by the flush at Line 168 or by another concurrent operation (**Contains**), thus updating the abstract state.

According to the induction hypothesis, $I(r)$ (Figure 5.2) holds for all events in H. This ensures that the **Scan** function returns the correct node where the target *key* would appear, and **Seek** returns $\perp$ only if *key* is absent in that node—and therefore absent in the entire linked list. If p_i reaches Line 169, then p_i has returned $j = \perp$ from an invocation of **Seek** at Line 159, indicating that *key* does not exist in the linked list at some point during that invocation of the function **Seek**. Since *pred.m* is locked, no other process could possibly have inserted *key*, ensuring that *key* is not in the list when p_i reaches Line 161 and remains absent at the time the **Insert** function is linearized. Eventually, if p_i does not crash, it will unlock *pred.m* (Line 185) and then return **true** (Line 186), which is the correct response. Therefore, L' is legal.

According to the sequential specification (Figure 2.4), I have $v'.\mathcal{A}[p_i] = \langle$**Insert**, $\langle key, value\rangle\rangle$, $v'.\mathcal{T}[p_i] = tag$, $v'.\mathcal{R}[p_i]$ = **true** and **items**(v') = **items**$(v) \cup \{\langle key, value\rangle\}$. As for the representation value, persisting the inserted pair changes Case

X2a in r to X2b in r', and adds $\langle key, value \rangle$ to the persisted list. Therefore,

$$\begin{aligned}
\textbf{nodeItems}(r') &= \textbf{nodeItems}(r) \cup \{\langle key, value \rangle\} && \text{(Lines 167 and 168)} \\
&= \textbf{items}(v) \cup \{\langle key, value \rangle\} && \text{(induction hypothesis)} \\
&= \textbf{items}(v')
\end{aligned}$$

$$\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{curr} op, r' \xrightarrow[i]{curr} args \rangle && \text{(Case X6 in } r') \\
&= \langle \textbf{Insert}, \langle key, value \rangle \rangle && \text{(Lines 110-111)} \\
&= v'.\mathcal{A}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{curr} tag && \text{(Case X6 in } r') \\
&= tag && \text{(Line 112)} \\
&= v'.\mathcal{T}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{curr} result && \text{(Case X6 in } r') \\
&= \textbf{true} && \text{(Line 105 in } \textbf{SetResult} \text{ at Line 184)} \\
&= v'.\mathcal{R}[p_i]
\end{aligned}$$

Accordingly, $A(r') = SL(L')$.

Since no other linked list variables are changed from r to r', then $I(r) = I(r')$. By the induction hypothesis, $I(r')$ holds. As a result $S(H')$ holds.

Line 90 in **SaP** *at Line 181: set the next pointer of previous node.* At Line 181, p_i replaces *curr* with two nodes new_1 and new_2. Now, I will show that the representation invariant r' resulted from such replacement satisfies $I(r')$.

First, p_i creates a local copy, *tmp*, of *curr*'s *pairs* array (Line 187) and sorts it by key (Line 188). New nodes new_1 and new_2 are allocated (Lines 189 and 190), with the smallest portion, $\lfloor K/2 \rfloor$, of the sorted pairs copied to new_1 and the remaining pairs to new_2 (Lines 191-196). The anchor key of new_1 is set to the anchor of *curr* (Line 197), while the anchor key of new_2 is set to the key of its first pair (Line 200). Since *tmp* is sorted, then all keys in new_1 are less than all keys in new_2. As a result, $new_1.anchor < new_2.anchor$. By the induction hypothesis, **isWeaklyOrdered**(r) implies that all keys in *curr*, and thus the two new nodes, are less than *succ.anchor*. The given pair, $\langle key, value \rangle$, is inserted into

new_1 if *key* is less than the anchor key of new_2; otherwise, it is inserted into new_2. Therefore, **isWeaklyOrdered**(r') holds. The pointer $new_1.next$ is set to point to new_2, $new_2.next$ is set to point to *succ* (Lines 202 and 203), respectively. Since $pred.anchor < curr.anchor = new_1.anchor < new_2.anchor < succ.anchor$, and the rest of the lined list is unchanged, then **isAnchorOrdered**(r') also holds. By the induction hypothesis, **isConnected**(r) holds, which implies that $r.tail$ can be reached from $r.head$. Since *succ* can be reached from new_2 (Line 203), which can be reached from new_1 (Line 202), which can be reached from *pred* (Line 90), then **isConnected**(r') holds.

At 181, p_i has previously returned $j = \perp$ from an invocation of **Seek** at Line 159, indicating that *key* does not exist in the linked list at some point during that invocation of the function **Seek**. Since $pred.m$ is locked, no other process could possibly change the state of the node *curr* until p_i unlocks $pred.m$ at Line 185 or a crash occurs, including when p_i inserts the new pair at Line 181. Therefore, **hasNoDuplicate**(r') holds. Since no other changes are applied to the representation value from r to r', then all the other conditions in $I(r')$ (Figure 5.2) hold from $I(r)$ by the induction hypothesis. Thus, $I(r')$ holds.

For the abstraction function $A(r)$, the replacement at Line 90, and thus the key insertion, is not persisted to PM in r'. Therefore, **nodeItems**(r') = **nodeItems**(r). At Line 167, does not change the Case X4a in r' from r. Functions **getSig**(r', i), **getTag**(r', i) and **getRes**(r', i) return the same value as in r since no changes to the descriptor variables occur. Accordingly, $A(r) = A(r')$. There is no change in L' from L since **Insert** is linearized at this point. By the induction hypothesis, $SL(L') = SL(L)$, L is legal, and $A(r') = SL(L')$. As a result $S(H')$ holds.

Line 91 in **SaP** *at Line 181: flush the next pointer of the previous node.* The operation is linearized when the value written at Line 90 is persisted to PM, either by the flush at Line 91 or by another concurrent operation (through the function **Scan**), thus updating the abstract state.

According to the induction hypothesis, $I(r)$ (Figure 5.2) holds for all events in H. This ensures that the **Scan** function returns the correct node where the target *key* would appear, and **Seek** returns $\perp$ only if *key* is absent in that node—and therefore absent in the entire linked list. If p_i reaches Line 169, then p_i has returned $j = \perp$ from an invocation of **Seek** at Line 159, indicating that *key* does not exist in the linked list at some point during that invocation of the function **Seek**. Since $pred.m$ is locked, no other process could possibly have inserted *key*,

ensuring that *key* is not in the list when p_i reaches Line 161 and remains absent at the time the **Insert** function is linearized. Eventually, if p_i does not crash, it will unlock *pred.m* (Line 185) and then return **true** (Line 186), which is the correct response. Therefore, L' is legal.

According to the sequential specification (Figure 2.4), I have $v'.\mathcal{A}[p_i] = \langle \textbf{Insert}, \langle key, value \rangle \rangle$, $v'.\mathcal{T}[p_i] = tag$, $v'.\mathcal{R}[p_i] = \textbf{true}$ and $\textbf{items}(v') = \textbf{items}(v) \cup \{\langle key, value \rangle\}$. As for the representation value, persisting the *pred.next*, which points to new_1 and new_2 that contains the inserted pair, changes Case X4a in r to X4b in r', and adds $\langle key, value \rangle$ to the persisted list. Therefore,

$$\begin{aligned}
\textbf{nodeItems}(r') &= \textbf{nodeItems}(r) \cup \{\langle key, value \rangle\} && \text{(Lines 174, 177 and 181)} \\
&= \textbf{items}(v) \cup \{\langle key, value \rangle\} && \text{(induction hypothesis)} \\
&= \textbf{items}(v')
\end{aligned}$$

$$\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{curr} op, r' \xrightarrow[i]{curr} args \rangle && \text{(Case X4b in } r') \\
&= \langle \textbf{Insert}, \langle key, value \rangle \rangle && \text{(Lines 110-111)} \\
&= v'.\mathcal{A}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{curr} tag && \text{(Case X4b in } r') \\
&= tag && \text{(Line 112)} \\
&= v'.\mathcal{T}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getRes}(r', i) &= \textbf{true} && \text{(Case X4b in } r') \\
&= v'.\mathcal{R}[p_i]
\end{aligned}$$

Accordingly, $A(r') = SL(L')$.

Only descriptor variables change from r to r'. According to Figure 5.2, such variables do not affect the representation invariant. Therefore, $I(r) = I(r')$. Since $I(r)$ holds, by the induction hypothesis, then $I(r')$ also holds. As a result $S(H')$ holds.

Lines 184-186: set result to true. When the program counter of p_i changes to 184, Case X5 applies in r'. However, the linked list variables are not changed from r to r', which implies that $I(r) = I(r')$ and $\textbf{nodeItems}(r') = \textbf{nodeItems}(r)$. In addition, no linearization on *DSet* occurs, and therefore, L' and $SL(L')$ are the same as

L and $SL(L)$, respectively. By the induction hypothesis, $I(r')$ holds and L is legal. As for $A(r')$, $v = v'$. Since the function is linearized in a prior step, according to the sequential specification (Figure 2.4), I have $v'.\mathcal{A}[p_i] = \langle \textbf{Insert}, \langle key, value \rangle \rangle$, $v'.\mathcal{T}[p_i] = tag$, $v'.\mathcal{R}[p_i] = \textbf{true}$ and $\textbf{items}(v') = \textbf{items}(v) = \textbf{nodeItems}(r)$. Hence, it follows that $\textbf{nodeItems}(r') = \textbf{items}(v')$, and

$$\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{curr} op, r' \xrightarrow[i]{curr} args \rangle && \text{(Case X5 in } r'\text{)} \\
&= \langle \textbf{Insert}, \langle key, value \rangle \rangle && \text{(Lines 110-111)} \\
&= v'.\mathcal{A}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{curr} tag && \text{(Case X5 in } r'\text{)} \\
&= tag && \text{(Line 112)} \\
&= v'.\mathcal{T}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{curr} result && \text{(Case X5 in } r'\text{)} \\
&= \textbf{true} && \text{(Line 105 in } \textbf{SetResult} \text{ at Line 184)} \\
&= v'.\mathcal{R}[p_i]
\end{aligned}$$

Accordingly, $A(r') = SL(L')$, and $S(H')$ holds.

CASE A2: operation **Remove**:

Lines 109-116 in **Prep-Remove** *at Line 205: prepare for removing a key-value pair.* In $\textbf{Remove}(key, tag)$, the function $\textbf{Prep-Remove}(key, \perp, tag)$ is called at Line 205, which involves changing r as follows:

- $r \xrightarrow[i]{next} op$ is set to **Remove** at Line 110.
- $r \xrightarrow[i]{next} args$ is set to $\langle key, \perp \rangle$ at Line 111.
- $r \xrightarrow[i]{next} tag$ is set to tag at Line 112.
- $r \xrightarrow[i]{next} result$ is set to $\perp$ at Line 113.
- $r \xrightarrow[i]{next} offset$ is set to 0 at Line 114.
- $\textbf{cval}(r.D[i].curr)$ is set to $1 - r.D[i].curr$ at Line 116.

Only descriptor variables change from r to r'. According to Figure 5.2, such variables do not affect the representation invariant. Therefore, $I(r) = I(r')$. Since $I(r)$ holds, by the induction hypothesis, then $I(r')$ also holds.

For the abstraction function $A(r)$, no keys have been inserted or removed from the list r'. Therefore, **nodeItems**(r') = **nodeItems**(r). Since **Remove** is not linearized at this point, L' and $SL(L')$ have no changes from L and $SL(L)$, respectively. In follows that L' is legal.

Furthermore, Lines 109-116, do not change $r \xrightarrow[i]{curr} op$, $r \xrightarrow[i]{curr} args$, $r \xrightarrow[i]{curr} tag$, $r \xrightarrow[i]{curr} result$ and **pval**$(r'.D[i].curr)$. Therefore, Case X6 is not changed from r to r', and **getSig**(r', i), **getTag**(r', i) and **getRes**(r', i) return the same value as in r. Accordingly, $A(r) = A(r')$ and, by the induction hypothesis, $A(r) = SL(L)$. Since L is equivalent to L', then $A(r') = SL(L')$. As a result $S(H')$ holds.

Line 117 in **Prep-Remove** *at Line 205: flush curr.* Only descriptor variables change from r to r'. According to Figure 5.2, such variables do not affect the representation invariant. Therefore, $I(r) = I(r')$. Since $I(r)$ holds, by the induction hypothesis, then $I(r')$ also holds. By the event e, no keys have been inserted or removed from the list r'. Therefore, **nodeItems**(r') = **nodeItems**(r). Since **Remove** is not linearized at this point, then L' and $SL(L')$ have no changes from L and $SL(L)$, respectively. Hence, L is legal and, by the induction hypothesis, **items**(v') = **items**(v) = **nodeItems**(r) = **nodeItems**(r').

When p_i flushes the *curr* to PM at Line 117, it also goes to Line 154, changing Case X6 to X1. Accordingly,

$$\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{next} op, r' \xrightarrow[i]{next} args \rangle && \text{(Case X1 in } r') \\
&= \langle r \xrightarrow[i]{curr} op, r \xrightarrow[i]{curr} args \rangle && \text{(Lines 116-117)} \\
&= \textbf{getSig}(r, i) && \text{(Case X6 in } r) \\
&= v.\mathcal{A}[p_i] && \text{(induction hypothesis)} \\
&= v'.\mathcal{A}[p_i] && (v = v')
\end{aligned}$$

$$\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{next} tag && \text{(Case X1 in } r') \\
&= r \xrightarrow[i]{curr} tag && \text{(Lines 116-117)} \\
&= \textbf{getTag}(r, i) && \text{(Case X6 in } r) \\
&= v.\mathcal{T}[p_i] && \text{(induction hypothesis)} \\
&= v'.\mathcal{T}[p_i] && (v = v')
\end{aligned}$$

$$
\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{next} \mathit{result} && \text{(Case X1 in } r'\text{)} \\
&= r \xrightarrow[i]{curr} \mathit{result} && \text{(Lines 116-117)} \\
&= \textbf{getRes}(r, i) && \text{(Case X6 in } r\text{)} \\
&= v.\mathcal{R}[p_i] && \text{(induction hypothesis)} \\
&= v'.\mathcal{R}[p_i] && (v = v')
\end{aligned}
$$

Accordingly, $A(r') = SL(L')$, and $S(H')$ holds.

Line 214: set result to false. At Line 214, p_i invokes the function **SetResult** in which it sets $r \xrightarrow[i]{curr} \mathit{result}$ to **false** (Line 105) from $\perp$ (previously set at Line 205), and then flushes that value to PM at Line 106. The operation is linearized when the value written at Line 105 is persisted by the flush at Line 106, thus updating the abstract state.

According to the induction hypothesis, the linked list maintains the representation invariant, specified in Figure 5.2, for all events in H. This guarantees that the **Scan** function returns the correct node where the target *key* would appear, and **Seek** returns $\perp$ only if *key* is absent in that node—and therefore absent in the entire linked list.

The if-statement at Line 213 indicates that *key* does not exist in the linked list at some point during **Seek** at Line 212. Since *pred* is locked, *key* also does not exist in the list when p_i reaches Line 214, including at the moment when the **Remove** function is linearized. Eventually, p_i unlocks *pred.m* (Line 215), then returns **false** (Line 216), which is the correct value to return. Hence, L' is legal.

According to the sequential specification (Figure 2.4), $v'.\mathcal{A}[p_i] = \langle$**Remove**, $\langle key, \perp\rangle\rangle$ (which is the signature of the current invocation of **Remove**), $v'.\mathcal{T}[p_i] = tag$, $v'.\mathcal{R}[p_i] =$ **false** and **items**(v') = **items**(v). As for the representation value,

persisting $r \xrightarrow[i]{curr} result$ changes Case X1 in r to X6 in r'. Therefore,

$$\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{curr} op, r' \xrightarrow[i]{curr} args \rangle && \text{(Case X6 in } r'\text{)} \\
&= \langle \textbf{Remove}, \langle key, \perp \rangle \rangle && \text{(Lines 110-111)} \\
&= v'.\mathcal{A}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{curr} tag && \text{(Case X6 in } r'\text{)} \\
&= tag && \text{(Line 112)} \\
&= v'.\mathcal{T}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{curr} result && \text{(Case X6 in } r'\text{)} \\
&= \textbf{false} && \text{(Line 105 in } \textbf{SetResult} \text{ at Line 214)} \\
&= v'.\mathcal{R}[p_i]
\end{aligned}$$

Accordingly, $A(r') = SL(L')$.

Only descriptor variables change from r to r'. According to Figure 5.2, such variables do not affect the representation invariant. Therefore, $I(r) = I(r')$. Since $I(r)$ holds, by the induction hypothesis, then $I(r')$ also holds. As a result $S(H')$ holds.

Line 218: set found key-value pair to $\langle \perp, \perp \rangle$. According to the induction hypothesis, the linked list satisfies the representation invariant, specified in Figure 5.2, for all events in H. This ensures that the **Scan** function returns the correct node where the target *key* would appear, and **Seek** returns $\perp$ only if *key* is absent in that node—and therefore absent in the entire linked list. At 218, p_i has previously returned $j \neq \perp$ from an invocation of **Seek** at Line 212, indicating that *key* exists in the linked list at some point during that invocation of **Seek** in the j-th element of the *pairs* array in p_i's *curr* node. Since *pred.m* is locked, no other process could possibly alter the state of *curr*, ensuring that p_i removes *key* by setting *curr.pairs*[j].*pair* to $\langle \perp, \perp \rangle$ at Line 218.

Since no other changes are applied to the representation value from r to r' (pointers or other node variables), then all conditions in $I(r')$ hold from $I(r)$ by the induction hypothesis. Thus, $I(r')$ holds.

For the abstraction function $A(r)$, the removed key is not persisted to PM in r'. Therefore, **nodeItems**(r') = **nodeItems**(r). Since **Remove** is not linearized at

this point, then L' and $SL(L')$ have no changes from L and $SL(L)$, respectively. Hence, L' is legal and, by the induction hypothesis, $\textbf{items}(v') = \textbf{items}(v) = \textbf{nodeItems}(r) = \textbf{nodeItems}(r')$. Executing Line 218 involves changing p_i's program counter to point at Line 219, which changes Case X1 in r to Case X2a in r'. Functions $\textbf{getSig}(r', i)$, $\textbf{getTag}(r', i)$ and $\textbf{getRes}(r', i)$ return the same value as in r since no changes to the descriptor variables occur, and the behavior of each of the three functions do not change from Case X1 to X2a (Figure 5.3). Accordingly, $A(r) = A(r')$ and, by the induction hypothesis, $A(r) = SL(L)$. Since L is equivalent to L', then $A(r') = SL(L')$. As a result $S(H')$ holds.

Line 219: flush removed key-value pair. The operation is linearized when the value written at Line 218 is persisted to PM, either by the flush at Line 219 or by another concurrent operation (**Contains**), thus updating the abstract state.

According to the induction hypothesis, the linked list satisfies the representation invariant, specified in Figure 5.2, for all events in H. This ensures that the **Scan** function returns the correct node where the target *key* would appear, and **Seek** returns $\perp$ only if *key* is absent in that node—and therefore absent in the entire linked list. At 218, p_i has previously returned $j \neq \perp$ from an invocation of **Seek** at Line 212, indicating that *key* exists in the linked list at some point during that invocation of **Seek** in the j-th element of the *pairs* array in p_i's *curr* node. Since *pred.m* is locked, no other process could possibly alter the state of *curr*, ensuring that *key* is in $\textbf{pval}(r.curr.pairs[j].pair)$ at the time the **Remove** function is linearized. Eventually, if p_i does not crash, it will unlock *pred.m* (Line 241) and then return **true** (Line 242), which is the correct response. Therefore, L' is legal.

According to the sequential specification (Figure 2.4), $v'.\mathcal{A}[p_i] = \langle \textbf{Remove}, \langle key, \perp \rangle \rangle$, $v'.\mathcal{T}[p_i] = tag$, $v'.\mathcal{R}[p_i] = \textbf{true}$ and $\textbf{items}(v') = \textbf{items}(v) \backslash \{\langle key, value \rangle\}$. As for the representation value, persisting the removed pair changes Case X2a in

r to X2b in r', and removes $\langle key, value\rangle$ from the persisted list. Therefore,

$$\begin{aligned}
\textbf{nodeItems}(r') &= \textbf{nodeItems}(r) \setminus \{\langle key, value\rangle\} && \text{(Lines 218 and 219)}\\
&= \textbf{items}(v) \setminus \{\langle key, value\rangle\} && \text{(induction hypothesis)}\\
&= \textbf{items}(v')
\end{aligned}$$

$$\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{curr} op, r' \xrightarrow[i]{curr} args\rangle && \text{(Case X6 in } r')\\
&= \langle \textbf{Remove}, \langle key, \perp\rangle\rangle && \text{(Lines 110-111)}\\
&= v'.\mathcal{A}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{curr} tag && \text{(Case X6 in } r')\\
&= tag && \text{(Line 112)}\\
&= v'.\mathcal{T}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{curr} result && \text{(Case X6 in } r')\\
&= \textbf{true} && \text{(Line 105 in } \textbf{SetResult} \text{ at Line 240)}\\
&= v'.\mathcal{R}[p_i]
\end{aligned}$$

Accordingly, $A(r') = SL(L')$.

Since no other linked list variables are changed from r to r', then $I(r) = I(r')$. By the induction hypothesis, $I(r')$ holds. As a result $S(H')$ holds.

Lines 90 and 91 in **SaP** *at Line 235: set next pointer of previous node.* If p_i reaches Line 235, then there are two scenarios: (1) p_i executed the function **Merge** at Line 230, or (2) p_i executed **Redistribute** at Line 232.

CASE (1): **Merge**.

At Line 235, p_i replaces *curr* and *succ* with a new node new_1. Now, I will show that the representation invariant r' resulted from such replacement satisfies $I(r')$.

In the function **Merge**, p_i creates a new node new_1 at Line 243. Then, it copies all pairs with keys $\neq \perp$ from *curr* and *succ* into new_1 (Lines 245-253). The anchor key of new_1 is assigned to the anchor of *curr* (Line 254). By the induction hypothesis, $curr.anchor < succ.anchor$ and $curr.anchor$ is less than or equal to all keys in *curr* and *succ* in r. This implies that

$new_1.anchor$ is less than or equal to all keys in $curr$ and $succ$ in r'. Therefore, **isWeaklyOrdered**(r') holds. The pointer $new_1.next$ is set to point to $succ.next$ at Line 256. Since $pred.anchor < curr.anchor = new_1.anchor < succ.anchor < succ.next.anchor$, and the rest of the lined list is unchanged, then **isAnchorOrdered**(r') holds. In addition, **isConnected**(r) holds, By the induction hypothesis, which implies that $r.tail$ can be reached from $r.head$. Since $succ.next$ can be reached from new_1 (Line 256), which can be reached from $pred$ (Line 90), then **isConnected**(r') holds.

CASE (2): **Redistribute**.

At Line 235, p_i replaces $curr$ and $succ$ with a two nodes: new_1 and new_2. Now, I will show that the representation invariant r' resulted from such replacement satisfies $I(r')$.

By Line 221, $curr.count$ is less than MinFull, and by Line 229, the combined count of $curr$ and $succ$ is at least MaxMerge $= 3 \times$ MinFull, implying $succ.count \geq 2 \times$MinFull $> curr.count$. Thus, $M = (\frac{\lceil curr.count + succ.count \rceil}{2} - curr.count)$ pairs are moved from $succ$ to $curr$ (Line 265).

Process p_i copies all pairs from $curr$ to new_1 in Lines 260-264, then moves the smallest M pairs from $succ$ to new_1 and the remaining pairs to new_2 (Lines 265-273). The anchor of new_1 is set to $curr$'s anchor key (Line 274), while new_2's anchor is set to its smallest key (first pair) from tmp, which is ordered (Line 277). Since tmp is sorted, then all keys in new_1 are less than all keys in new_2. As a result, $new_1.anchor < new_2.anchor$. By the induction hypothesis, **isWeaklyOrdered**(r) implies that all keys in $curr$ and $succ$, and thus the two new nodes, are less than $succ.next.anchor$ and greater than or equal $pred.anchor$. Therefore, **isWeaklyOrdered**(r') holds. The pointer $new_1.next$ is set to point to new_2 and $new_2.next$ is set to point to $succ$ at Lines 279 and 280, respectively. Since $pred.anchor < curr.anchor = new_1.anchor < new_2.anchor < succ.next.anchor$, and the rest of the lined list is unchanged, then **isAnchorOrdered**(r') also holds. By the induction hypothesis, **isConnected**(r) holds, which implies that $r.tail$ can be reached from $r.head$. Since $succ$ can be reached from new_2 (Line 203), which can be reached from new_1 (Line 202), which can be reached from $pred$ (Line 90), then **isConnected**(r') holds.

In both scenarios, From Lines 224 and 227, it is guaranteed that $curr$ and $succ$ (to be potentially replaced) are not the $head$, first non sentinel, or $tail$ nodes. Therefore, the following conditions hold: $r'.head \neq \perp$, $r'.tail \neq \perp$, $r'.tail.next = \perp$,

$r'.head.anchor = -\infty$, $r'.head.next.anchor = -\infty$ and $r'.tail.anchor = \infty$. Since no other changes are applied to the representation value from r to r', then all the other conditions in $I(r')$ (Figure 5.2) hold from $I(r)$ by the induction hypothesis. Thus, $I(r')$ holds.

For the abstraction function $A(r)$, the replacement at Line 90, does not change the **nodeItems**(r') from **nodeItems**(r) as merging and redistributing do not add or remove pairs into the list. At Line 168, does not change the Case X2b in r' from r. Functions **getSig**(r', i), **getTag**(r', i) and **getRes**(r', i) return the same value as in r since no changes to the descriptor variables occur. Accordingly, $A(r) = A(r')$. There is no change in L' from L since **Remove** has been previously linearized. By the induction hypothesis, $SL(L') = SL(L)$, L is legal, and $A(r') = SL(L')$. As a result $S(H')$ holds.

Lines 240-242: set result to true. When the program counter of p_i changes to 240, Case X5 applies in r'. However, the linked list variables are not changed from r to r', which implies that $I(r) = I(r')$ and **nodeItems**(r') = **nodeItems**(r). In addition, no linearization on *DSet* occurs, and therefore, L' and $SL(L')$ are the same as L and $SL(L)$, respectively. By the induction hypothesis, $I(r')$ holds and L is legal. As for $A(r')$, $v = v'$. Since the function is linearized in a prior step, according to the sequential specification (Figure 2.4), I have $v'.\mathcal{A}[p_i] = \langle$**Remove**$, \langle key, \perp\rangle\rangle$, $v'.\mathcal{T}[p_i] = tag$, $v'.\mathcal{R}[p_i]$ = **true** and **items**(v') = **items**(v) = **nodeItems**(r). Hence, it follows that **nodeItems**(r') = **items**(v'), and

$$\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{curr} op, r' \xrightarrow[i]{curr} args\rangle && \text{(Case X5 in } r'\text{)}\\
&= \langle \textbf{Remove}, \langle key, \perp\rangle\rangle && \text{(Lines 110-111)}\\
&= v'.\mathcal{A}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{curr} tag && \text{(Case X5 in } r'\text{)}\\
&= tag && \text{(Line 112)}\\
&= v'.\mathcal{T}[p_i]
\end{aligned}$$

$$\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{curr} result && \text{(Case X5 in } r'\text{)}\\
&= \textbf{true} && \text{(Line 105 in } \textbf{SetResult} \text{ at Line 240)}\\
&= v'.\mathcal{R}[p_i]
\end{aligned}$$

Accordingly, $A(r') = SL(L')$, and $S(H')$ holds.

CASE A3: operation **Resolve**:

Lines 282-283: resolve the last successful operation. Process p_i does not modify r in Lines 282-283. Thus, the values of $A(r)$ remain the same as $A(r')$, and likewise, $I(r')$ remains unchanged from $I(r)$. By the construction of L, L' is formed by appending the event e with response $(r \xrightarrow[i]{curr} op, r \xrightarrow[i]{curr} args, r \xrightarrow[i]{curr} tag, r \xrightarrow[i]{curr} result) = (v.\mathcal{A}[p_i], v.\mathcal{T}[p_i], v.\mathcal{R}[p_i])$, which is the correct response. Therefore, L' is a legal extension of L. There is also no alteration in $SLin(L')$ from $SLin(L)$. Consequently, it directly follows from the induction hypothesis that $A(r') = SL(L')$, $I(r')$ holds, and thus $s(H')$ holds.

CASE A4: operation **Contains**:

Lines 144-151: search for a key-value pair. Process p_i does not modify r at Lines 144-151. Thus, the values of $A(r)$ remain the same as $A(r')$, and likewise, $I(r')$ remains unchanged from $I(r)$. When p_i returns at Line 150 or 151, L' is formed by appending the event e to L with some response $res \in \{\textbf{true}, \textbf{false}\}$. According to the sequential specification, no alteration in $SLin(L')$ from $SLin(L)$.

Therefore, now all what I need to prove $S(H')$ is to show that L' is a legal extension of L.

p_i returns at Line 150.

If p_i reaches Line 150, then p_i read some pair $\langle k, v\rangle$, where $k = key$ (Line 149), and returns v (which represents the response **true**). There are two possibilities:

- If $\langle k, v\rangle$ was already persisted to PM at the time p_i reads it, then linearize at the time of the pair is read by p_i. According to the sequential specification, the response is **true**. Therefore, L' is legal.
- If $\langle k, v\rangle$ was not persisted to PM at the time p_i reads it, the variable has be set to $\langle k, v\rangle$ and was not persisted. In *DULL*, this is only possible by another process p_j (for $j \neq i$) at Line 167 in **Insert**. In **Insert**, p_j at Line 167, implies that p_j locked its predecessor mutex lock (Line 155), which implies that no other process can modify the variable set at 167 before p_j unlocks the lock. Therefore, the value is guaranteed to be persisted either by p_i at Line 148 or p_j at Line 168. The function is then linearized at the time the pair is read by p_i. According to the sequential specification, the response is **true**. Therefore, L' is legal.

p_i returns at Line 151.

First, I show that if p_i returns at Line 151, then there exists a time during the execution of the function **Contains**, such that *key* does not exist in the linked list.

Suppose for contradiction that p_i returns at Line 151 and *key* is in the list during the whole execution of **Contains**. If p_i is at Line 151, then p_i returned a node *curr* from **Scan** at Line 144, and j from **Seek** at Line 145. Since $j = \perp$ (Line 146), then p_i read all pairs in *curr*, during its invocation of **Seek**(*curr*, *key*) at Line 145, and that each read key $k \neq key$ (Line 141). Hence, there are two possible scenarios:

- *key* exists in the node *curr*: Since p_i observed that all keys in *curr* are not equal to *key*, then *key* was removed from one position before p_i read it and reinserted into a different position afterward in *curr.pairs*. If the removal happened before the reinsertion, there must be a point in time between these two operations where *key* was absent from the linked list, which contradicts our assumption. Conversely, if the removal took place after the reinsertion, there would be a point in time where *key* appears twice in the linked list, violating **hasNoDuplicate**(r), and hence $I(r)$, which contradicts our induction hypothesis.

- *key* does not exist in the node *curr*: By the induction hypothesis, **isAnchorOrdered** implies that for each node n in the list, $n.anchor < n.next.anchor$. Assuming for contradiction that *key* exists in some node n', it follows from the induction hypothesis and **isWeaklyOrdered** that $n'.anchor \leq key < n'.next.anchor$. Let *succ* denote the successor node of *curr*. According to Line 133, $curr.anchor \leq key < succ.anchor$. By **isAnchorOrdered** in the representation invariant, this implies that $n' = curr$, contradicting our assumption.

Therefore, there exists a time during the execution of the function **Contains** by p_i, such that *key* does not exist in the linked list. The function is linearized at that time with the response **false** (represented as $\perp$ in Algorithm 5.6), which is the correct response according to the sequential specification (Figure 2.4). Hence L' is legal.

p_i crashes before it returns at Line 150 or 151.

If the function does not return due to a crash event, then it is not linearized. Therefore, no change in L' from L, and the legality of L' directly follows from

L by the induction hypothesis.

CASE B: e is an event of $DSet$.

The event e must either be an invocation or a response related to an operation on $DSet$. If e is an invocation, and no additional lines of $DULL$ code are executed following it, then no state change occurs, as it is the final event. If e is a response, it is added to the history after an **Insert**, **Remove**, or **Contains** operation concludes, specifically at Line 150, 151, 163, 186, 216, or 242. Thus, the values of $A(r)$ remain unchanged from $A(r')$, and similarly, $I(r')$ is unchanged from $I(r)$. By the construction of L, there is no alteration of L' from L or $SLin(L')$ from $SLin(L)$. Consequently, $s(H')$ directly follows from $s(H)$ according to the induction hypothesis.

CASE C: e is a crash event.

According to Linearization Points 1-4, none of these points occur in the recovery phase. As a result, no operation reaches its linearization point throughout recovery, which implies $L' = L$. Thus, by the induction hypothesis, $SLin(L')$ remains equal to $SLin(L)$, and L' is considered legal since L is legal. The recovery process for $DULL$, function **Recover** in Algorithm 5.13, is initiated following a system crash to confirm that $A(r') = Lin(L')$.

First, D, *head*, and *tail* are set to the addresses of the descriptors array, head node, and tail node, respectively (Lines 284-286). Once the list *head* and *tail* pointers are initialized, I know by $I(r)$ that at this point $head \neq \perp$, $tail \neq \perp$, $tail.next = \perp$, $head.anchor = -\infty$, $head.next.anchor = -\infty$ and $tail.anchor = \infty$. In **Recover**, no pairs are inserted into or removed from the linked list. Therefore, by the induction hypothesis, $\textbf{nodeItems}(r') = \textbf{nodeItems}(r) = \textbf{items}(v) = \textbf{items}(v')$. The rest of the proof of CASE C shows that $\textbf{getSig}(r') = v'.\mathcal{A}[p_i]$, $\textbf{getTag}(r') = v'.\mathcal{T}[p_i]$, and $\textbf{getRes}(r') = v'.\mathcal{R}[p_i]$, and thus $A(r') = Lin(L')$. The recovery protocol classifies pending detectable operations into three categories, based on which specific actions taken as needed to ensure that $A(r') = Lin(L')$. Let p_i be the process executing the pending operation under consideration:

1. Non-linearized and unprepared pending **Insert** or **Remove**: This category includes **Insert** and **Remove** operations that crashed before the *curr* Boolean in the corresponding descriptor was flushed at Line 117 during the **Prep-***op* call (Lines 152 in **Insert** and 205 in **Remove**). As a result, $r \xrightarrow[i]{curr} op$, $r \xrightarrow[i]{curr} tag$, and $r \xrightarrow[i]{curr} res$ remain unchanged in that invocation of **Insert** or **Remove**. Such descriptor variables are left unmodified in **Recover**. In

addition, Case X6 holds both immediately before and after the crash. Consequently, **getSig**(r'), **getTag**(r'), and **getRes**(r') return the same values as **getSig**(r), **getTag**(r), and **getRes**(r). Thus, by the induction hypothesis, it follows that $A(r') = SL(L')$.

2. Non-linearized and prepared pending **Insert** or **Remove**: This category includes **Insert** and **Remove** operations that crashed after altering $r \xrightarrow[i]{curr} op$, $r \xrightarrow[i]{curr} tag$, and $r \xrightarrow[i]{curr} res$ during the **Prep-***op* call (Lines 152 in **Insert** and 205 in **Remove**) and before reaching a linearization point. Operation **Insert** is linearized at Line 161, 167 or 181, and Operation **Remove** is linearized at Line 214 or 218. Accordingly, if the pending operation is prepared and not linearized before the crash, then upon recovery, each corresponding descriptor has $r \xrightarrow[i]{curr} op \neq \perp$ (Line 110), $r \xrightarrow[i]{curr} offset = 0$ (Line 114) and $r \xrightarrow[i]{curr} res = \perp$ (Line 113). These descriptor variables are used by the recovery protocol to identify this category of operations (Line 301). In addition, Case X1, X2a, X3 or X4a applies immediately before the crash occurs. Therefore, when the recovery protocol sets the corresponding descriptor's *curr* to $1 - curr$, then:

$$
\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{curr} op, r' \xrightarrow[i]{curr} args \rangle && \text{(Case X6 in } r') \\
&= \langle r \xrightarrow[i]{next} op, r \xrightarrow[i]{next} args \rangle && \text{(Line 302)} \\
&= \textbf{getSig}(r, i) && \text{(Case X1, X2a, X3 or X4a in } r) \\
&= v.\mathcal{A}[p_i] && \text{(induction hypothesis)} \\
&= v'.\mathcal{A}[p_i] && (v = v')
\end{aligned}
$$

$$
\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{curr} tag && \text{(Case X6 in } r') \\
&= r \xrightarrow[i]{next} tag && \text{(Line 302)} \\
&= \textbf{getTag}(r, i) && \text{(Case X1, X2a, X3 or X4a in } r) \\
&= v.\mathcal{T}[p_i] && \text{(induction hypothesis)} \\
&= v'.\mathcal{T}[p_i] && (v = v')
\end{aligned}
$$

$$
\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{curr} result && \text{(Case X6 in } r'\text{)} \\
&= r \xrightarrow[i]{next} result && \text{(Line 302)} \\
&= \textbf{getRes}(r, i) && \text{(Case X1, X2a, X3 or X4a in } r\text{)} \\
&= v.\mathcal{R}[p_i] && \text{(induction hypothesis)} \\
&= v'.\mathcal{R}[p_i] && (v = v')
\end{aligned}
$$

As a result, $A(r') = SL(L')$.

3. Linearized pending operations: This category includes **Insert** and **Remove** operations that crashed after reaching a linearization point. Operation **Insert** is linearized at Line 161, 167 or 181, and Operation **Remove** is linearized at Line 214 or 218. If the linearization point of the pending operation is at Line 161, or 214 then no actions are taken by the recovery protocol. As a result:

$$
\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{curr} op, r' \xrightarrow[i]{curr} args \rangle && \text{(Case X6 in } r'\text{)} \\
&= \langle r \xrightarrow[i]{curr} op, r \xrightarrow[i]{curr} args \rangle && \text{(no changes in } \textbf{Recover}\text{)} \\
&= \langle op, args \rangle && \text{(Line 152 or 205)} \\
&= v'.\mathcal{A}[p_i] && \text{(Figure 2.4)}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{curr} tag && \text{(Case X6 in } r'\text{)} \\
&= r \xrightarrow[i]{curr} tag && \text{(no changes in } \textbf{Recover}\text{)} \\
&= tag && \text{(Line 152 or 205)} \\
&= v'.\mathcal{T}[p_i] && \text{(Figure 2.4)}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{curr} result && \text{(Case X6 in } r'\text{)} \\
&= r \xrightarrow[i]{curr} result && \text{(no changes in } \textbf{Recover}\text{)} \\
&= \textbf{false} && \text{(Line 161 or 214)} \\
&= v'.\mathcal{R}[p_i] && \text{(Figure 2.4)}
\end{aligned}
$$

As a result, $A(r') = SL(L')$. If the linearization point of the pending operation is at Line 167, 181 or 218, the $r \xrightarrow[i]{curr} \textit{offset}$ must have been set to the offset of $r.pred$ at Line 164 or 217, and may or may not set $r \xrightarrow[i]{curr} result$ to **true** at Line 184 or 240. Operations that set and persisted $r \xrightarrow[i]{curr} result$ be-

fore it crashes, need no further actions upon recovery. The argument used for Lines 161 and 214 can be used to show that if Lines 184 and 240 are executed before the crash, no action is needed and $A(r') = SL(L')$. Therefore, the recovery protocol identifies the pending operations where $r \xrightarrow[i]{curr} \mathit{offset} \neq \bot$ and $r \xrightarrow[i]{curr} \mathit{result} = \bot$ (Line 299). These operations are logged in an associative array, mapping each predecessor node address to the process ID that was executing the operation during the crash (Line 300). Required actions, if any, for each of these operations are performed in the for-each-loop at Lines 303-315, to ensure that $A(r') = SL(L')$. For each **Insert** operation, if the successor node's anchor key is smaller than the key to be inserted, both the node split and key insertion occurred before the crash. If the key is found in *curr*, it confirms insertion without node splitting. In these cases, the operation **Insert** is linearized and the recovery protocol simply marks the operation as complete (Line 311). As a result,

$$
\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{curr} \mathit{op}, r' \xrightarrow[i]{curr} \mathit{args} \rangle && \text{(Case X6 in } r'\text{)} \\
&= \langle r \xrightarrow[i]{curr} \mathit{op}, r \xrightarrow[i]{curr} \mathit{args} \rangle && \text{(no changes in } \textbf{Recover}\text{)} \\
&= \langle \textbf{Insert}, \langle \mathit{key}, \mathit{value} \rangle \rangle && \text{(Line 152)} \\
&= v'.\mathcal{A}[p_i] && \text{(}\textbf{Insert}\text{ is linearized before crash)}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{curr} \mathit{tag} && \text{(Case X6 in } r'\text{)} \\
&= r \xrightarrow[i]{curr} \mathit{tag} && \text{(no changes in } \textbf{Recover}\text{)} \\
&= \mathit{tag} && \text{(Line 152)} \\
&= v'.\mathcal{T}[p_i] && \text{(}\textbf{Insert}\text{ is linearized before crash)}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{curr} \mathit{result} && \text{(Case X6 in } r'\text{)} \\
&= r \xrightarrow[i]{curr} \mathit{result} && \text{(no changes in } \textbf{Recover}\text{)} \\
&= \textbf{true} && \text{(Line 184)} \\
&= v'.\mathcal{R}[p_i] && \text{(}\textbf{Insert}\text{ is linearized before crash)}
\end{aligned}
$$

Hence, $A(r') = SL(L')$. Otherwise, the **Insert** procedure was not linearized but prepared before the crash, which is equivalent to Category 2 where setting the corresponding descriptor's *curr* to $1 - \mathit{curr}$ (Line 315) ensures that $A(r') = SL(L')$. For interrupted **Remove** operations, if the key was deleted

prior to failure (Line 312), Lines 221-242 of **Remove** are executed to merge, redistribute, or remove the node as necessary, bypassing lock acquisition and release. CASE A of the induction step proves that Lines 221-242 satisfy $I(r')$. Hence,

$$\begin{aligned}
\textbf{getSig}(r', i) &= \langle r' \xrightarrow[i]{curr} op, r' \xrightarrow[i]{curr} args \rangle && \text{(Case X6 in } r'\text{)} \\
&= \langle r \xrightarrow[i]{curr} op, r \xrightarrow[i]{curr} args \rangle && \text{(no changes in } \textbf{Recover}\text{)} \\
&= \langle \textbf{Remove}, \langle key, \bot \rangle \rangle && \text{(Line 205)} \\
&= v'.\mathcal{A}[p_i] && (\textbf{Remove} \text{ is linearized before crash})
\end{aligned}$$

$$\begin{aligned}
\textbf{getTag}(r', i) &= r' \xrightarrow[i]{curr} tag && \text{(Case X6 in } r'\text{)} \\
&= r \xrightarrow[i]{curr} tag && \text{(no changes in } \textbf{Recover}\text{)} \\
&= tag && \text{(Line 205)} \\
&= v'.\mathcal{T}[p_i] && (\textbf{Remove} \text{ is linearized before crash})
\end{aligned}$$

$$\begin{aligned}
\textbf{getRes}(r', i) &= r' \xrightarrow[i]{curr} result && \text{(Case X6 in } r'\text{)} \\
&= r \xrightarrow[i]{curr} result && \text{(no changes in } \textbf{Recover}\text{)} \\
&= \textbf{true} && \text{(Line 240)} \\
&= v'.\mathcal{R}[p_i] && (\textbf{Remove} \text{ is linearized before crash})
\end{aligned}$$

Accordingly, $A(r') = SL(L')$. If the key removal (Line 312) was not flushed before the crash, then the procedure was not linearized but prepared before the crash, which is equivalent to Category 2 case where setting the corresponding descriptor's *curr* to $1 - curr$ (Line 315) ensures that $A(r') = SL(L')$.

In all cases, $A(r') = SL(L')$, and thus $s(H')$ holds.

Now, I shall demonstrate that the recovery protocol is idempotent, meaning that executing it multiple times results in the same system state. Lines 284-292 read the node pairs, D, *head*, and *tail*. None of these variables are modified during the execution of the recovery protocol. Thus, reading them multiple times guarantees consistent results. Consequently, Line 293 always assigns the same value to *count*. Additionally, all locks will eventually be reset at Line 294. Repeating Lines 296-302 involves two possible cases for each descriptor resolved at Line 298:

1. **No Changes:** If *res*, *op*, and *pred* remain unchanged, then repeating Lines 296-

302 produces the same outcome.

2. **Changes Occur:** If any of the three variables change, it means that one of the following lines has executed: Line 302, 311, 313, or 315.

For Lines 302 and 315, the descriptor is reset to the valid unit, as the other unit is guaranteed to be valid by the induction hypothesis. This scenario corresponds to Case 2, where a crash occurs in a non-linearized and prepared pending **Insert** or **Remove** operation. Repeating the recovery process ensures such operations are excluded from *incompleteOp* at Line 300. Consequently, Lines 302 and 315 take effect only once during multiple consecutive executions of the recovery protocol.

For Lines 311 and 313, which correspond to Case 3 (linearized pending operations), the following applies: If *result* is set to **true**, then repeating recovery ensures such operations are excluded from *incompleteOp* at Line 300. Otherwise, the operations remain in *incompleteOp*, and *result* is eventually set to true at either Line 311 or Line 313. Thus, these lines also take effect only once over multiple consecutive executions of the recovery protocol.

Since the recovery protocol is idempotent, a crashed followed by a repeated invocation of **Recover** in H can be replaced by a single execution of the recovery protocol.

□

5.4.2 Liveness

Lemma 5.2. *DULL satisfies recoverable deadlock-freedom (Definition in 2.10).*

Proof. I prove that, given a fair infinite history H, if a process p_i executes a pending operation in H, there exists a process (possibly p_i) that completes operations, by reaching a matching response event, infinitely many times in H, or a system crash occurs. Assume for contraction that p_i executes an infinitely pending operation *op* and no process completes operations in H. I will analyze for each possible operation *op* as follows:

CASE A: *op* = **Resolve**.

Since p_i is the only process that accesses its own descriptor, the progress of the **Resolve** procedure depends solely on the steps taken by p_i. Therefore, eventually p_i reaches a matching response for the invocation of **Resolve** under consideration, which is a contradiction.

CASE B: $op =$ **Contains**.

If p_i is executing the **Contains** function forever, then either p_i is in the **Seek** (Line 145) or **Scan** (Line 144) functions as the number of steps in Lines 146-151 is bounded by $O(n)$ of p_i's own steps. The function **Seek** has one loop which is bounded by the constant parameter K. Then, p_i must be stuck in the **Scan** function. The while loop ends when p_i reads $succ.anchor > key$, which implies that $succ$ points to the $tail$ node. By $S(H)$ (Lemma 5.1), I know that $tail$ is reachable from $head$, and that $tail.anchor = \infty$. Accordingly, if p_i is looping forever, then there exists some other process p_j that is adding nodes infinitely many times to the linked list. Process p_j can add more nodes only at Line 181, after which it is guaranteed to eventually return, and thus complete the operation, at Line 186. Hence, a contradiction.

CASE C: $op =$ **Insert** or $op =$ **Remove**

As for the update procedures, **Insert** and **Remove**, I discuss the progress for the **Insert** function, and an analogous argument can be applied to the **Remove** function. From CASE B, I know that p_i cannot possibly be executing the **Scan** or **Seek** functions forever; otherwise, it contradicts our assumption. Therefore, if p_i is executing **Insert** forever, then either:

1. p_i executes Line 158 infinitely many times: If a process executes Line 158, then, by the condition in the if-statement at Line 156, one of the following must hold:
   (a) $pred$ or $curr$ is marked: By $S(H)$ (Lemma 5.1), a marked node cannot be reached from $head$, indicating that another process p_j removed this node from the list while p_i was executing **Insert**.
   (b) $pred.next \neq curr$: This implies that another process p_j modified $pred.next$ to point to a different node while p_i was executing **Insert**.

   Both node removal and updating $pred.next$ can only occur at Line 181 or 235 by p_j. After either of these lines, p_j is guaranteed to eventually return and complete its operation at Line 186 or 242, respectively. Thus, a contradiction arises.
2. p_i waits to acquire $pred.m$ at Line 155 or $curr.m$ at Line 171: In **Insert** and **Remove**, locks are acquired in the order of the predecessor, current and then successor nodes (from head to tail). Therefore, if p_i is waiting to acquire a lock held by p_j, then it is impossible that p_j is waiting to acquire a lock that is held by p_i. Hence, either p_j eventually releases the lock (Line 183, 185, 238, 239 or 241) and completes its operation at Line 186, 163, 216 or 242, contradicting our assumption, or p_j indefinitely waits to acquire a lock that is held by another process. Let $p^1, p^2 \dots p^k$ be a maximal sequence of processes that are executing

Insert or **Remove** operations on consecutive nodes $n_1, n_2 \ldots n_k$, respectively, such that $\forall m \in \{1 \ldots k-1\}, p^m$ waits to acquire a lock held by p^{m+1}. Since number of processes, N, and number of nodes in the linked list, call it M, are finite, then p^k and n_k exist and $k \leq \min\{N, M\}$. Therefore, p^k does not wait for any lock acquisition, and thus is guaranteed eventually to complete its operation, contradicting our assumption.

□

5.4.3 Correctness Theorem

Theorem 5.3. *DULL is strictly-linearizable and recoverably deadlock-free.*

Proof. DULL is strictly linearizable by Lemma 5.1, and recoverably deadlock-free by Lemma 5.2. □

5.5 Experimental Evaluation

The recoverability of the proposed algorithm is tested using a strict-linearizability checker, which is an extended variant of the algorithm proposed by Anderson et al. [2], and its performance is evaluated against the state-of-the-art lock-free and lock-based solutions [20, 65, 66, 39], as well as its non-recoverable variant [62]. The constants MinFull and MaxMerge are set to $K/4$ and $3 \times K/4$, respectively, and $K = 8$ is used in all my experiments. All lists are implemented in C++ with the `-funroll-loops` optimization flag using gcc 9.3.0 on Ubuntu 20.04.1. A flush instruction is implemented as a CLWB (`_mm_clwb`) compiler intrinsic followed by a memory store fence (`_mm_sfence`). The server used in my experiments has a 2.1 GHz Intel Xeon processor that has 20 physical cores, in which CLWB instructions are supported. Turbo boost is disabled to obtain consistent evaluation results. Moreover, the server has 6 Intel Optane modules of 128GB each, PM is accessed using PMDK 1.7 library through memory-mapped files in App-Direct mode.

5.5.1 Recoverability Testing

To test the recoverability of the proposed linked list, I have instrumented the code base such that each thread logs the invocation and response for each of the following operations:

Contains, **Insert** and **Remove**. The result is a sub-history file per thread of sequential operations, which later are combined into one history file of interleaving operations. I implemented an extended variant of the algorithm proposed by Anderson et al. [2] to examine the produced execution history for strict linearizability violations. When creating history files to test recoverability, the benchmark ensures that the assumptions in [2] are satisfied, including:

- All procedures are reduced to read and write operations. The atomicity of reads immediately followed by writes to the same key-value pair is ensured through the lock-based synchronization mechanism employed by DULL.
  - **Contains**(key): a failed search is recorded as a read of the pair $(key, \perp)$[3] while a successful search is logged as a read of $(key, value)$, for some $value \neq \perp$.
  - **Insert**(key, $value$): a failed insertion is a read of $(key, value)$ while a successful insertion is logged as a read of $(key, \perp)$ followed by a write of $(key, value)$.
  - **Remove**(key, $value$): a failed removal is a read of $(key, \perp)$, while a successful removal is logged as a read of $(key, value)$ followed by a write of $(key, \perp)$.
- Each key-value pair is unique.
  - **Insertion**: Insertions of all pairs with the same key are guaranteed to have unique values. Thus, a key can be reinserted but with different values.
  - **Removal**: Absence of a key is represented using distinct values. A key that was never inserted has the value $value = \textsc{Init}$. A key's value is assigned to $\perp$ at most once, when it is firstly removed. Any subsequent successful removal of the same key only marks the value as deleted without actually removing the key from the list. Marking a value is implemented as follows: Bit-wise shift all values to the left upon logging. Use the least significant bit as the marker (1 for deleted and 0 otherwise).

The linearizability checker in [2] is based on the original definition of linearizability [42], where failure-free executions are assumed. Therefore, I extended the code to accommodate strict linearizability. When the application is restarted after a crash, the last entry of each history file is examined for an incomplete operation op. If op is a read, then op is removed from the log file. If op is a write, then PM content is examined to check whether the written value exists in memory. If the value is found in PM, then the response of the

[3] A unique value outside the key space is used to represent $\perp$.

write operation is appended. Otherwise, the write operation is removed from the log file. Removing an incomplete write operation $w(k, v)$ or appending its response based on the existence of the pair (k, v) in PM comes from the following intuition: If (k, v) is in PM, then a write operation $w'(k, v)$ must have written and persisted the pair (k, v). According to Algorithms 5.7 and 5.9, the thread that has started $w(k, v)$ must have locked the predecessor (Lines 155 and 208) of the node containing (k, v) and then found that k is not in PM (Lines 159 and 212). Therefore, $w(k, v)$ must be the operation that has written (k, v), i.e., $w'(k, v)$. Otherwise, $w(k, v)$ and $w'(k, v)$ would have simultaneously acquired the predecessor's lock, which contradicts the safety property of the mutex locks. The linearizability check is implemented in python and used for assessment.

I tested the algorithm initially by crashing the program (via `kill -9`), and then by repeatedly power-cycling the server, using an external power controller, to evaluate recoverability under system-wide failure.[4] The resultant history files were fed into the implemented linearizability checker, and no violations were reported.

5.5.2 Performance

The performance of the following list implementations is evaluated:

- **Vanilla-ULL-NONE-PM**: the original, non-recoverable, unrolled linked list [62], runs on PM with no memory reclamation.
- **DULL-DEBRA-PM**: our proposed algorithm runs on PM and uses Brown's distributed EBR variant, DEBRA [12], for node reclamation, with a custom bitmap allocator. Upon recovery, the allocator is reset and reinitialized by traversing the list to prevent memory leaks from crashes. I also replace our hand-tuned persistence mechanism (based on [71]) with the FliT library [72] and compare its performance to our custom solution.
- **DFC-ULL-NONE-PM**: a detectable unrolled linked list implemented using the flat-combining technique proposed in [66]. In DFC-ULL, insertions and removals are submitted to a global array and then handled by a combiner thread. DFC-ULL runs on PM with no memory reclamation.
- **OneFile-LL-PM**: a recoverable singly-linked list implemented using a lock-free persistent transactional memory (PTM) algorithm [65], where updates are performed using transactions, each has a unique ID. Inserting, or removing, an item and updating

[4] A test script was used to crash the server repeatedly for days.

the transaction ID occurs atomically using *double-word-compare-and-swap* (DCAS). OneFile-LL runs on PM and uses transactions to reclaim nodes.

- **Romulus-LL-PM**:A recoverable singly-linked list uses a lock-based PTM algorithm [20]. Romulus maintains two copies of the list. Updates modify the primary copy first, with the other updated afterward. In case of a crash, the consistent copy is used for recovery. Romulus-LL runs on PM and uses transactions for node reclamation.
- **Harris-LL-Hazard-DRAM**: the non-recoverable Harris' lock-free list algorithm, runs on DRAM and uses Michael's lock-free Hazard pointers [59] to reclaim nodes.
- **Harris-LL-DEBRA-DRAM**: Harris' lock-free linked list, runs on DRAM and uses Brown's distributed variant of EBR, DEBRA [12], to reclaim nodes.
- **PMDK-LL-PM**: a recoverable singly-linked list that is based on the undo-logging algorithm provided by PMDK object library (`libpmemobj`). PMDK runs on PM and uses transactions to reclaim nodes.

The source code for OneFile, Romulus, and PMDK was obtained from [63], and the source code for the FliT was acquired from [18]. FliT is a persistence library that tracks dirty cache lines using counters, ensuring efficient data persistence with minimal redundancy. It can be integrated by annotating the declarations of persistent variables with the `persist<>` template. Note that only the DULL and DFC-ULL are detectable while all the lists are recoverable, except the Vanilla-ULL and Harris lists. Although the Vanilla-ULL is not recoverable, its variables are stored in PM to assess the overhead of DULL. In addition, it is worth noting that OneFile relies on the double-width CAS primitive under the assumption of its failure atomicity. However, as demonstrated in [32], this assumption is not valid across all systems, such as Intel x86 architectures, where failure atomicity is not consistently guaranteed.

Our evaluation involves experiments conducted under various workloads: All-read workload which comprises 100% search operations. A Balanced workload of 25% insertions, 25% removals and 50% searches. All-write comprises operations of 50% insertions and 50% removals. In all experiments, lists are pre-filled with random key-value pairs to reach their *expected size*. The expected size of a list can be estimated by the key space and the percentage of insertions relative to removals. Mathematically, the expected size is $\frac{\text{Insertion}(\%)}{\text{Removal}(\%)+\text{Insertion}(\%)} \times$ Key Space. Furthermore, key space of 1000 and 5000 are used to evaluate the list performance.

5.5.3 Results

When evaluating performance I focus on crash-free executions as applications are expected to run free of failures most of the time in practice. Flush instructions are used in our failure-free experiments to reflect the lists' practical performance. Each run involves multiple threads that continuously insert, remove and lookup key-value pairs for a specific period of time. The metric used for evaluating throughput is the total number of operations per second over a varying number of threads. Experiments engage up to 250 threads in three different modes: The first mode uses only physical cores where up to 20 threads are one-to-one mapped to processor cores. The second mode utilizes hyper-threading in addition to physical cores and goes up to 40 threads. The third mode over-subscribes the 40 logical cores with up to 250 threads. The first two modes evaluate the scalability and performance of each list under consideration. The goal of the third mode is to assess system behaviour under high thread preemption. Empirical results are presented, in Figures 5.5-5.8, as graphs in which each point denotes the median of three consecutive runs of five seconds. For each point, there is an error bar that represents ± one standard deviation.

In all-write and balanced workloads, Figures 5.5-5.7, DULL outperforms all the other lists by several-fold in all of the three modes when considering a key-space of 1000 and 5000. The FliT-based DULL algorithm performed second best in terms of overall performance. FliT, designed as a general-purpose transformation tool, provides persistence by utilizing a custom persistent atomic type in place of the standard atomic library, making it particularly effective for persisting word-sized variables like pointers [72]. However, I found that FliT's generalized approach is not as efficient when persisting complex objects containing multiple variables, such as our descriptors. Specifically, our custom method, which issues a cache-line write-back instruction for all descriptor variables followed by a single store fence, achieves faster performance. This optimization is valid when the object variables are exclusively read and written by a single thread in the absence of failures, with other processes accessing them only during recovery after a crash. While FliT's generalization does not take advantage of such specific optimizations, it offers broader applicability, which limits its ability to maximize performance in our case. The two non-recoverable Harris lists are faster than the other algorithms at high concurrency levels. Note that Harris lists are faster than other lists because, unlike other algorithms, they run on DRAM and are not recoverable. Nevertheless, Harris lists are surpassed by our proposed algorithm.

In the first mode, where threads are one-to-one mapped to physical cores, the DFC-ULL algorithm comes in second place when comparing recoverable solutions in most of the workloads, except for the all-write workload when the key space is 1000 as Romulus-LL has a higher throughput. In addition, the Romulus-LL surpasses the OneFile-LL and PMDK-

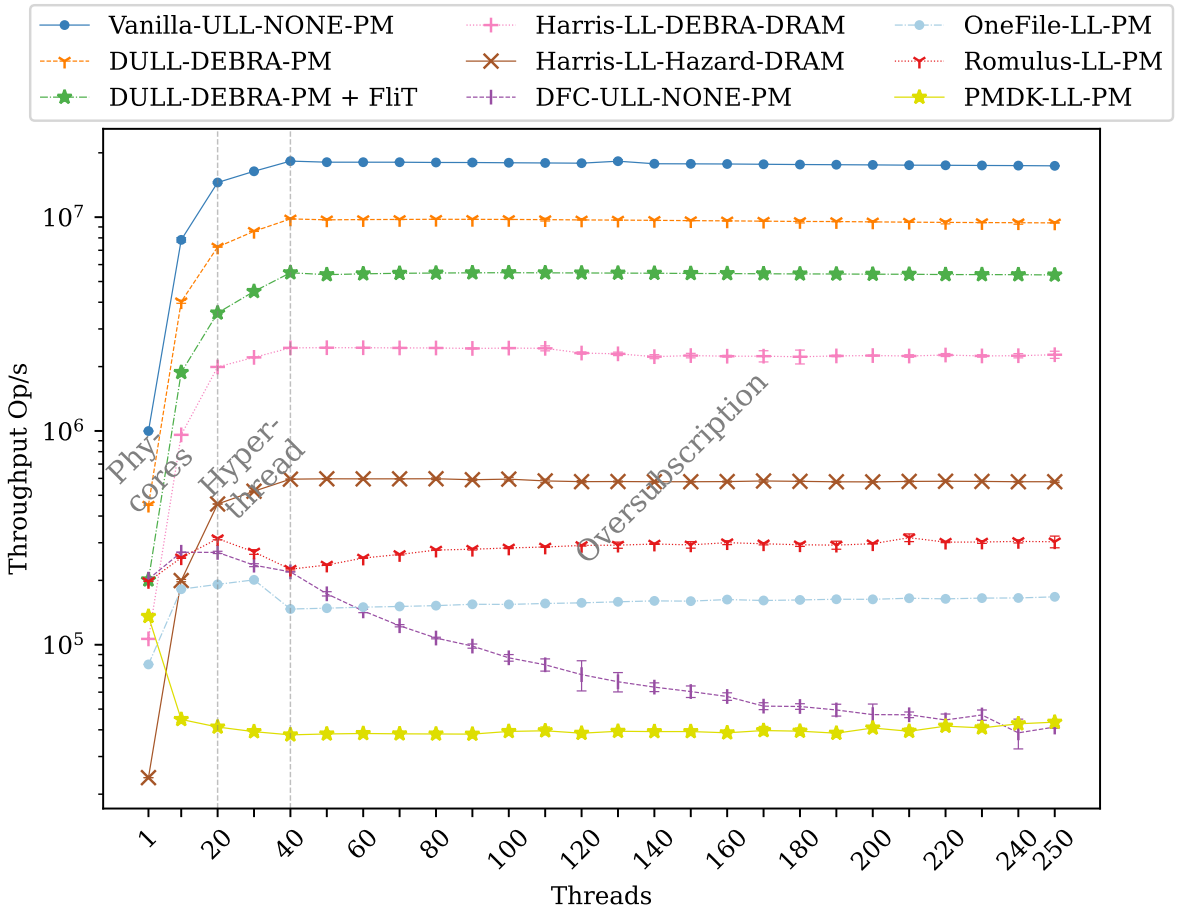


Figure 5.5: Balanced workload – key space 1000

LL in all scenarios. PMDK-LL is dominated by all algorithms except in all-read workloads, Figure 5.8, where it yields higher throughput than OneFile-LL. The reason is, for every node while traversing the list, OneFile-LL performs a more complex load *interposition* that involves two lock acquisitions and may search the write-set [65]. On the other hand, DULL, DFC-ULL, Romulus-LL and PMDK-LL employ simpler reads.

In hyper-threading mode, the throughput of DULL increases in high concurrency levels, unlike the other recoverable algorithms in all-write and balanced workloads. This shows the scalability of our proposed algorithm. The performance of DFC-ULL, Romulus-LL and OneFile-LL degrades when more hyper-threaded cores are used in update workloads. The

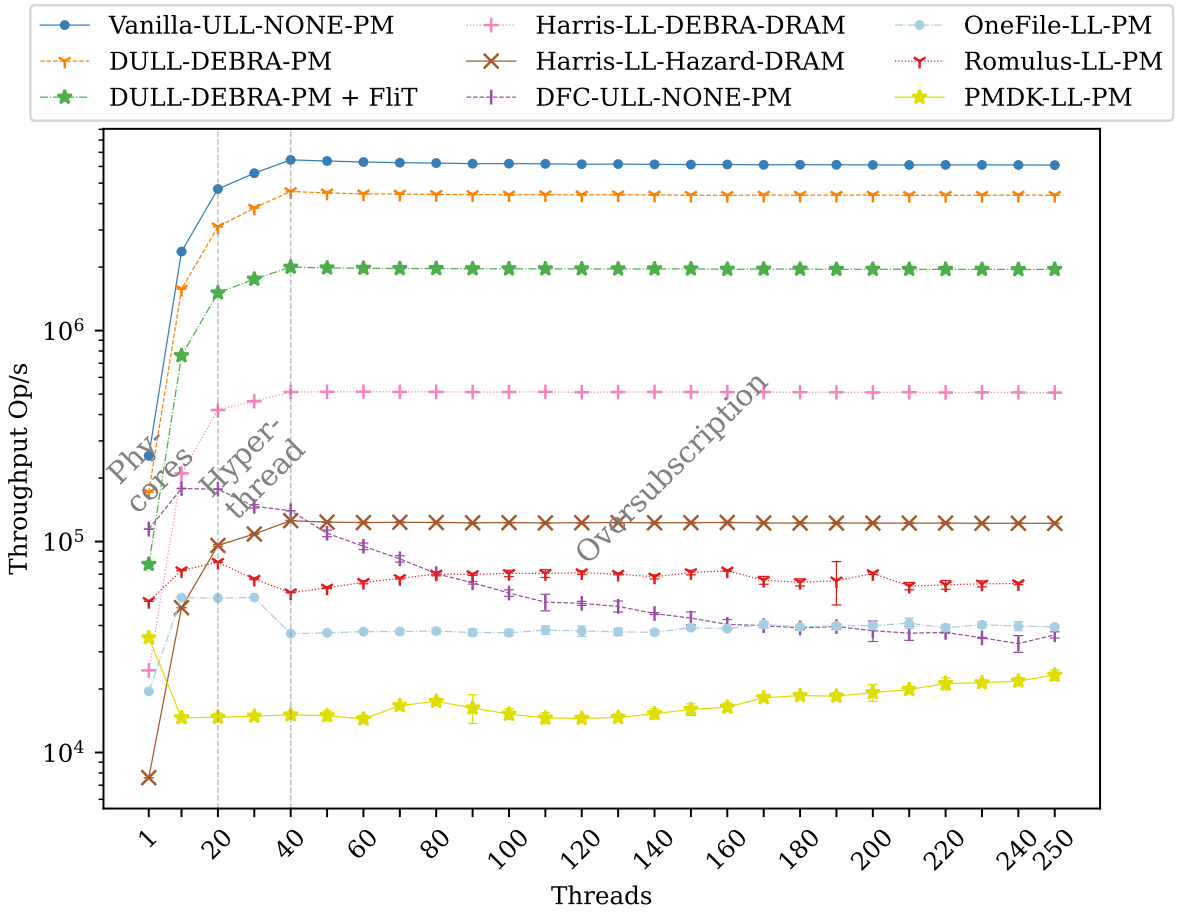


Figure 5.6: Balanced workload – key space 5000

DFC-ULL and Romulus-LL are flat-combining techniques. Therefore, more concurrent updates raise the global combining overhead, which prolongs blocking other threads. The OneFile-LL behaviour is expected as PTMs tend not to scale in scenarios of high degree of simultaneous updates. On the other hand, our proposed algorithm does not require adding locks, or transactions, beyond the ones required for the correctness of the non-recoverable variant. In DULL, I utilize the fact that the unrolled linked list is already lock-based to achieve detectability. Therefore, in high levels of concurrency (> 20 threads), DULL is significantly more scalable than the competition.

In over-subscription mode, the performance of DULL, Romulus-LL and OneFile-LL

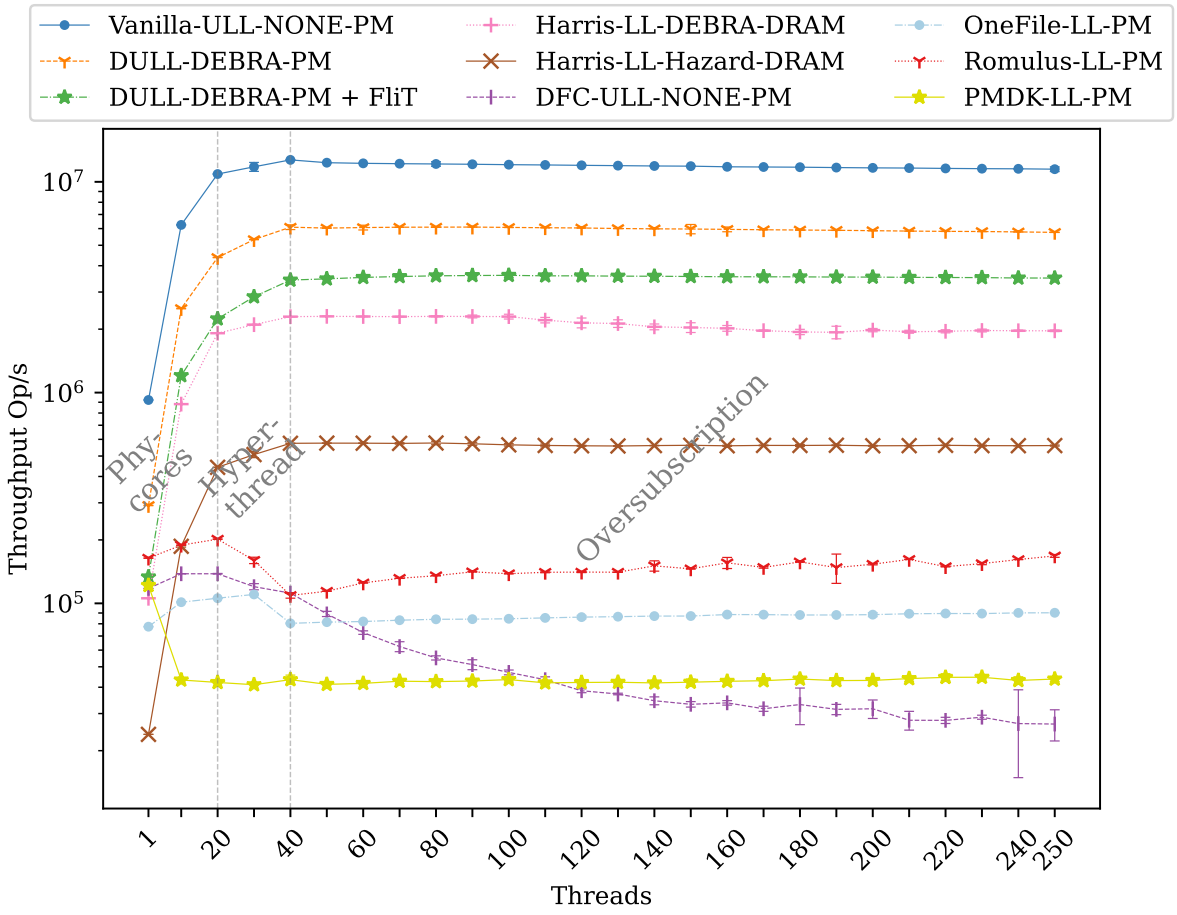


Figure 5.7: All-write workload – key space 1000

is shown to be stable and saturated as the number of threads increases in all-write and balanced workloads. The DFC-ULL performance, in contrast, drops significantly when the concurrency level rises. It was interesting at first to see that the DFC-ULL algorithm collapses in scalability compared to the Romulus-LL, although both algorithms are based on flat-combining. This observation can be explained as a result of the *reduction* step in DFC-ULL. In DFC, the combiner first examines the submitted requests by other threads and attempts to reduce the number of operations by *elimination*. For example, the combiner may resolve the result of each simultaneous pair of opposite operations (an insertion and a removal of the same key) without modifying the data structure. Such a technique is mostly advantageous when used in implementing *abstract data types* (ADTs) that have a

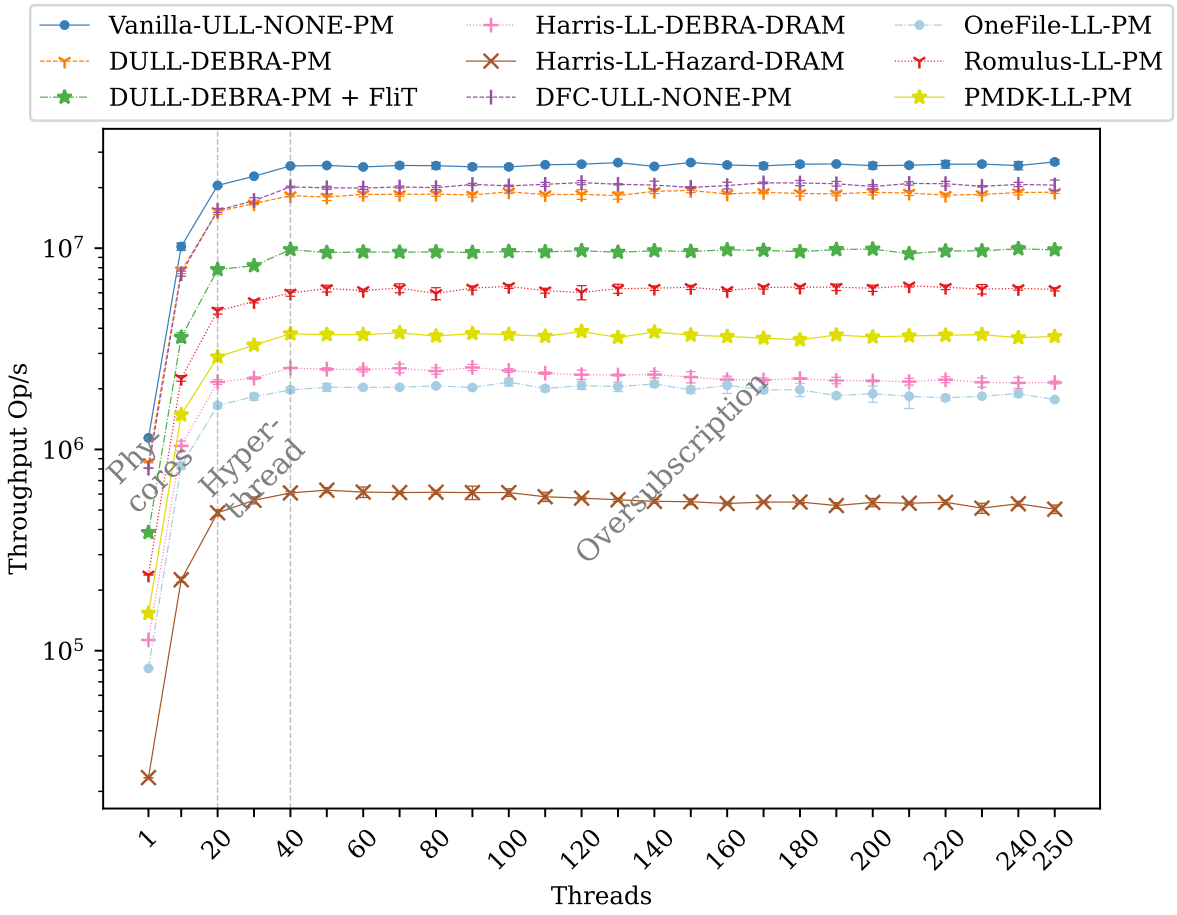


Figure 5.8: All-read workload – key space 1000

parameterless remove procedure, like stacks, queues and deques, where a removal can be matched and eliminated with any concurrent insertion. In contrast, for a correct set ADT implementation, like a linked list, a key removal can only be eliminated by a simultaneous insertion of the same key, which is improbable when the key space is larger than the number of threads.

The results considering all-read workloads are demonstrated in Figure 5.8. The overhead of adding recoverability while traversing DULL is found to be small when compared to the Vanilla-ULL, thanks to the efficient flush-on-read implemented as part of the *PMptr* and *PMpair* persistent objects. The same is noticeable for the DFC-ULL as well. In fact,

DFC-ULL is slightly faster than DULL since it does not add flush steps to the original traversing algorithm. The Romulus-LL, OneFile-LL and PMDK-LL are slower compared to the DULL and DFC-ULL in all-read workloads as a consequence of using the unrolled list as the underlying list algorithm.

Figure 5.6 illustrates the throughput considering a balanced workload and a key space of 5000 to assess and compare the list algorithms for a larger list size, roughly 2500 key-value pairs. When compared to balanced workloads with key space of 1000, Figure 5.5, a similar pattern with a consistent lower performance is observable in all algorithms, except for DFC-ULL. That is, the DFC-ULL is impacted the least when increasing the key space and list size. The fact that the degradation in performance of the DFC-ULL algorithm is barely noticeable shows that the DFC component is the limiting factor of the list scalability. Specifically, although a larger list slows down traversal, the flat-combining on updates is relatively more complex and, hence, is the performance bottleneck in the DFC-ULL algorithm. The results for all-write and all-read workloads for key space of 5000 are very similar to the results for key space of 1000. Therefore, such outcomes are omitted due to lack of space.

As Vanilla-ULL demonstrates the fastest performance among all the algorithms due to its unrolled nodes and lack of recoverability mechanisms, I used it as a baseline to evaluate the performance of each algorithm over different list sizes. Figure 5.9 illustrates the relative throughput of each list compared to Vanilla-ULL using 20 threads, a balanced workload consisting of 50% reads and 50% updates, and varying key spaces.

Since key insertions and removals are uniformly distributed across the key space, the list sizes are approximately half the key space. Based on the experimental results presented in Figure 5.9, I observe that Harris-LL achieves throughput nearly identical to Vanilla-ULL for smaller list sizes, specifically when the key space is 128, corresponding to approximately 64 list elements. DULL follows, achieving about one-third of the throughput of Vanilla-ULL and Harris-LL, primarily due to the additional overhead introduced by the recoverability and detectability mechanisms in DULL.

As the key space—and consequently, the list size—increases, the relative performance of Harris-LL compared to Vanilla-ULL declines sharply, as does DULL's. This decline is attributable to lock contention: with smaller lists, lock-based algorithms experience higher contention overhead when accessing locks. However, this performance bottleneck diminishes as list sizes grow. Therefore, when the key space exceeds 512, DULL outperforms all other algorithms, demonstrating at least a 2x improvement in throughput compared to Harris-LL and several-fold improvements over the remaining recoverable competitors.

In conclusion, this chapter addressed the second and third research questions by in-

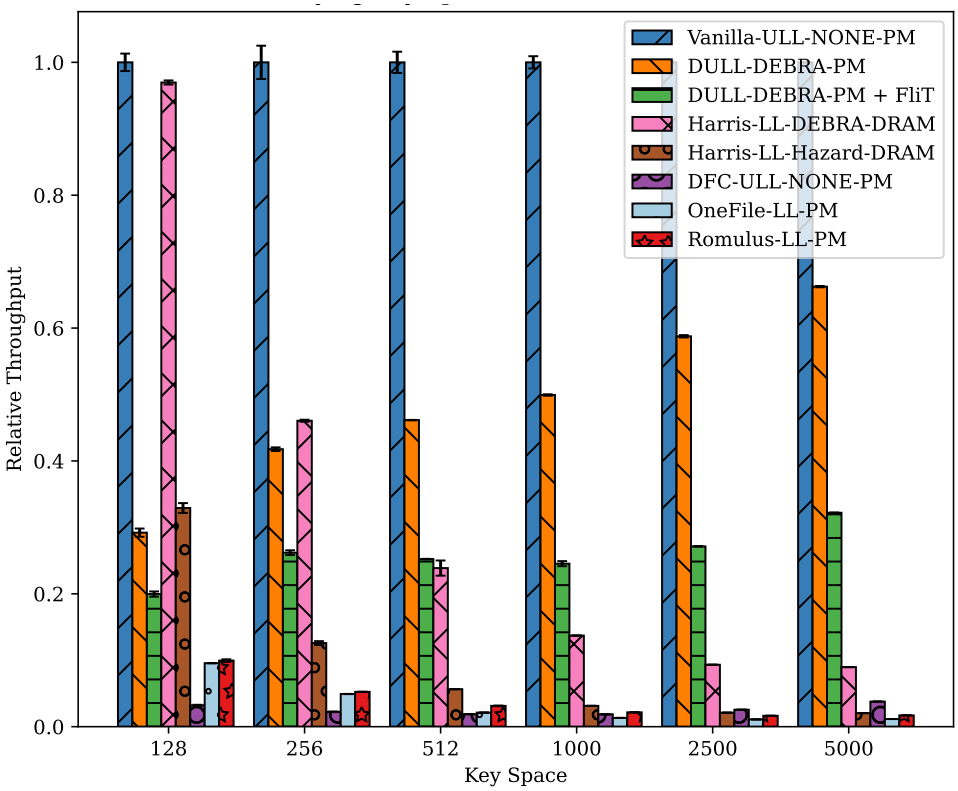


Figure 5.9: Throughput relative to the Vanilla-ULL - 20 threads - balanced workload.

troducing the Detectable Unrolled Lock-based Linked List (DULL), a high-performance, detectable lock-based linked list tailored for persistent memory environments. Interestingly, DULL, which employs fine-grained locking, was shown to outperform its lock-free competitors in most workloads. It achieved strict linearizability and demonstrated that volatile locks can add detectability to recoverable data structures, highlighting their performance benefits. Additionally, the applicability of RME to volatile locks was explored, showing that recoverability is not exclusive to persistent memory.

Chapter 6

Conclusion

This thesis encapsulates the contributions made towards addressing key challenges in recoverable synchronization algorithms and lock-based data structures, particularly within the context of persistent memory (PM) environments and non-uniform memory access (NUMA) systems.

The thesis aimed to tackle three research questions:

1. What is the performance impact of using recoverable mutual exclusion (RME) locks on modern multicore multiprocessor systems, and how can this performance be improved?
2. Does the current definition of the RME problem include volatile locks?
3. Can underlying locks in data structures be exploited to achieve detectability, and what impact does this approach have on performance?

In addressing the first question, the thesis introduced the *Recoverable Filter (RF) lock*, a novel solution that enhances the performance of RME locks in NUMA architectures. By transforming NUMA-oblivious RME algorithms into NUMA-aware versions, the work demonstrated how memory locality could be leveraged to improve performance, achieving up to 45% improvement in multi-socket configurations. The RF lock preserves correctness properties such as mutual exclusion and starvation-freedom while minimizing overhead, particularly in single-socket setups. This contribution directly answers the first research question by providing an approach to improve RME lock performance on multicore systems.

For the second and third research question, the thesis introduced the *Detectable Unrolled Lock-based Linked List (DULL)*, the fastest known detectable lock-based linked list designed for persistent memory environments. DULL achieved strict linearizability while maintaining high performance under update-intensive workloads. This work proved that underlying RME locks could be leveraged to add detectability to recoverable data structures, showcasing the performance benefits of this approach. In addition, the applicability of RME to volatile locks was explored through the demonstration that locks do not need to rely solely on persistent memory to achieve recoverability.

Building on the advancements presented in this thesis, there are several promising avenues for future research that can further enhance RME locks and detectable data structures in the theoretical and practical aspects.

One promising direction for future research involves a deeper analysis of the RF lock's performance characteristics. In this thesis, my analysis approached, but did not fully establish, an amortized Remote Memory Reference (RMR) complexity of $O(1)$ per process. In certain settings, however, I was able to demonstrate this complexity. Specifically, I observed that the amortized RMR complexity holds if the order in which processes execute Line 50 aligns with the order in which they access the critical section (CS), and if, whenever a process attempts to hand over the lock, there is another process on the same node concurrently seeking access to the RF lock. This observation naturally raises an intriguing question: Is it feasible to construct a NUMA-aware transformation that guarantees an RME lock with an amortized RMR complexity of $O(1)$ per process? Exploring this question could lead to significant advances in lock efficiency for NUMA systems, potentially paving the way for highly optimized, scalable synchronization primitives in modern multiprocessor environments.

A promising direction for future research lies in expanding the detectability techniques developed for DULL and generalizing them to support other lock-based data structures, particularly those with complex operations or hierarchical organization, such as trees and graphs. Detectability in structures like trees and graphs, where nodes are interconnected in non-linear ways, introduces unique challenges, as operations frequently require traversing multiple nodes and preserving structural invariants across various sections of the structure. To develop a generalized detectability approach for these data structures, innovative strategies would be needed to track the status of operations, coordinate locks across multiple hierarchical levels, and manage recovery from crashes in a way that preserves the data structure's integrity. A robust generalization would ideally consider both independent process failures, where individual operations can recover without impacting other processes, and system-wide failures, where all active processes must recover in a consistent state. This dual consideration would be essential in designing a detectability model that balances

resilience with efficiency, ensuring that both local and global recoverability are maintained across complex data structures. In addition, integrating detectable data structures into larger system frameworks, such as databases or real-time analytics platforms, could yield valuable insights into their practical performance, highlighting further opportunities for optimization and refinement.

Future research could also explore alternative persistent memory (PM) technologies, especially as modern data centers face unprecedented growth in data volume, prompting a transformative shift in computing architecture. In particular, servers are now increasingly outfitted with purpose-built accelerators to offload specialized tasks from CPUs, improving efficiency for diverse, data-intensive workloads. To support this architectural shift, the Compute Express Link (CXL) technology was developed by the CXL Consortium, enabling memory coherency between CPUs and attached devices. CXL not only facilitates resource sharing for heightened performance but also simplifies the software stack and reduces overall system costs, creating new opportunities for high-performance, cost-efficient memory solutions. The integration of CXL in distributed systems opens intriguing theoretical and practical research questions: How might system models for CXL-supported architectures be formalized to reflect the distinct memory hierarchies and coherency domains enabled by CXL? What formal definitions can capture the varying levels of *memory remoteness* associated with CXL's memory pooling and sharing features? Is there an intersection between PM and CXL paradigms that could unify their use in fault-tolerant, persistent systems? If such an intersection exists, what are the key research challenges, and how could they shape the design of recoverable and detectable data structures under these hybrid memory models? These questions could inspire a new generation of system architectures that leverage both PM and CXL, fostering innovative solutions for resilient, high-performance computing in the era of data-driven applications

In conclusion, this thesis advances the field of concurrent data structures and synchronization algorithms, with particular focus on recoverable and detectable lock-based systems in the context of persistent memory technologies. Through rigorous analysis and experimentation, this research addressed key questions related to enhancing performance and ensuring reliability in modern concurrent systems. The solutions presented not only demonstrate practical improvements in the efficiency and resilience of these systems but also establish foundational methodologies for integrating persistent memory with state-of-the-art concurrency techniques. Moreover, the approaches proposed in this work provide a solid basis for potential future exploration. The techniques and theoretical insights gained from this research open up numerous possibilities for extending these principles to more complex data structures, new memory technologies, and broader system architectures.

References

[1] Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. https://www.hpl.hp.com/techreports/2003/HPL-2003-241.pdf, 2003.

[2] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What consistency does your key-value store actually provide? In *Proceedings of the 6th International Conference on Hot Topics in System Dependability*, HotDep'10, pages 1–16, 2010.

[3] T.E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[4] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Tracking in order to recover: Recoverable lock-free data structures. https://arxiv.org/abs/1905.13600, 2019. arXiv preprint.

[5] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. Detectable recovery of lock-free data structures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'22, pages 262–277, 2022.

[6] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Nesting-safe recoverable linearizability: Modular constructions for non-volatile memory. In *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing*, PODC'18, pages 7–16, 2018.

[7] Hillel Avni and Trevor Brown. Persistent hybrid transactional memory for databases. *VLDB Endowment*, 10(4):409–420, November 2016.

[8] Ohad Ben-Baruch, Danny Hendler, and Matan Rusanovsky. Upper and lower bounds on the space complexity of detectable objects. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC'20, pages 11–20, 2020.

[9] Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'19, pages 253–264, 2019.

[10] Naama Ben-David, Michal Friedman, and Yuanhao Wei. Survey of persistent memory correctness conditions. https://arxiv.org/abs/2208.11114, 2022. arXiv preprint.

[11] Vibhor Bhatt and Prasad Jayanti. Specification and constant rmr algorithm for phase-fair reader-writer lock. In *Proceedings of the 12th International Conference on Distributed Computing and Networking*, ICDCN'11, pages 119–130, 2011.

[12] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing*, PODC'15, pages 261–270, 2015.

[13] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. High performance locks for multi-level numa systems. *ACM SIGPLAN Notices*, 50(8):215–226, January 2015.

[14] Milind Chabbi and John Mellor-Crummey. Contention-conscious, locality-preserving locks. *ACM SIGPLAN Notices*, 51(8), February 2016.

[15] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 29th ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'14, pages 433–452, 2014.

[16] David Yu Cheng Chan and Philipp Woelfel. Recoverable mutual exclusion with constant amortized rmr complexity from standard primitives. In *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing*, PODC'20, pages 181–190, 2020.

[17] David Yu Cheng Chan and Philipp Woelfel. Tight lower bound for the rmr complexity of recoverable mutual exclusion. In *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 533–543, 2021.

[18] CMU Parlay Group. Source Code for FliT: A Library for Simple and Efficient Persistent Algorithms. https://github.com/cmuparlay/flit, 2024.

[19] Joel Coburn, Adrian M. 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 *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, pages 105–118, 2011.

[20] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA'18, pages 271–282, 2018.

[21] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-free concurrent data structures. In *Proceedings of the 39th USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'18, pages 373–385, 2018.

[22] Sahil Dhoked, Wojciech Golab, and Neeraj Mittal. Modular Recoverable Mutual Exclusion Under System-Wide Failures. In *Proceedings of the 37th International Symposium on Distributed Computing*, DISC'23, pages 17:1–17:24, 2023.

[23] Sahil Dhoked and Neeraj Mittal. An adaptive approach to recoverable mutual exclusion. In *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing*, PODC'20, pages 1–10, 2020.

[24] Dave Dice and Alex Kogan. Compact numa-aware locks. In *Proceedings of the 14th EuroSys Conference 2019*, EuroSys'19, 2019.

[25] Dave Dice and Alex Kogan. Fissile locks. In *Proceedings of the 8th Networked Systems*, NETYS'20, pages 192–208, June 2020.

[26] Dave Dice, Virendra J. Marathe, and Nir Shavit. Flat-combining numa locks. In *Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'11, pages 65–74, 2011.

[27] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Transaction on Parallel Computing*, 1(2), February 2015.

[28] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 26(1):21–22, January 1983.

[29] Ahmed Fahmy and Wojciech Golab. A numa-aware recoverable mutex lock. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'22, pages 295–305, 2022.

[30] Ahmed Fahmy and Wojciech Golab. Brief announcement: A fast scalable detectable unrolled lock-based linked list. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'24, pages 73–75, 2024.

[31] Ahmed Fahmy and Wojciech Golab. DULL: A Fast Scalable Detectable Unrolled Lock-Based Linked List. In *Proceedings of the 28th International Conference on Principles of Distributed Systems*, OPODIS'24, pages 6:1–6:25, 2025.

[32] Ahmed Fahmy, Adrian Jendo, and Wojciech Golab. The illusive failure-atomic double-width compare-and-swap. In *Proceedings of the 6th Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems*, ApPLIED'24, page 1–5, 2024.

[33] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. Nvtraverse: In nvram data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'20, pages 377–392, 2020.

[34] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'18, pages 28–40, 2018.

[35] Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: Making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI'21, pages 1218–1232, 2021.

[36] Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the 36th ACM Symposium on Principles of Distributed Computing*, PODC'17, pages 211–220, 2017.

[37] Wojciech Golab and Danny Hendler. Recoverable mutual exclusion under system-wide failures. In *Proceedings of the 37th ACM Symposium on Principles of Distributed Computing*, PODC'18, pages 17–26, 2018.

[38] Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. *Distributed Computing*, 32(6):535–564, December 2019.

[39] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC'01, pages 300–314, 2001.

[40] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems*, pages 3–16, 2006.

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

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

[43] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, K. Keeton, and Patrick Th. Eugster. Nvthreads: Practical persistence for multi-threaded applications. *Proceedings of the 12th European Conference on Computer Systems*, 2017.

[44] IBM. Overview: Address space layout randomization. https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization, 2023. Accessed: 2024-03-22.

[45] Ramin Izadpanah, Christina Peterson, Yan Solihin, and Damian Dechev. Petra: Persistent transactional non-blocking linked data structures. *ACM Transactions on Architecture and Code Optimization*, 18(2), March 2021.

[46] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 427–442, 2016.

[47] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Distributed Computing*, pages 313–327, 2016.

[48] Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Optimal recoverable mutual exclusion using only fasas. In *Proceedings of the 6th International Conference on Networked Systems*, NETYS'18, pages 191–206, 2019.

[49] Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic rmr on both cc and dsm. In *Proceedings of the 38th ACM Symposium on Principles of Distributed Computing*, PODC'19, pages 177–186, 2019.

[50] Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Constant rmr system-wide failure resilient durable locks with dynamic joining. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'23, pages 227–237, 2023.

[51] Prasad Jayanti and Anup Joshi. Recoverable FCFS Mutual Exclusion with Wait-Free Recovery. In *Proceedings of the 31st International Symposium on Distributed Computing*, DISC'17, pages 30:1–30:15, 2017.

[52] Prasad Jayanti and Anup Joshi. Recoverable mutual exclusion with abortability. In *Proceedings of the 7th International Conference on Networked Systems*, NETYS'19, pages 217–232, 2019.

[53] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 586–599, 2019.

[54] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable numa-aware blocking synchronization primitives. In *Proceedings of the 38th USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'17, pages 603–615, 2017.

[55] Daniel Katzan and Adam Morrison. Recoverable, abortable, and adaptive mutual exclusion with sublogarithmic rmr complexity. In *Proceedings of the 24th International Conference on Principles of Distributed Systems*, OPODIS'20, pages 15:1–15:16, 2021.

[56] Nan Li and Wojciech Golab. Detectable Sequential Specifications for Recoverable Shared Objects. In *Proceedings of the 35th International Symposium on Distributed Computing*, DISC'21, pages 29:1–29:19, 2021.

[57] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 33rd USENIX Annual Technical Conference*, USENIX ATC'12, pages 65–76, June 2012.

[58] Victor Luchangco, Dan Nussbaum, and Nir Shavit. A hierarchical CLH queue lock. In *Proceedings of the 12th International Euro-Par Conference*, pages 801–810, 2006.

[59] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[60] Mohammad Moridi, Erica Wang, Amelia Cui, and Wojciech Golab. A closer look at detectable objects for persistent memory. In *Proceedings of the 4th Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems*, ApPLIED'22, pages 56–64, 2022.

[61] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A Periodically Persistent Hash Map. In *Proceedings of the 31st International Symposium on Distributed Computing*, DISC'17, pages 37:1–37:16, 2017.

[62] Kenneth Platz, Neeraj Mittal, and S. Venkatesan. Practical concurrent unrolled linked lists using lazy synchronization. *Journal of Parallel and Distributed Computing*, 139:110–134, 2020.

[63] Tiago Pramalhe. Onefile: Concurrent programming simplified. https://github.com/pramalhe/OneFile, 2023. Accessed: 2024-03-22.

[64] Z. Radović and E. Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA-9, pages 241–252, 2003.

[65] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN'19, pages 151–163, 2019.

[66] Matan Rusanovsky, Hagit Attiya, Ohad Ben-Baruch, Tom Gerby, Danny Hendler, and Pedro Ramalhete. Flat-combining-based persistent data structures for non-volatile memory. In *Stabilization, Safety, and Security of Distributed Systems*, pages 505–509, 2021.

[67] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *Proceedings of the 8th ACM Conference on LISP and Functional Programming*, LFP'94, pages 185–195, 1994.

[68] Karthik Srinivasan, Andy Rudoff, and Sudarshan Srinivasan. EADR: New opportunities for persistent memory applications. https://www.intel.com/content/www/us/en/developer/articles/technical/

eadr-new-opportunities-for-persistent-memory-applications.html, 2022. Accessed: 2022-12-09.

[69] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, 2011.

[70] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, pages 91–104, 2011.

[71] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *Proceeding of the 34th IEEE International Conference on Data Engineering*, ICDE'18, pages 461–472, 2018.

[72] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. Flit: a library for simple and efficient persistent algorithms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'22, pages 309–321, 2022.

[73] Jeffrey Xiao, Zheng Zhang, and Wojciech Golab. Benchmarking recoverable mutex locks. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'20, pages 583–585, 2020.

[74] Jae Heon Yang and Jams H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, March 1995.

[75] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages*, 3(128):1–26, October 2019.