RGLock: Recoverable Mutual Exclusion for Non-Volatile Main Memory Systems

by

Aditya Ramaraju

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Master of Applied Science

in

Electrical and Computer Engineering (Computer Software)

Waterloo, Ontario, Canada, 2015

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.

Aditya Ramaraju.

Abstract

Mutex locks have traditionally been the most popular concurrent programming mechanisms for inter-process synchronization in the rapidly advancing field of concurrent computing systems that support high-performance applications. However, the concept of recoverability of these algorithms in the event of a crash failure has not been studied thoroughly. Popular techniques like transaction roll-back are widely known for providing fault-tolerance in modern Database Management Systems. Whereas in the context of mutual exclusion in shared memory systems, none of the prominent lock algorithms (e.g., Lamport's Bakery algorithm, MCS lock, etc.) are designed to tolerate crash failures, especially in operations carried out in the critical sections. Each of these algorithms may fail to maintain mutual exclusion, or sacrifice some of the liveness guarantees in presence of crash failures. Storing application data and recovery information in the primary storage with conventional volatile memory limits the development of efficient crash-recovery mechanisms since a failure on any component in the system causes a loss of program data. With the advent of Non-Volatile Main Memory technologies, opportunities have opened up to redefine the problem of Mutual Exclusion in the context of a crash-recovery model where processes may recover from crash failures and resume execution. When the main memory is non-volatile, an application's entire state can be recovered from a crash using the in-memory state near-instantaneously, making a process's failure appear as a suspend/resume event. This thesis proceeds to envision a solution for the problem of mutual exclusion in such systems. The goal is to provide a first-of-its-kind mutex lock that guarantees mutual exclusion and starvation freedom in emerging shared-memory architectures that incorporate non-volatile main memory (NVMM).

Acknowledgements

This thesis wouldn’t have been possible without the expertise, guidance, and the incredible patience of my advisor Prof. Wojciech Golab. The precision and non-dualism in everything that he says and writes is something I have continuously strived to inculcate throughout the duration of my MASc program. I shall always remain as the greatest admirer of his vast knowledge and skills in many areas, among which his technical writing is a class apart. Few people can inspire interest and passion towards the subject of Distributed Computing better than he does. I am deeply indebted to him for mentoring, educating, employing, and in a way, tirelessly parenting me during the last two years that made a significant difference in my life and shaped me as a better individual. I sincerely doubt that I will ever be able to express my appreciation fully, but I owe him my eternal gratitude. Every academic and professional achievement I ever make, starting with RGLock, is only a dedication and tribute to him.

I owe my gratitude to Prof. Rajay Vedaraj at VIT University for all the knowledge sharing, constant encouragement, and inspiration he provided ever since my undergrad days. My sincere thanks to my friend Kartik Josyula for motivating me to pursue higher studies in Canada, and special thanks to Ankur Agarwal and Krunal Barot, my supervisors at IBM India, for duly encouraging me in this regard. I am also very thankful to my close friends, particularly Youcef Tebbal and Nicholas Petroff, for making my time in Waterloo so wonderful with as many thought-provoking discussions as the numerous mindless debates we ran into. I must also acknowledge the important role of my brother, Ramakrishna Rao Ramaraju, in being the primary source of inspiration for everything I have ever learnt about computers, not to mention the readily available financial assistance he provided whenever required.

Finally, I am ever grateful to my parents for all their support and the sacrifices they have made for providing me with high quality education at only the best schools in every stage, of which, home remains my most favorite.

Dedication

To Annaya,

and

To my Guru.

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Mutex locks have traditionally been the most popular concurrent programming mechanisms for inter-process synchronization in the rapidly advancing field of concurrent computing systems that support high-performance applications. This popularity can largely be attributed to their simplicity in design and ease of implementation. The primary objective of a mutex lock is to avoid simultaneous use of shared data by labelling certain fragments of code as critical sections. Abundant contributions were made to the research work in this field with interests ranging from defining strategies that are correct by design and have good performance, to designing sophisticated fine-grained locks which enable highly concurrent access to shared data by avoiding a serialization of non-conflicting operations. However, the concept of recoverability of these algorithms in the event of a crash or a failure has not been studied thoroughly.

Several techniques have been proposed to continuously monitor the process state in a system and automatically restart a failed process [1], [2]. Popular techniques like transaction roll-backs have been widely known for providing fault-tolerance in modern Database Management Systems. Whereas in the context of mutual exclusion, none of the prominent existing lock algorithms (e.g., Lamport's Bakery algorithm, MCS lock, etc.) are designed to tolerate crash failures, especially in operations carried out in the critical sections. Each of these algorithms may fail to maintain mutual exclusion, or sacrifice some of the liveness guarantees in the presence of crash failures. Every time the system experiences a crash, the facts that a loss of program data is incurred as any unsaved user data and application settings in the volatile memory (DRAM) are lost, and that recording recovery information in disk storage for persistence incurs unacceptable overheads and performance degradation on the application, have been the primary obstacles in designing efficient crash-recovery systems. However, we have reasons [3] to believe that in the future, the Non-Volatile Main Memory (NVMM) systems can provide ways and means to overcome this problem.

With the advent of Non-Volatile Main Memory technologies, opportunities have opened up to redefine the problem of Mutual Exclusion in the context of a crash-recovery model where processes may recover from crash failures and resume execution. When the main memory is non-volatile, an application's entire state can be recovered from a crash using the in-memory state near-instantaneously, making a process's failure appear as a suspend/resume event. While the implementation techniques specific to automatically restarting failed processes and bringing them to resume execution from the point of last-known configuration lie beyond the scope of our work, we proceed to envision a solution for the problem of mutual exclusion in such systems. In this thesis, our goal is to provide a first-of-its-kind mutex lock that guarantees mutual exclusion and starvation freedom in emerging shared-memory architectures that incorporate non-volatile main memory (NVMM).

1.1 Preliminaries

Concurrency in a modern multi-processor system allows multiple processors to access a common resource, while mutual exclusion guarantees that only one of the contending processors gains an exclusive access to the shared resource. Dekker's algorithm [4] was the first software solution for a 2-process mutual exclusion. The problem of n-process mutual exclusion was first formulated by Dijkstra in [5] and later formalized by Lamport in [6]. A *race condition* arises when any two concurrent processes simultaneously modifying the value of a shared variable can produce different outcomes, depending on their sequence of operations. To avoid such conflicts, the program contains *Critical Section* (CS), a block of code that can be executed by only one process at a time. Formally, *Mutual Exclusion* (ME) is the problem of implementing a critical section such that no two concurrent processes execute the CS at the same time.

Practical algorithms [7]–[10] for mutual exclusion traditionally have a strong reliance on the Read-Modify-Write instructions, where a value from a shared variable is read and updated in an indivisible action called an *atomic step*. Generally, processes are required to acquire a *mutex*, a mutual exclusion lock, to access the shared resource protected by the CS. Each process *acquires* the lock by executing a fragment of code called the *entry protocol*. The entry protocol may contain a wait-free block of code called the *doorway*, which the process completes in a bounded number of its own steps [11]. If the mutex is already being held by another process, *busy-waiting* is performed by a technique called *spinning*, in which the process repeatedly checks on a shared variable to see if a pre-defined condition is true. After completing the CS, the lock-holding process *releases* the lock by executing an *exit protocol* (EP), wherein one or more of the other contending processes are notified that the CS can now be entered. Actions that do not involve the protected shared resource are categorized under *non-critical section* (NCS). Any variables accessed in the entry and exit protocols, except the process's program counter, may not be accessed when the process is either in the CS or NCS. A *concurrent program* is thus defined as a non-terminating loop alternating between critical and non-critical sections. A *passage* is a single iteration of such loop consisting of four sections of code in a concurrent program with the following structure:

```
while true do
    NCS;
    Entry Protocol;
    CS;
    Exit Protocol;
od
```

1.2 Background & Motivation

1.2.1 Spin-lock algorithms

In a simple variant of a mutual exclusion algorithm, a process attempts to acquire the lock by repeatedly polling a shared variable, e.g., a Boolean flag, with a `test-and-set` instruction; and releases the lock by changing the bit on the flag. The key to these algorithms is for every process to spin on a distinct locally-accessible flag variable, and for some other process (the lock-holder) to terminate the spin with a single remote write operation at an appropriate time. These flag variables may be made locally available either by allocating them in the local portion of the distributed shared memory, or through coherent caching [12]. Memory references that can be resolved entirely using a process's cache (e.g., in-cache reads) are called *local* and are much faster than *remote memory references* (RMRs), the ones that traverse the processor-to-memory interconnection network (*interconnect*, for short). Protocols based on test-and-set are designed [13] to reduce contention on the memory and the interconnect when the lock is held, particularly in cache-coherent machines, but a system of N competing processes can still induce $\mathcal{O}(N)$ remote memory references each time the lock is freed. An alternative solution is to delay or pause between polling operations, and an exponential backoff was found [14] to be the most effective form of delay. In a "ticket lock" [15], a process acquires the lock by executing a `fetch_and_increment` instruction on its *ticket*, a request counter, and busy-waits until the value is equal to the lock's release counter. The release counter is incremented when the process releases the lock. Since a *first-come-first-serve* (FCFS) order is ensured by these counters, the number of `fetch_and_Φ` operations [16] per lock acquisition can be effectively reduced if the backoff is suitably designed to a delay that is proportional to the difference between the requester's ticket and the lock's release counter.

The simplest spin locks are generally considered to involve high contention on a single cache line and are thus poorly scalable [14]. This issue is addressed by the queue locks. Therefore, the idea of queue-based mutual exclusion algorithms has been particularly appealing to researchers in concurrent programming for over two decades. Existing literature contains several algorithms [7], [8], [17]–[22] proposed in this context, and one of the common underlying features in their lock design is a queue-like shared data structure. Fundamentally, in a queue-based lock algorithm, the contending processes "line up" in a *queue*, which is essentially a sequence of processes already busy-waiting in line for the lock, to access the critical resource, while only the head of the queue may enter the critical section. The advantage of this approach is that it ensures a FCFS order in lock acquisition and release. Each process trying to acquire the lock leaves the NCS and enqueues itself at the end of the queue. If it is not the head of the queue, then it links behind its *predecessor* in the queue and busy-waits on a local spin variable until it acquires the lock. A lock-holding process dequeues itself from the head of the queue after executing the critical section and signaling its immediate *successor* in the queue (if exists) to stop waiting and proceed to the CS.

The algorithm used for providing synchronization has a key impact on the performance of applications running in multi-threaded or multi-processor environments. Particularly in multi-processor environments, choosing the correct type of synchronization primitive and the waiting mechanism used for the synchronization delays is even more important [23]. In 1991, Mellor Crummey and Scott proposed the MCS lock [15] which generates $\mathcal{O}(1)$ remote memory references per lock acquisition on machines with and without coherent caches, independent of the number of concurrent processes that are trying to acquire the lock, and requires only a constant amount of space per lock per process. Their lock gained one of the most widespread usage and prominence in the multiprocessor computing community. For instance, the MCS lock is known to dramatically improve the performance of Linux kernel due to its scalability [24]. The key innovation in MCS lock is ensuring that each process spins only on locally-accessible locations, i.e., locations that are not target of spinning references by any other processes, making it most resilient to contention [25]. The MCS lock is hardware assisted, requiring an atomic `fetch_and_store` (`FAS`) instruction for the lock acquisition protocol and makes use of a `compare_and_swap` (`CAS`) instruction to ensure the FCFS order in the lock release protocol. A FAS operation exchanges a register with memory, and a CAS compares the contents of a given memory location against the value at a destination and sets a conditional to indicate whether they are equal. Only if both the values are equal, the contents of the destination are replaced with a second given value.

The pseudo-code for the MCS list-based queueing lock is shown in Algorithm 1. In a high-level overview, each process trying to acquire the MCS lock allocates a record called *qnode* containing a queue link (`next` pointer) and a Boolean flag. Portions of the qnode record may be made locally available on a cache-coherent multiprocessor. Processes holding or waiting for the lock are lazily chained together in a queue by the queue links, i.e., each process in the queue holds the address of the record for the process immediately behind it in the queue – the process it should notify when releasing the held lock. The lock itself contains a pointer to the record of the process that is at the tail of the queue if the lock is held, or to `null` if the lock is free. The `fetch_and_store` operation constitutes the doorway instruction, in which, every process trying to acquire the lock swaps the tail pointer of the queue to its own qnode and adds its own qnode to the end of the queue. The swap operation returns the previous value of pointer L. If that pointer is null, then the process knows that it has succeeded in acquiring the lock. Once the CS is executed, the process either sets the lock free if there is no contention for it, or *passes on* the lock to its immediate successor in the queue. So to release a lock, the lock holder must reset the Boolean flag of its successor, or if there is no successor, then L is set to null atomically in the CAS operation. The MCS lock thus provides an FCFS order by passing the lock to each process in the order that the processes enqueued. The local spin in the `acquire_lock` procedure waits for the lock ownership, while the local spin in `release_lock` compensates for the timing window between the `fetch_and_store` operation and the pred.next := I assignment in the `acquire_lock` procedure.

```
type qnode = record
   next : *qnode                                  //pointer to successor in queue
   locked : Boolean                               // flag for busy-waiting
type lock = *qnode                                //pointer to tail of queue
// I points to a qnode record allocated in the locally accessible
//shared memory location of the invoking process.
procedure acquire_lock  (L: *lock, I: *qnode)
    I.next := null                                //initially no successor
    pred : *qnode := FAS (L,I)                    //queue-up for lock (doorway)
    if  pred ≠ null                               //queue was non-empty
        I.locked := true                          //prepare to spin
        pred.next :=  I                           //link behind predecessor
        repeat while I.locked                     //busy-wait for lock

procedure release_lock (L: *lock, I: *qnode)
     if  I.next := null                           //no known successor
        if compare_and_swap (L, I, null)
              return                              //no successor, lock free
        repeat while I.next = null                //wait for successor
    I.next.locked := false                        //pass lock
```

Algorithm 1 MCS Lock

1.2.2 Crash-recoverable Mutex

Fault-tolerance is one of the most important issues to take into consideration in designing modern asynchronous multiprocessor systems, since it is aimed at guaranteeing continued availability of the shared data even if some of the processes in the system experience crash failures due to reasons such as system crash, power loss, accidental or intentional termination, heuristic deadlock recovery mechanisms, etc. Each failure results in a loss of the shared state stored in the local volatile memory of the failed process. A process that crashes in a *crash-stop failure model* permanently stops the execution of its algorithm and is supposed to never recover. However, in a *crash-recovery model* a failed process may be resurrected after a crash failure, and hence the problem of *crash-recovery* is to restore the lost state in such a way that the whole memory remains in a consistent state. Existing research consists of several crash-recovery techniques proposed for general distributed systems in the message passing paradigm [26]–[29] that rely on recovering the state information of a crashed process either from its stable secondary storage or from a process on a different node. Techniques proposed for crash-recovery in the distributed shared memory (DSM) and cache-coherent (CC) models can be classified into two categories: the first consisting of techniques relying on checkpointing [30]–[33], and the second approach being message logging [34], both originally developed for message-passing systems [35]. However, such techniques are poorly suited for the architectures that emphasize volatile main memories such as SRAM-based caches and DRAM-based memories [36], because the frequent accessing of a non-volatile secondary storage to save the current state of the computation (e.g., a checkpoint) incurs significant overheads and performance degradation in the system's efficiency.

The possibility of sudden loss of data from the portion of the shared memory protected by the critical section in the event of a crash failure is problematic, since it could result in violation of the safety and liveness properties guaranteed by the mutual exclusion algorithm. In particular, none of the prominent mutual exclusion algorithms (e.g., Peterson's algorithm, Lamport's Bakery algorithm, or MCS lock) is fault-tolerant, since each of these algorithms may fail if the state of the shared variable used for communication among the processes is lost in a crash failure. To that end, these algorithms fail even if processes are executing entry and exit protocols all by themselves, i.e., without any contention. Without mechanisms for detecting and recovering from crashes, conventional algorithms may lock a shared object indefinitely in presence of failures. Therefore, a 'crash-recoverable mutual exclusion lock' is needed to solve this problem. Among the earlier attempts in this regard [37]–[41], none of the proposed solutions are immune to a violation of at least one of the progress properties conventional locks guarantee in the absence of failures. Motivated by these observations, this thesis aims at investigating a recovery protocol which would overcome the limitations of existing solutions by guaranteeing each of mutual exclusion, fairness, starvation freedom and fault-tolerance in presence of failures in a crash-recovery model.

As a first step towards that solution, we consider the emerging shared memory architectures that incorporate a non-volatile main memory (NVMM), which has a promising potential to change the very fundamental approach to the 40-year old architectures with fast/volatile and slow/non-volatile levels of storage. NVMM systems can be built using a variety of media including phase-change memory (PCM) [42], memristors [43], magnetoresistive RAM (MRAM) [44], and ferroelectric RAM (FeRAM) [45], to name a few. The operating systems research community corroborates [3] that the architecture option that entirely replaces DRAM in the current architectures with only a single non-volatile RAM (NVRAM) in a flat, non-volatile physical memory space could prove to be the most advanced alternative to the conventional CPU, DRAM and disk design. NVMM promises to combine the benefits of the high speed of static RAM (SRAM), the density of dynamic RAM (DRAM) and the non-volatility of flash memory [46]. The idea of non-volatile memory as a primary storage inspires us to rethink several aspects of how users and programmers interact with applications in a concurrent setting. Particularly, this creates a possibility to revisit the problem of mutual exclusion in a crash-recovery model.

Since all execution state can be dissociated from process crashes and power failures by storing it on a persistent non-volatile medium, this motivates our research for a new recoverable mutual exclusion algorithm dedicated for systems with NVMM. The assumption is that the concurrent datastructures such as stacks, queues, and trees used by the mutex lock reside directly in the NVMM and that the lock can access them using conventional synchronization techniques. Accordingly, we investigated the dynamically scalable, contention-free MCS lock as a potential solution for the recoverability problem. Despite all the virtues the MCS lock algorithm has when implemented using the contemporary architectures, we found that the "out of the box" protocol of MCS lock as seen in Algorithm 1, is not an end solution for a crash-recovery model even when augmented with persistent shared memory to store the lock queue. That is, the lock could become unavailable if one or more of the contending processes crashes while holding the lock or while waiting in a queue to acquire the lock, since each process loses its private state when it resumes execution after the crash, i.e., when it recovers. In particular, even if the qnode records in the persistent shared memory are preserved across the crash, the process may not resume the program from its last point of execution since the state of the program counter is lost in the crash. For that reason, executing the algorithm from the beginning each time a process recovers from a crash causes certain violations of the liveness and safety guarantees as discussed below. On taking a closer look, the inadequacy of the original MCS lock algorithm in a crash-recovery model can be attributed to the following reasons:

i. Each process executing the MCS lock algorithm dynamically constructs a lock acquisition queue using the `FAS` instruction on the lock pointer L. Since the returned value of the `FAS` operation is only held in a private variable (`pred`), a process recovering from a crash that occurred immediately after the `FAS` may have no evidence of having ever entered the lock queue since all the private state of that process is lost during the crash.

ii. As a consequence of (i), a process may recover from its last crash and yet remain oblivious of the ownership of the lock. Consequently, even if that process is in the critical section as the lock-holder, it may try to enter the lock queue again by performing the FAS operation. This compromises the integrity of the queue structure.

iii. Furthermore, a process may execute the *acquire_lock* procedure again when it recovers, although it crashed while executing the *release_lock* procedure, and thus never relinquishes its ownership of the lock.

iv. As a consequence of (iii), other processes waiting in the queue in their own *acquire_lock* procedures may wait forever for a lock that will never be released. Thus, the progress of the other active processes in the system is impeded.

v. If a process crashes after entering a non-empty queue and before linking behind the previous tail of the queue by completing the pred.next := I instruction, then its immediate predecessor in the queue can never pass on the lock since the lock-holder is hindered in its operation of flipping the Boolean flag 'locked' on the crashed process's qnode record until the 'next pointer' link is complete. And as a consequence of (i), the crashed process can never complete the link when it recovers, since its immediate predecessor in the queue just before it crashed is not known.

vi. If a process crashes just after it completes the pred.next := I instruction and recovers after its predecessor has relinquished the lock by completing the last line of the *release_lock* procedure in its own passage, then the recovered process may execute the I.locked := true instruction again since it has no evidence of having already completed that instruction before its last crash, and thus never becomes the lock-holder.

It is obvious from the above description that the MCS lock algorithm cannot guarantee safety and liveness simultaneously in a crash-recovery model when implemented "out of the box". Therefore the task that lies ahead of us is to modify the algorithm so that the integrity of the queue structure is maintained even in the event of multiple crash failures in any number of the contending processes. Informally, a crash-recoverable queue lock must ensure the following:

- No process’s queue entry is lost in the crash. Therefore, no process in the system should starve due to a crash.

- Each process contains at most one instance of its record in the lock queue.

- At most one process owns the lock. Also, at most one process at a time believes it is the lock-holder.

- If a lock-holder crashes, then it should not lose the ownership when it recovers from the crash.

- No process should wait indefinitely to relinquish its lock ownership.

1.3 Summary of Contributions

In an effort to provide a crash-recoverable mutex, we present our queue-based design inspired by the popular MCS lock [15]. The research contributions in this thesis include:

1. A formal specification of the problem of ‘Recoverable Mutual Exclusion’ in a crash-recovery model.

2. RGLock: a first-of-its-kind spin-lock algorithm that is recoverable by design and preserves the properties of recoverable mutual exclusion in systems that incorporate NVMM.

3. A comprehensive proof of correctness for the specified algorithm.

Chapter 2

Related Work

Mutual exclusion remains as one of the most important and widely-studied problems in concurrent programming and distributed computing in general. In 1986, Michel Raynal published a comprehensive survey [47] of existing research outcomes for solving the problem of mutual exclusion in parallel or distributed computing. James Anderson et al. supplemented this survey in 2003 [10] by surveying major research trends in the mutual exclusion algorithms specific to the shared memory model since 1986. They highlight the limitations of earlier shared-memory algorithms that result in performance degradation due to the excessive traffic generated on the interconnect network, and discuss how local-spin algorithms published since 1986 avoid such problems. Their survey also notes how Lamport [48] became a trend-setter for the "fast" algorithms after his publication in 1987 [49]. Other major trends surveyed include a broader study of the "adaptive" algorithms [50] that ensure only a gradual increase in their time complexity in a proportional manner as the contention increases; the "timing-based" algorithms that exploit the notions of synchrony to reduce time complexity; and the mutual exclusion algorithms with "non-atomic" operations [51]. More recently in 2014, Buhr et al.[52] examined the correctness and performance of N-thread implementations of the high-performance mutual exclusion algorithms published in over 30 years, along with a meticulous interpretation of how each of these algorithms work and the intuition behind their design. These surveys inspire much more ambitious work to come in the future years and serve as the strongest primers for the topic of mutual exclusion in the shared memory distributed computing literature.

The idea of fault-tolerant computing has existed for 50 years now with one of the first publications on the subject dating back to 1965 [53]. Randell's publication in 1975 [54] laid the foundations for facilitating software fault-tolerance by introducing the "recovery block" scheme, in which a programmer can exploit the knowledge of the functional structure of the system to achieve software error detection and recovery. Recovery-oriented computing has become a fundamental requirement in most modern applications, and the strong consistency guarantees it offers are often serviced by transactional databases with centralized checkpoints and write-ahead logging (WAL) or shadow-paging mechanisms [34] that leverage sophisticated optimizations on disk-based persistent storage. While database tables and indexes are classical long-term storage solutions, many enterprises have been increasingly adopting in-memory data management [55] for their transactional and analytical processing needs to offer high performance computing, ascertaining that ‘in-memory’ is the new disk. However, the in-memory data structures used for synchronization are generally designed for scale-out and performance over fault-tolerance. For instance, most in-memory databases rely on a cluster-architecture to avoid a single point of failure, since a stand-by node in the cluster rolls back the incomplete transactions on the crashed process and replays the transaction logs replicated from the disk or network storage anyway[56]. Therefore, either non-blocking data structures such as lock-free [57], wait-free [57] and obstruction-free [58] implementations, or transactional memories such as [60], [61] are the usually preferred design choice in these systems since they offer resilience against crash-stop failures.

Designing fault-tolerant algorithms for unreliable memories has found an increasing interest in academia over the last four decades [62]. In 1977, Lomet first proposed synchronization and recovery using atomic actions and showed how atomic actions can be used to isolate recovery activities to a single process [63]. In more recent literature on shared-memory computing, notable contributions were made for making algorithms resilient to failure [31], [64]–[67] relying on techniques like coordinated global checkpointing and rollback recovery from centralized logs. In [68], Molesky and Ramamritham proposed crash-recovery protocols for making locks recoverable in shared memory databases. In the context of mutual exclusion, Bohannon et al. [40], [41] pioneered the research aimed at providing fault-tolerance in standard concurrency control mechanisms in shared-memory computing environments. The concept of making a spin lock ‘recoverable’ in shared-memory models is severely limited by the lack of methods to persistently save information regarding the ownership of the spin lock when a process acquires it so that the ownership information is potentially useful in restoring the shared resource protected by the spin-lock to consistency in the event of a crash failure.

In [40], the authors emphasize on the importance of processes registering their ownership of the spin lock (a test-and-set based algorithm) by writing the process identifiers to a known location in the shared memory. The key idea is to take a global picture of the entire memory instead of a local (i.e., per-process) one, which enables an examination of *all* processes that may have tried to acquire the lock before a crash failure so as to give enough information about the ownership of the spin lock. A “cleanup_in_progress” flag is used, which when raised, prevents any processes that have not registered their interest in the lock before the crash failure occurred from acquiring the lock while the recovery actions are being taken, and then an “overestimation snapshot” of processes that could already have or could get the lock during the recovery is taken. And within the “cleanup” routine the system waits for the situation to resolve by itself, i.e., eventually the ownership of the lock becomes known either because the lock is registered with a live process, or because no process holds it any longer, i.e., the lock is free.

In [41], the authors present a Recoverable MCS lock (RMCS-lock), in which each per-process qnode structure contains a ‘next’ pointer for forming the queue link, and a ‘locked’ field that indicates whether a given process owns the lock. The lock itself consists of a single pointer ‘tail’ which either points to the tail of the queue or to null when the lock is free. RMCS-lock comes with a ‘cleanup’ activity in which the application can query the operating system whether a particular process has died. Their deviation from the original MCS lock lies in the allowed values for the ‘locked’ field viz., WAITING, OWNED or RELEASED, each of which is set by the owning process depending on which line of code in the acquire or release protocols it executed. Further, they augment each qnode with the following fields: ‘wants’, a pointer to the lock a process sets before executing the entry protocol and is changed only after it releases the lock; ‘volatile’, a Boolean flag a process raises before modifying the queue structure (e.g., swap operation) and resets only when the modification is complete; ‘cleanup_in_progress’, a Boolean flag writable by the system/OS during cleanup activities; and ‘clean_cnt’, an integer that determines if the cleanup activity has completed. The recovery mechanism is similar to that shown in [40], where the ‘cleanup’ routine prevents new processes from entering the lock’s queue when the cleanup_in_progress flag is set and waits until each process that already set their ‘wants’ flag before the crash to acquire and release the lock. The cleanup routine also checks for the consistency of the queue structure and adds the ‘next’ links where they are missing due to failed processes.

In [37], Michael and Kim presented a fault-tolerant mutual exclusion lock that guarantees recovery from process failures in DSM systems using techniques to enable an active process or thread waiting to acquire a lock to “usurp” it in case it determines that the previous lock-holder has crashed. Their recovery mechanism relies on the programming environment (or the OS) maintaining a log of the status of all processes whose failure may lead to a permanent unavailability of the lock. The lock queue contains information on three shared variables, namely Head, Tail and Lock-Holder. The Lock-Holder field is used as a backup for Head, and identifies whether a process died in its lock release protocol before completely releasing the lock. Each process’s qnode record contains four fields: Process ID, Status, Next and LastChecked. The Status field has one of the three values in HASLOCK, WAITING, or FAILED. If a given process is determined to have crashed when some other process queries the programming environment, then the crashed process’s Status is updated to FAILED. The Next field is a pointer for queue formation.

The LastChecked field is updated by the owning process itself and records the last time the process was alive, an optimization to limit the OS queries, which are expensive with respect to system resources. In order to prevent multiple processes attempting recovery simultaneously, the recovery mechanism is designed in such a way that if the first N processes in the queue crashed, then only the N+1th process may execute the recovery routine and if successful, becomes the new lock-holder, or the N+2nd process eventually 'usurps' the lock after a certain pre-determined 'timeout' has passed.

In comparison to our goals for a crash-recoverable mutex (as defined in 1.2.2), we found the following shortcomings in the recovery mechanisms proposed in the above approaches:

1) The assumption that the application can query the OS whether a particular process has died requires a separate central monitoring process or thread that never crashes, e.g., a fault-tolerant lock manager [69].
2) Requiring a single process or thread to perform all the required recovery actions is an inefficient design choice in large non-homogeneous systems and might not be feasible at all since not all processes have the same capabilities and a single process that never crashes and is also capable of performing all recovery actions, possibly multiple times, often does not exist [70].
3) The 'cleanup' routine removes all crashed/dead processes from the queue. This poses a serious problem particularly in a crash-recovery model where processes may resume execution after recovery, and a process holding or waiting for a lock is not only removed from the queue, but also is exempted from acquiring the lock until the 'cleanup' is complete.
4) Processes get killed in the 'cleanup' if they do not make progress in a "reasonable" amount of time. Particularly, in asynchronous environments, an active process may be removed from the queue, possibly many times.
5) In case a process dies after it has been included in the queue by the 'cleanup' routine itself, it necessitates a subsequent run of yet another 'cleanup' routine.
6) Any live processes that did not register their interest in acquiring the lock before the crash failure are stalled until the 'cleanup' is complete. If there are multiple failures occurring frequently, some processes in the system may be perpetually denied from acquiring the lock i.e., the concurrency mechanism is not starvation free.
7) The process or thread that runs the 'cleanup' routine is assumed to never crash. Therefore it is ambiguous how the recovery mechanism deals with a case of system crash, i.e., all processes are dead simultaneously, or a case of power loss.

8) The ‘cleanup’ routine may choose a new owner for the lock in case it recognizes that the previous owner has crashed and therefore, the recovery mechanism does not guarantee an FCFS order in the lock acquisition and release even if the previous lock-holder resumes execution while the ‘cleanup’ is in progress itself.
9) In recovery activities, the ‘cleanup’ routine requires special mechanisms [71] in the operating system to reinitialize the lock and unlock it for other active processes, particularly when a process crashes while executing the CS. Therefore, this often necessitates maintaining a mapping of addresses pointing to the shared data objects and their values through an additional logging mechanism such as [72], which is also expensive in terms of system resources.

In our work, we wish to address the above mentioned issues by taking advantage of the persistent data storage the byte-addressable, non-volatile memories would provide. We make two assumptions in this regard: 1. that the future systems support these technologies; and 2. that access to non-volatile memories is provided by means of word-sized reads and writes as is the case with conventional memory. Researchers believe that non-volatile random access memory (NVRAM), when used as primary storage, i.e., when placed directly on the main memory bus, will enable performance-critical applications to harness the fast access and strong durability guarantees it provides. Michael Wu and Willy Zwaenpoel [73] proposed eNVy in 1993, one of the earliest alternatives to the conventional CPU-DRAM-magnetic disk approach, in which the architecture of a large non-volatile memory main memory storage system was built primarily using solid-state memory (Flash). A survey of emerging concepts and the main characteristics of new materials viz. PCM, FeRAM, MRAM, etc., being developed in the field of non-volatile memory technologies has been presented in [74]. Nevertheless, these technologies are not inert to some of the challenges experienced in the development of persistent data stores for filesystems and databases, especially in terms of facilitating a wear-aware memory allocation that is robust to data corruption, and techniques to provide cache-efficient and consistency-preserving updates. Katz et al. [75] proposed a method to detect and correct various data errors that may arise due to power loss in non-volatile memory, and particularly to identify if a write operation was interrupted due to a power loss and to reconstruct any data that may have become inconsistent in the interim. Moraru et al. in [76], proposed a general-purpose and flexible approach to address some of these challenges in preventing data loss and corruption, while imposing minimal performance overhead on the hardware, OS, and software layers as well.

In recent years, an abundance of research emphasized on creating consistent and durable non-volatile main memory (NVMM) technologies [77]–[83], on redesigning data structures that benefit from persistence

[84]–[89], as well as on how application-level interfaces are to be modified to exploit the benefits the persistent memories provide when placed on the main memory bus [90]–[93]. While transactional memories can be used to implement synchronization structures for high performance, strong consistency guarantees and resilience against failures, the fact that the vast majority of such memories are designed based on the volatile main memory renders them unusable for recovery from system-wide crashes. Recently, Coburn et al. proposed NV-Heaps [94], a system that provides transactional semantics to persistent in-memory data while preventing data corruption that may occur due to application and system failures. NV-Heaps avoid the operating system overheads by enabling applications to directly access non-volatile memory and thus exploit the high-speed and performance of NVRAM to build robust and persistent objects such as search trees, hash tables, sparse graphs and arrays. Allowing each process to own a set of 'operational descriptors', the fixed-size redo-logs similar to database recovery logs stored in the non-volatile memory assists the application in avoiding the overheads incurred by maintaining centralized recovery log. NV-Heap uses an 'epoch barrier' to ensure that a log entry in the operational descriptors is durably recorded in the non-volatile memory, and replays only valid log entries when recovering from failure, thus adding robustness even in the case of multiple failures. However, NV-Heaps have certain shortcomings that need to be addressed before being used for implementing a crash-recoverable concurrency control mechanism, such as: all locks being instantly released after a system failure; and stalling the transaction system until the recovery is done cleaning up the state of the system.

The idea is that main memory persistence will eventually be the norm in both enterprise and consumer-grade computing devices, as the recent advancements in NVRAM technologies position NVMM for a widespread deployment in the systems of the future. This advancement has served as a catalyst for our innovative algorithm that guarantees mutual exclusion and starvation freedom in a crash-recovery model, without the need to transfer the ownership of a lock in the event of a crash failure. We designed a new crash-recoverable mutex, RGLock, based on the premise that NVMM can be used to reconstruct and recover the in-memory state locally and near-instantaneously after crash failures, and any shared-state processes store is protected from data loss throughout recovery. The core of RGLock's merits is in the fact that there is no need to temporarily 'freeze' the shared data structure, nor is a 'cleanup' activity necessary. What particularly sets our lock apart is that there is no assumed requirement for crashes to be restricted to non-critical section only, and we consider this as a big leap forward in comparison to existing mutex lock algorithms.

Chapter 3

Model Specification

This chapter presents the execution model and terminology used in RGLock.

3.1 Hardware Considerations

The *system* is a finite number of processors in an asynchronous multi-processor architecture of Cache Coherent (CC) model that communicate with each other through a finite number of read-modify-write shared variables. For exposition, we abstract each processor in the system as an individual process. The processes communicate via shared variables whose values can be modified using atomic primitives such as read/write operations and special instructions like $swap\_and\_store$ (defined later in 5.2) and $compare\_and\_swap$. We assume that the main memory modules are based on the persistent and reliable Non-Volatile Random Access Memory (NVRAM) medium, i.e., the information stored on the medium is never lost or corrupted, and that the caching and memory ordering can be controlled to the point where the shared memory operations are atomic and durable [92], [95]–[98].

The memory locations in a CC model can be read from and written to by any process with the hardware maintaining consistency. Specifically, any memory location can be made locally accessible to a process at runtime by storing its contents in a local cache, which is maintained up-to-date by means of a cache coherence protocol [36] that ensures each modification of the data is atomically propagated across the shared memory either through updating or invalidating the copies held in other caches. When a process writes to memory, the cache follows one of the two policies: *write-through* or *write-back*. In the write-though policy, when a write occurs, the updates are written to the cache as well as to the lower layers in the memory hierarchy, i.e., either another cache or the physical memory itself. Whereas in the write-back policy, an update to a memory block is written to lower layers only when the contents of that cache line is modified. Although write-through policy incurs heavier traffic on the processor-to-memory interconnect, since the transactions with main memory are more often than in write-back policy, it propagates the updates to memory blocks to the main memory and the rest of the system more effectively than the latter.

Memory references that can be resolved entirely using a process's cache (e.g., in-cache reads) are called *local* and are much faster than *remote memory references* (RMRs), the ones that traverse the global processor-to-memory interconnect (e.g., cache misses). The time complexity of our algorithm is measured by counting the RMRs performed during a passage [99]. In a CC model, the RMR complexity depends on the state of each process's cache and the coherence protocol used for maintaining consistency. In our model, we count the first read of a shared variable on the main memory to make a local copy in a process's cache as one RMR, and all the subsequent reads on the cached copy are considered local until some other process overwrites it (possibly with the same value as before), which accounts for another RMR.

3.2 Formalism

A program is composed of *procedures*, which are fragments of code comprised of atomic statements in a deterministic algorithm. A *process* is a sequential program consisting of operations on variables. Each variable is either *private* or *shared*, depending on the scope of its definition; a *private* variable is defined within the scope of a procedure, whereas a *shared* variable is defined globally across all procedures. A variable stored on the global non-volatile shared memory can be made *locally accessible* by maintaining an up-to-date *cached copy* of its state, and can be modified by any process in the system for inter-process communication. Each process also holds in its volatile memory, a special private variable called the *program counter* that determines the next statement to be executed from the program's algorithm.

In describing our system, we use a less formal approach to the I/O automata model [100] by defining the behavior of processes using a pseudo-code representation. The interactions of processes with shared variables through operations applied is represented as a collection of *steps*. The values assigned to private and shared variables during each step are denoted by their *state*. The *statements* in the program's code comprise of per-process indivisible set of deterministic steps executed on a finite set of variables. We say a process is *enabled* to execute a given statement, when the process's program counter (PC) is pointing to that statement. Formally, the system is represented as a triple $\mathcal{S} = (\mathcal{P}, \mathcal{V}, \mathcal{H})$ where a finite set of processes with unique identifiers in $\mathcal{P} = \{p_1, p_2, \dots p_N\}$ interact with a finite set of variables $\mathcal{V}$ in corresponding sequence of steps recorded in an *execution history* $\mathcal{H}$.

Specifically, an *execution history* (or *history*, in short) is defined as a sequence of steps taken by the processes, each involving some shared memory operation and a finite amount of internal computation. A given statement can be executed multiple times in a history, and each such execution relates to a distinct step. A step in a history corresponds to a statement execution or a crash (formal definition in 3.3). The sequence of steps in a history corresponding to some procedural invocation is called an execution of a procedure. An execution of a procedure is *complete* if the last step in the execution is the last statement of the procedure.

For a given history H, set of process IDs $\mathcal{P}'$ (where $\mathcal{P}' \subseteq \mathcal{P}$) and a set of variables $\mathcal{V}$, the maximal subsequence of a history corresponding to the steps taken by only the processes in $\mathcal{P}'$ is called a *local history,* denoted by $H|\mathcal{P}'$. Henceforth, we use the notation p_i to refer to a single process in the system in general, where $p_i \in \mathcal{P}.$ Likewise, the maximal subsequence of H corresponding to only the steps taken by a single process p_i is denoted by $H|p_i$. For every process $p_i \in \mathcal{P}$ we define p_i to be *active* in a history H if the local history $H|p_i$ is non-empty. The notion of *fairness* in a system is that each individual process in the system is given an opportunity to perform its locally controlled steps infinitely often. A history may be either finite or infinite. Accordingly, a *fair history H* is either a finite, or an infinite history where every process that is active in H takes infinitely many steps.

The formal specification of a *protocol* consists of a collection of procedural statements for each process. Within a passage, a process *leaves* a protocol when it completes executing all the steps defined in its procedure and proceeds to the next protocol. In a fair history, once a process enters the entry protocol of its passage, we can depend on it to continue to interact with other processes until it has reached its critical section (if ever) and subsequently returned to its non-critical section. A process can remain in its non-critical section until it enters the entry protocol in a new passage. For simplicity, we assume that each process's CS and NCS executions are recorded as individual atomic steps in a history.

3.3 Crash-recoverable Execution

We consider a crash-recovery model for our system, where any process may crash and eventually recover, but the shared memory is reliable. A *crash* is a failure in an execution of one process where the private variables of the crashed process are reset to their initial values and the process simply stops executing any computation until it is active again. A *system crash* is a simultaneous failure in all processes in the system which resets all private variable to their initial state. A *crash-recovery procedure* is the sequence of steps taken by a crashed process to reconstruct its state and resume its active execution from the point of failure in the algorithm. A process is said to be *in recovery* until the execution of its crash-recovery procedure is complete. We classify the types of steps in our model into the following:

1) In a *normal step*, a process atomically accesses a set of variables in memory, executes some local computation depending on its state, and finally changes state as applicable.

2) In a *crash-recovery step* (*crash step*, for short), the program counter of the crashed process is set to a pre-defined crash-recovery procedure in the program code and any other private variable is reset to an initial state. However, any shared variable recorded by that process in non-volatile memory is unmodified.

3) In a *CS step*, a process executes the Critical Section of the passage it is in. Therefore, we say a process is *in the CS* if it is enabled to execute the *CS step*.

Extending the formalism introduced in 3.2, we define a *crash-recoverable execution history* H' as a fair history wherein every process either executes infinitely many passages or crashes a finite number of times. Note that permanent crash-stop failures are excluded from our definition of a crash-recoverable execution and indefinitely recurrent failures are allowed as long as they do not impede the overall progress in the system. In the absence of failures, H' is identical to the H defined in 3.2, in that it is just a sequence of normal steps. As illustrated in Figure 3.1, a *failure-free passage* is an iteration of a loop among the NCS, entry, CS and exit protocols without any crash-recovery step. A *crash-recoverable passage* is composed of NCS, entry, CS and exit protocols with an invocation of a crash-recovery procedure whenever a crash occurs. As indicated by the recovery transitions, a crash-recovery procedure resumes the process's execution based on its state reconstructed from NVMM. Intuitively, whenever a process crashes, it does not leave the protocol it crashed in, but only takes a crash-recovery step instead. Once the active execution is resumed, it is up to the process's program to complete each remaining protocol within that passage.

We assume that the code for critical section is idempotent, i.e., harmlessly repeatable by a process in recovery if it has the necessary exclusive access to do so, i.e., even if a process crashes during the execution of a CS step and/or if the process repeats the CS step within the same passage more than once when in recovery, the program works correctly. This assumption is reasonable since there are no race conditions on the shared state protected by the critical section as it cannot be concurrently modified by any other process when one process already owns an exclusive access to the critical section (guaranteed by the mutual exclusion property discussed later on). Moreover, since the same parameters are applied each time a process executes the idempotent operations, there is no inconsistency caused in the program state by some process repeating the CS execution within the same passage in recovery, provided it has the necessary exclusive access to the shared resource.

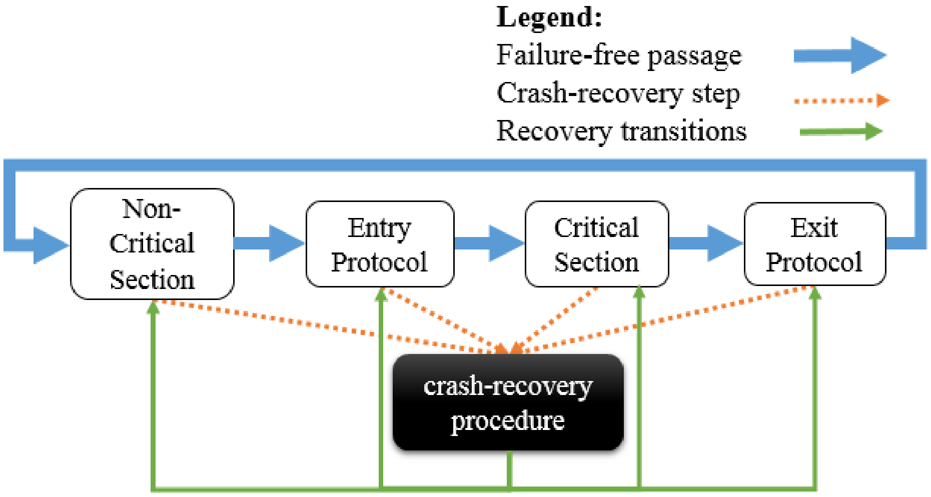

Figure 3.1 Failure-free and crash-recoverable passages.

Finally, the assumptions made in our crash-recovery model are as follows:

A1. A process in recovery reconstructs its state from the shared variables stored in non-volatile memory.

A2. Process crashes are independent, i.e., failure of one process does not crash other active processes in the system.

A3. Other active processes in the system may read, modify and write to the globally accessible shared variables of a process in recovery.

According to **A3**, a crashed process can reconstruct its state and resume computation based on the latest changes (if any) made to its shared variables by other processes. For example, a busy-waiting process p_i could crash in a step immediately before the step in which the existing lock-holder changes the bit on p_i's shared variable used for spinning, indicating that p_i is now the new lock-holder. In recovery, p_i can read this change and then proceed to its critical section. The assumption **A1** is crucial to our model in that a process in recovery does not restart with its shared state reset to initial values. Resetting shared variables in the system each time a process restarts from a crash would render the concept of using persistent memory trivial.

Chapter 4

Recoverable Mutual Exclusion

The challenge in designing a recoverable spin-lock algorithm lies in specifying the correctness properties of mutual exclusion for a crash-recovery model where processes can crash at any point within a passage, i.e., the algorithm should be tolerant to crashes even within the critical section. Mutual Exclusion is a *safety* property (informally construed as 'some particular bad things never happen'), which assists in reasoning with the correctness of computation in a multi-processor system. Livelock-freedom and starvation-freedom are *liveness* properties, informally stated as 'something good eventually happens'. While livelock-freedom guarantees the overall progress of the system as each process eventually releases the lock held by it, starvation-freedom guarantees a per-process progress in the system. Formally, the correctness properties of a recoverable mutual exclusion algorithm are stated as the following:

Mutual Exclusion (ME): No two processes are in the critical section simultaneously.

First-come-first-served (FCFS): If a process p_i completes its doorway before another process p_j enters its doorway, then p_j cannot enter the critical section before p_i does in their corresponding passages.

Livelock-freedom (LF): In a *crash-recoverable history*, if some process is in its entry section, then some process eventually enters its critical section.

Starvation-freedom (SF): In a *crash-recoverable history*, if a process is in its entry section, then that process eventually enters its critical section, i.e., no process in its entry protocol is perpetually denied access to the critical section.

Terminating Exit (TE): In a *crash-recoverable history,* if a process is in its exit protocol, then it completes that protocol within a finite number of steps.

Finite Recovery (FR): Every crash-recovery procedure invoked in a *crash-recoverable history* completes within a finite number of steps.

Importantly, although ME is the guarantee that solves the problem of allowing only one process to execute the CS at any time, we do not solve the problem of restricting a lock-holder from executing the CS more than once per passage in the event of crash failures, based on the assumption that the CS is idempotent. FCFS guarantees that when a slow-running process is in recovery there is no accidental ‘usurping’ of its lock by other contenders, contrary to the recovery mechanism defined in [37]. While ME and FCFS are the same as the conventional properties guaranteed by most existing spin lock algorithms in related literature, the properties of LF, SF, TE and FR are refined in our model by introducing the context of crash-recovery. The finite recovery (FR) property follows immediately from the definition of crash-recoverable history, and is crucial to the overall progress in the system as the absence of which renders little sense to the very concept of crash-recovery. Notice that FR does not exempt a process from crashing again while it is already in recovery. Instead, it guarantees that the process eventually completes some crash-recovery procedure. In our knowledge, RGLock is the first mutual exclusion algorithm for a crash-recovery model that conforms to *all* the properties stated above.

Chapter 5

Algorithm

In this chapter, our algorithm is presented along with a high-level description of the included procedures. RGLock is a new locking mechanism that

- exploits the benefits of memory persistency in NVMM systems;
- guarantees FCFS ordering of lock acquisitions;
- spins on locally accessible memory locations only; and
- guarantees both safeness and liveness properties of recoverable mutual exclusion even in the presence of crash failures.

5.1 RGLock

In this thesis, we present a scheme for effectively dealing with the problem faced by conventional MCS spin-lock in a crash-recovery model. In particular, the MCS lock suffers from the limitation of the lock becoming unavailable whenever one or more of the processes holding or waiting to acquire the lock crash. Our solution to the problem is based on maintaining and manipulating information in the persistent shared memory. The fundamental idea behind our algorithm is to reconstruct the shared state of the process from the persistent memory to perform effective recovery activities in the event of a crash failure. The key novel feature of our algorithm is that the integrity of the lock structure i.e., the sequence of processes trying to acquire the lock, is preserved even in the presence of failures. In addition to the most commonly available read-modify-write atomic primitives, the design of RGLock proposes an atomic $swap\_and\_store$ (SAS) instruction (described formally in 5.2) that is not supported by current generation of multiprocessors for fair lock acquisitions, and benefits from the availability of atomic $compare\_and\_swap$ (CAS) instruction in providing strict FIFO ordering. Without the CAS instruction, inspecting if there are any other processes waiting in line to acquire the lock and setting the lock free cannot happen atomically. Without the SAS instruction in the entry protocol, the FIFO ordering in lock acquisitions cannot be guaranteed and some of the contending processes are prone to starvation when crashes occur.

Each process using the lock allocates a per-process lock access structure called *qnode* and appends it to a linked-list of qnodes, wherein processes holding or waiting for the lock are chained together by the backward pointers ($ahead$ pointers). The lock is represented by a pointer L, set either to the qnode of that process at the tail of the linked-list, or to a predefined $null$ when the lock is free. For any process p_i, the process that appended its qnode (if exists) to the linked-list immediately before p_i's completion of the doorway instruction is its *predecessor* and the process that swaps the lock pointer from p_i's qnode is its *successor*. Each qnode contains the following fields:

- a checkpoint number chk that signifies the protocol of a passage the process is in,
- an $ahead$ pointer to hold the address of the predecessor and act as the variable for busy-waiting,
- and a $next$ pointer to hold the address of the successor.

The pseudo-code for a failure-free passage and crash-recoverable procedures are shown in Algorithm 5.a and 5.b respectively. Our programming notation is largely self-explanatory. At system startup the processes running the program initialize their shared variables before their first execution of the default procedure, $\mathrm{main}(\,)$. Once initialized, processes execute the $\mathrm{main}(\,)$ procedure, which invokes the spin lock acquisition and release protocols as shown in the pseudo-code. Therewith, whenever a process takes a *crash-recovery step*, its program counter is reset to the beginning of the $\mathrm{main}(\,)$ procedure. Throughout the pseudo-code, indentation is used to indicate nesting and each statement is labeled for reference. Angle brackets ($\langle \ldots \rangle$) are used to enclose the operations to be performed atomically. Shared variables stored in NVMM are termed 'non-volatile' and any additional temporary private variables are declared within the procedures they are used in as required.

At system startup, a process p_i initializes its qnode accessible by pointer q_i with the above described fields as shared variables in its non-volatile memory. Although the $q_i.chk$ field is treated as a shared variable, we assume that only p_i writes or updates this field in lines of code (e.g., $E2$, $E7$, $D1$, $D4$, $D8$, etc.) that are recorded as steps in the execution history. During execution of a passage, the checkpoint chk is updated to an appropriate value from a predefined set consisting of $\{0, 1, 2, 3\}$ whenever the process is transitioning to a new protocol in its passage. This checkpointing variable aids a process taking a *crash-recovery* step in invoking the appropriate crash-recovery procedure, depending on the value read. In absence of failures, a process only executes the $failureFree$ procedure which contains the entry and exit protocols for acquiring the lock, executing the CS, and releasing the lock. The execution of a passage begins at line $CR1$ and ends immediately after the first execution of line $CR8$.

The *head* qnode in the linked-list has a $null$ predecessor and the *tail* qnode has a $null$ successor. Each busy-waiting process spins on its own local variable, i.e., the $ahead$ pointer on its own qnode, until it reaches the head of the list. $Swap\_and\_store$ instruction enables a process to determine the link to its predecessor in the event of a crash. $Compare\_and\_swap$ enables a process to determine whether the linked-list contains only its qnode while releasing the lock, and if so remove itself correctly as a single atomic action. If a lock-holder identifies a non-null successor in the linked-list, then the lock is relinquished by *promoting* the successor, i.e., by resetting the $ahead$ pointer on the successor's qnode to $null$. A *timing window* exists in the $acquire\_lock$ procedure between the step in which a process completes the SAS instruction i.e., in which it appends its qnode to the linked-list, and the step in which it completes the instruction in which it provides information to its predecessor (if applicable) about how to be notified when the predecessor is granting it the lock, i.e., by setting the $next$ pointer from the predecessor to its own qnode. The spin in the $acquire\_lock$ waits for the lock to become free, and the spin in the $release\_lock$ compensates for the timing window. Both the spins in the entry and exit protocols are local.

```
type qnode = record
    non-volatile next : *qnode := $q_i$
    non-volatile ahead : *qnode := $q_i$
    non-volatile chk : int := 0

type lock = *qnode

/* $q_i$ is a qnode record of the invoking process $p_i$.
$Q$ is a history variable. $RMEQ$ is a linked-list of
qnodes.*/

main (L: *lock, $q_i$: *qnode)
/* Default procedure that guarantees
recoverable mutual exclusion for every $p_i \in \mathcal{P}$ */

  CR1.  cp := $q_i$.chk
  CR2.  if cp = 1
  CR3.    if recoverBlocked(L,$q_i$)=false
  CR4.        failureFree(L,$q_i$)
  CR5.  if cp = 2   recoverHead(L,$q_i$)
  CR6.  if cp = 3   recoverRelease(L,$q_i$)
  CR7.  else failureFree(L,$q_i$)
  CR8.  Non-Critical Section

failureFree (L: *lock, $q_i$: *qnode)

  FF1.  acquire_lock(L,$q_i$)
  FF2.  Critical Section
  FF3.  release_lock(L,$q_i$)
```

```
acquire_lock (L: *lock, $q_i$: *qnode)

  E1.  $q_i$.next := null
  E2.  $q_i$.chk := 1
  E3.  $\left\langle \begin{matrix} E3a.\ SAS(L, q_i, q_i.ahead), \\ E3b.\qquad Q := Q \circ \langle p_i \rangle \end{matrix} \right\rangle$
  E4.  if $q_i$.ahead ≠ null
  E5.      $q_i$.ahead.next:= $q_i$
  E6.      repeat while $q_i$.ahead ≠ null
  E7.  $q_i$.chk := 2
  E8.  return

release_lock (L: *lock, $q_i$: *qnode)

  D1.  $q_i$.chk := 3
  D2.  if $q_i$.next = null
  D3.      $\left\langle \begin{matrix} D3a.\ \textbf{if}\ CAS(L, q_i, null), \\ D3b.\qquad Q := Q \backslash \langle p_i \rangle \end{matrix} \right\rangle$
  D4.              $q_i$.chk := 0
  D5.              return
  D6.      repeat while $q_i$.next = null
  D7.  $\left\langle \begin{matrix} D7a.\ q_i.next.ahead = null, \\ D7b.\quad Q := Q \backslash \langle p_i \rangle \end{matrix} \right\rangle$
  D8.  $q_i$.chk := 0
  D9.  return
```

Algorithm 5.a Failure-free and main procedures of RGLock

Boolean recoverBlocked (L: *lock, q_i:*qnode)

/ If $q_i \in RMEQ$ holds, complete the passage and return true. Else return false. */*

```
RB1.  tailNow: *qnode := L
RB2.  if tailNow = null
RB3.      return false
RB4.  else if tailNow = $q_i$
RB5.      waitForCS($q_i$)
RB6.  else
RB7.      if $q_i$.next = null
RB8.          if findMe(L,$q_i$)
RB9.              waitForCS($q_i$)
RB10.         else return false
RB11.     else waitForCS($q_i$)
RB12. recoverHead(L,$q_i$)
RB13. return true
```

recoverRelease (L: *lock, q_i: *qnode)

/ If $q_i \in RMEQ$ holds, then release the lock. Else complete the passage */*

```
RR1.  tailNow: *qnode := L
RR2.  if tailNow = null
RR3.      $q_i$.chk := 0
RR4.      return
RR5.  else if tailNow = $q_i$
RR6.      release_lock(L,$q_i$)
RR7.  else if findMe(L,$q_i$)
RR8.      release_lock(L,$q_i$)
RR9.  $q_i$.chk := 0
RR10. return
```

recoverHead (L: *lock, q_i: *qnode)

// execute CS and release the lock.

```
RH1. Critical Section
RH2. release_lock(L,$q_i$)
RH3. return
```

waitForCS (q_i: *qnode)

//link q_i behind its predecessor in $RMEQ$ (if applicable) and busy-wait to enter CS

```
W1. if $q_i$.ahead ≠ null
W2.     if ($q_i$.ahead≠null ∧ $q_i$.ahead.next=null)
W3.         $q_i$.ahead.next := $q_i$
W4.     repeat while $q_i$.ahead≠null
W5. $q_i$.chk := 2
W6. return
```

Boolean findMe (L: *lock, q_i: *qnode)

*/*scans $RMEQ$ to locate q_i and returns true if q_i is found. N is total no. of processes in $\mathcal{P}$ and run is loop iterator*/*

```
F1.  temp: *qnode := L
F2.  run: int := 1
F3.  while(run<N)
F4.      temp := temp.ahead
F5.      if temp = null
F6.          return false
F7.      if temp = $q_i$
F8.          return true
F9.      run := run +1
F10. return false
```

Algorithm 5.b Crash-recovery procedures of RGLock

For exposition, we distinguish a number of phases in which a process may be at the end of an execution history. The transitions among these phases are governed by atomic shared memory operations as illustrated in Figure 5.1. At system startup, the lock is free, i.e., L points to $null$, and all processes are in $DELETED$ phase. In the *doorway* instruction at line $E3$, every process p_i that is trying to acquire the lock appends its qnode at the trailing end of the linked-list atomically, and transitions to the $APPENDED$ phase. The atomic block of statements at line $E3$ contains an operation on the history variable Q at line $E3b$. Note that the history variable is used only for reasoning about the correctness of the algorithm and does not indicate any shared memory operation. The notation $Q \circ \langle p_i \rangle$ denotes appending the ID of a process p_i to a sequence Q, such that $Q[|Q|] = p_i$, where $|Q|$ denotes the length of Q.

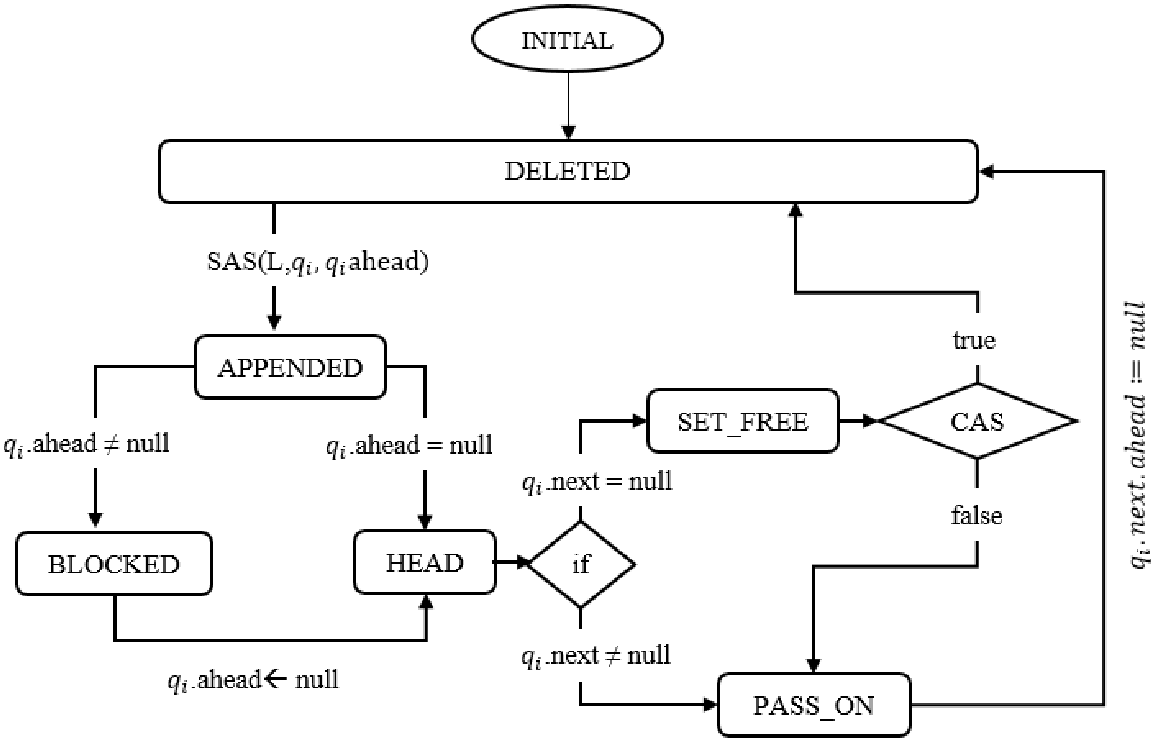

Figure 5.1 Phase transitions of a process p_i executing RGLock algorithm.

If p_i has a non-null predecessor, then p_i *links behind* it by setting the $next$ pointer of its predecessor to q_i at line $E5$, and spins on its own $q_i.ahead$ field, waiting for it to become $null$ in the $BLOCKED$ phase. Otherwise, p_i is the *head* process and becomes the lock holder ($HEAD$ phase). A process p_i transitions from $BLOCKED$ phase to $HEAD$ phase when its predecessor writes $null$ to the *ahead* field on its qnode, denoted by $q_i.ahead \leftarrow null$. After executing its critical section, the head process releases the lock in the exit protocol defined by the statements $D1 - D9$. There are two cases when the lock-holder releases the lock. If the tail pointer L still points to the lock-holder's qnode, then there are no processes waiting to acquire the lock and the head process releases the lock by setting L to $null$ (SET_FREE phase). Otherwise, setting the lock free by pointing L to $null$ in such case violates the FCFS property and could potentially introduce starvation. As a counter-measure, the program takes advantage of the $compare\_and\_swap$ (CAS) instruction for releasing the lock. If the CAS operation at line $D3a$ fails, then the lock-holder waits in a spin loop (at line $D6$) until the successor updates its $next$ pointer, in order to avoid the *timing window* where a process has appended its qnode to the linked-list but has not yet linked behind its predecessor.

When the lock-holder finds a non-null successor, i.e., when $q_i.next \neq null$, the lock is relinquished at line $D7$ by *promoting* the successor as the new head ($PASS\_ON$ phase). The notation $Q \backslash \langle p_i \rangle$ at lines $D3b$ and $D7b$ denotes the deletion of element p_i from the sequence Q, where p_i is the head process at $Q[1]$. The operations on Q are treated to be executed atomically along with the write, CAS, or SAS operations they are enclosed with. The operation at lines $D8$ and $E1$ are only house-keeping actions required for the reuse of the process's qnode in subsequent passages. The process is in the $DELETED$ phase when executing these statements. After releasing the lock a process enters NCS in the $DELETED$ phase. A process may remain indefinitely in its non-critical section until its subsequent completion of the doorway instruction.

5.2 Swap and Store

Consider the atomic $fetch\_and\_store$ (swap) based implementation of the entry protocol in the original MCS spin lock. Each process appends its qnode to a linked-list via the swap operation and busy-waits until it is notified of being the lock owner. The first and most obvious challenge in the event of a crash is tracing the ownership of such a spin lock. In particular, when a process attempts to resume execution following a failure, the lock acquisition is in an ambiguous state, in that any evidence of it ever swapping L is lost in the crash. From the 'enqueue' operation defined in the MCS lock (see pred : *qnode := swap (L,I) instruction in Algorithm 1), if the atomic swap is immediately followed by a write which stores the address of its immediate predecessor in its persistent memory, it greatly simplifies the membership (or position) tracking of a crashed process in the linked-list when it is *in recovery*. Unfortunately, the swap operation cannot be used to also register the lock ownership atomically using either the basic hardware instructions or the non-blocking techniques developed using *compare_and_swap* [101]. Contemporary architectures perform these operations sequentially in two independent steps as a swap followed by a write. However, in case a process crashes immediately after the swap and before the succeeding write, the ambiguity remains unresolved in its crash-recovery procedure. In such scenario, the ability of other active processes to acquire the lock is restricted, and thus, the overall liveness of the system is also affected. For instance, a process cannot release the lock to its successor if the successor crashed while the lock-holder is waiting in its exit protocol for the successor to link behind and consequently the successor has no knowledge of its membership in the linked-list in recovery, thus causing a permanent delay in lock release.

Clearly, if the two operations viz. appending to the linked-list and registering the address of the immediate predecessor were performed atomically, we could always trace out which process currently holds the lock and which processes are busy-waiting in line for the lock. Similarly, if a lock-holder crashes immediately after promoting its last known successor, then the recovery mechanism becomes complicated since the process in recovery may execute the exit protocol once again causing a safety violation by promoting an already promoted process twice. As a first step towards resolving this problem, we propose a special instruction as defined below. We assume that the address of a process's qnode is a pointer to a single word that can be written atomically.

atomic ***swap_and_store*** **(SAS):**

In one indivisible atomic step, a swap is immediately followed by a store that writes the result of the swap operation to a location in the non-volatile memory. Also, no other write is allowed to that memory location between the load and store parts of a swap. Consequently, an FCFS order is ensured in the lock acquisitions in RGLock due to the SAS operation in the doorway instruction. Given two elements to swap (for e.g., reference to the tail of the linked-list and the qnode to be added to the linked-list, in the doorway instruction) and a memory location to store the result of the swap operation atomically, a pseudo-code representation of the SAS instruction is shown as:

```
function SAS (old_element: address, new_element: value, location: address)
            atomic {
                    temp: val_type  := *old_element
                    *old_element    := new_element
                    *location        := temp
                    }
```

5.3 Crash-recovery Procedures

We will now discuss RGLock implementation in a crash-recoverable execution. The primary objective of a crash-recovery procedure is to identify the protocol the process executed before the crash. Recall that the checkpointing number chk is used for this purpose. As the shared variables stored on the non-volatile medium remain persistent during a crash, the process p_i in recovery can easily identify its qnode through its q_i pointer. Then, the recovery function to be invoked is determined by the checkpoint value read at line $CR1$. If $q_i.chk$ is 1, the process had already completed line $E2$ in its entry protocol before crashing. Then the *recoverBlocked* method is invoked at line $CR3$, which identifies the position of the process in the linked-list, on the following basis:

a) If the lock pointer L is currently $null$, it implies that the linked-list is empty and q_i was not appended to it before the crash.

b) If L points to q_i itself, clearly p_i is in the $APPENDED$ phase, and the next steps in recovery are about completing the remainder of the entry and exit protocols in that passage.

c) If L points to some other process's qnode, further investigation is necessary to verify if q_i was appended to the linked-list before the crash or not.

Statements *RB7-RB11* in $recoverBlocked$ method construe case (c). We use the $findMe$ function to determine the membership of q_i in the list of qnodes. If $findMe$ determines that q_i is reachable by tracing the *ahead* pointers starting from the qnode of the tail process in the current linked-list, then the $waitForCS$ method defined in lines $W1 - W6$ identifies q_i's predecessor (if exists) and emulates the busy-waiting steps of the $acquire\_lock$ procedure so that the process in recovery maintains the liveness of the system. We assume that the qnode of a process fits into a single word on the memory and that the read operations in W2 incur only one remote memory reference. Particularly, the assumption is that a process p_i makes a local copy of the qnode pointed by $q_i.ahead$ in a temporary variable during the first read of the $q_i.ahead$ value and the subsequent read of the value of $q_i.ahead.next$ is read from the temporary variable itself instead of referencing the actual qnode structure on the non-volatile main memory.

The execution returns from $waitForCS$ when q_i is promoted (if ever) as the head process, and then p_i completes the $recoverHead$ procedure before returning to NCS at $CR8$. As p_i is already in $APPENDED$ phase in case (b), it is hence a straightforward execution of $waitForCS$ invoked at line $RB5$, followed by

$recoverHead$ and $CR8$. For case (a), since the lock is free and p_i still remains in $DELETED$ phase as it has not completed the doorway instruction before the crash, the crash-recovery procedure invokes the $failureFree$ procedure at line $CR4$ when the process returns from $recoverBlocked$ with a $false$ response, which is essentially an execution of a new failure-free passage.

If the checkpoint read at $CR1$ is 2, it implies that p_i was the head when it crashed. Then the $recoverHead$ procedure is invoked at $CR5$. As shown in the code, p_i can then execute the critical section at $RH1$ and eventually release the lock as it would in a failure-free passage. On the other hand, if the checkpoint at $CR1$ is 3, it indicates that the process had begun executing the exit protocol before crashing and hence, the $recoverRelease$ method is invoked. Again, there are three possible cases to be considered at this point:

A) L points to $null$, which implies that p_i has already set the lock free, but crashed immediately after completing line $D3$. Hence, p_i returns to NCS from $RR4$.

B) L points to q_i, which implies that p_i has not set the lock free before crashing and is still the tail. Hence, the $release\_lock$ procedure is invoked at $RR6$, following which p_i returns to NCS at $CR8$.

C) L points to some other qnode. In such case it is ambiguous whether p_i relinquished the lock before the crash or not. Hence, the $findMe$ method invoked at $RR7$ determines q_i's membership in the linked-list. Then p_i takes the next steps according to the response returned by the $findMe$ method and finally returns to NCS at $CR8$.

Evidently, $findMe$ plays a crucial role in both the crash-recovery procedures described above. As seen at line $F1$, the current tail is identified by reading the lock pointer L and a temporary variable is assigned to it. The idea is to scan through the list of qnodes from the read tail, all the way to the head. Should the scan find a match in the queue with the invoking process's qnode, it returns $true$, otherwise it returns $false$. The scan loop at $F3$ terminates when either the head qnode is reached, or when the number of iterations reaches a limit of $N - 1$, where N is the number of processes in the system. The reasoning behind limiting the loop to $N - 1$ iterations is that in case a process's qnode is still referenced by some process in the linked-list, then it takes a maximum of $N - 1$ hops to reach its qnode starting from the tail assuming the worst case of all N processes having their respective qnodes appended to the linked-list. On the other hand, if the qnode of the invoking process does not exist in the linked-list, then allowing the $findMe$ scan to run for an unbounded number of iterations would result in a non-terminating loop in case other processes in the system pass through CS infinitely often. Specifically, in an asynchronous system, there exists a possibility of the $findMe$ scan never reaching a qnode whose $ahead$ pointer is $null$.

Chapter 6

Correctness

In this chapter, the RGLock algorithm is proved correct with respect to the correctness properties of Recoverable Mutual Exclusion described in Chapter 4. The correctness of our algorithm is derived by an induction on the length of the execution history or by contradiction where applicable, following the style of the proof for Generic Queue-based Mutual Exclusion algorithm in [102].

6.1 Notational Conventions

The system in consideration is represented as a triple $\mathcal{S} = (\mathcal{P}, \mathcal{V}, \mathcal{H})$ where a finite set of processes with unique identifiers in $\mathcal{P} = \{p_1, p_2 \dots p_N\}$ interact through operations on a finite set of shared variables in $\mathcal{V} = \{L, q_i.ahead, q_i.chk, q_i.next\}$ in a sequence of steps recorded in an execution history $H \in \mathcal{H}$, where $\mathcal{H}$ is a set of all histories starting from the initial state. Unless stated otherwise, i and j are integers that range over $1 \dots N$. A step that occurs in H is denoted by $s \in H$. For any execution histories $G, H \in \mathcal{H}$, $G \preccurlyeq H$ denotes that G is a prefix of H, and $G \prec H$ denotes that G is a proper prefix of H. $G \circ H$ denotes the concatenation of G and H (i.e., elements of H appended to G). If G is finite, $|G|$ denotes the length of G, and $G[s'..s'']$ denotes the subsequence of G consisting of every step s such that $s' \leq s \leq s''$. The state of an object $v \in \mathcal{V}$ at the end of $H \in \mathcal{H}$ is denoted by v^H. For instance, $\mathrm{q_i}.ahead^H$ refers to the state of the $ahead$ field on p_i's qnode at the end of history H. Finally, $p_i@label$ is used to denote the line of code from the algorithm with the given label a process $p_i \in \mathcal{P}$ is enabled to execute.

6.2 Preliminaries

Consider a concurrent system $\mathcal{S} = (\mathcal{P}, \mathcal{V}, \mathcal{H})$ executing the RGLock algorithm. Based on the informal description in Chapter 5, processes in $\mathcal{P}$ start from the initial state where each process has a distinct qnode record in the shared memory. An execution of each *passage* by a process p_i begins when p_i is enabled to execute line $CR1$ and ends immediately after the first subsequent execution of line $CR8$. Each concurrent execution of the RGLock algorithm is expressed as a history $H \in \mathcal{H}$ in which the shared memory operations are atomic. H may contain *normal steps*, *CS steps* and *crash-recovery steps* as defined in 3.3. For exposition, a history variable Q is used, whose state at the end of H is a sequence of process IDs corresponding to those processes contending for the resource in critical section in H. Q supports append and delete operations. In particular, p_i is appended to Q when the corresponding process executes line $E3b$ in the entry protocol, and is deleted from Q when the process executes either line $D3b$ or $D7b$ in the exit protocol. A delete operation removes all instances of p_i in Q and has no effect if p_i is not in Q. As shown later on in Lemma 6.2.4-(ii) and in Corollary 6.3.3, Q contains at most one instance of p_i at all times and processes delete themselves from Q in a FIFO order.

Definition 6.2.1. *For any finite history $H \in \mathcal{H}$, and any process $p_i \in \mathcal{P}$, $p_i \in Q^H$ denotes that p_i is in the sequence of process IDs corresponding to the state of Q at the end of H.*

Definition 6.2.2. *Let $RMEQ$ denote a linked-list of per-process qnode structures (each denoted by q_i) whose links are determined by their '$ahead$' pointers. L is a pointer to the tail of $RMEQ$ and L is $null$ when the lock is free. If there is a qnode q_i such that $q_i.ahead = null$ and $q_i \in RMEQ$, i.e., q_i can be reached by following the $ahead$ pointers from the tail L, then q_i is the head of $RMEQ$. A non-empty $RMEQ$ is acyclic (from the result of Lemma 6.2.4-(ii) and Invariant-6.3(H, p_i)-(e)) either since it has only a single qnode, or since the head qnode is duly dereferenced by the process itself when it 'promotes' its successor.*

Informally, $RMEQ$ is the shared data structure that facilitates processes in appending their qnodes at the tail end for acquiring the lock, and in scanning the list of qnodes from the tail to the head (if required) in a crash-recovery procedure. The correctness of the RGLock algorithm is proved based on the abstract properties of $RMEQ$.

Lemma 6.2.3. *For any $H \in \mathcal{H}$, and any process $p_i \in \mathcal{P}$, if p_i completes line $E3b$ in step s in H, then p_i does not complete $E3b$ again in H after s, unless it first completes line $D3b$ or $D7b$.*

Proof. The lemma is proved by contradiction. Let step s be the first execution of line $E3b$ by process p_i in H. Suppose for contradiction that the hypothesis stated in the lemma is false, i.e., p_i completes $E3b$ again in H after s, say in step s', without executing $D3b$ or $D7b$ between s and s'. This supposition is denoted by $\star$ for convenience.

Case 1: p_i does not crash between s and s'. According to the algorithm, a process p_i appends itself to Q only through an execution of line $E3b$, and deletes itself from Q only through an execution of either line $D3b$ or $D7b$. As per the lines of code corresponding to a failure-free execution ($FF1 - FF3$), once p_i completes line $E3b$ in a passage, it does not execute $E3b$ again until and unless it has completed lines $D3b$ or $D7b$, and $CR8$, i.e., the process p_i has to complete the exit protocol associated with that passage before the subsequent execution of line $E3b$, which contradicts $\star$.

Case 2: p_i crashes between s and s', say in step s''. Without loss of generality, suppose that s'' is the last crash-recovery step taken by p_i after s and before s'. Then p_i's steps from s'' to s' are failure-free. Since p_i takes a crash-recovery step s'' after completing line $E3b$ in step s, the $q_i.chk$ value immediately after s'' depends on the procedures completed by p_i between s and s''. Note that $q_i.chk$ is set to 1 only at line $E2$, to 2 only at line $E7$ or $W5$, to 3 only at line $D1$ and to 0 at line $D4$, $D8$, $RR3$, or $RR9$ in the code. The proof proceeds by a case analysis on the value of $q_i.chk$ immediately after s''.

> ***Subcase 2.1*:** $q_i.chk$ is 1. Then the next steps by p_i in recovery in $H[s''..s']$ are an execution of the *recoverBlocked* method (lines $RB1 - RB13$) invoked at $CR3$. Since p_i has already completed line $E3b$ in s, $p_i \in Q$ holds in $H[s..s'']$ unless it deletes itself from Q by executing line $D3b$ or $D7b$ in some step in $H[s..s'']$, which contradicts $\star$. Since $L = q_i$ by the action of s, the condition at $RB2$ holds only if p_i sets $L = null$ by completing line $D3$ in $H[s..s'']$, which contradicts $\star$. Moreover, the *findMe* method invoked at $RB8$ returns $false$ only if p_i's qnode is not found in the linked-list $RMEQ$. Since q_i is appended to $RMEQ$ in step s, the only possibility for *findMe* to return false is in case if p_i completed $D7$, or if p_i set $L = null$ by completing line $D3$ before some other processes appended their qnodes to $RMEQ$, both contradicting $\star$. Consequently, $recoverBlocked$ cannot return false at either $RB3$ or $RB10$, without contradicting $\star$. Accordingly, p_i completes the $waitForCS$ procedure invoked either at $RB5$ or $RB9$, followed by the $recoverHead$ procedure

invoked at $RB12$, which restricts p_i to complete the passage by executing $D3b$ or $D7b$ and return to line $CR8$ before s', thus contradicting $\star$.

***Subcase 2.2*:** $q_i.chk$ is 2. Then the *recoverHead* procedure at line $CR5$ is invoked after s'', which has code only for executing CS and exit protocol (lines $D1 - D9$), and the process then returns to NCS at line $CR8$ before reaching $E3b$ again in step s'. Thus, $H[s''..s']$ contradicts $\star$.

***Subcase 2.3*:** $q_i.chk$ is 3. In recovery, the *recoverRelease* procedure invoked at line $CR6$ determines whether the process has already completed the passage through a series of checks at lines $RR2, RR5$, and $RR7$. Since $L = q_i$ by the action of s, the condition at $RR2$ holds only if p_i sets $L = null$ by completing line $D3$, which contradicts $\star$. And if p_i reaches $RR5$ and if the condition at $RR5$ holds, then the *release_lock* procedure invoked at $RR6$ will execute $D3a$ or $D7b$ between s'' and s', which also contradicts $\star$. Finally, as argued in Subcase 2.1 above, the *findMe* method invoked at $RR7$ after s'' cannot return $false$ without contradicting $\star$. Then if *findMe* at $RR7$ returns $true$, then the process deletes itself from Q by executing the *release_lock* procedure at $RR8$ and then returns to NCS at $CR8$, and thus $H[s''..s']$ contradicts $\star$.

***Subcase 2.4*:** $q_i.chk$ is 0. Then p_i has already completed line $D4$, or $D10$, or $RR3$, or $RR9$ in $H[s..s'']$, and this contradicts $\star$ because by the structure of the algorithm, line $D3b$ or $D7b$ is always executed before p_i executes line $D4$, $D10$, $RR3$, or $RR9$.

□

Lemma 6.2.4. *For any finite history $H \in \mathcal{H}$, and any process $p_i \in \mathcal{P}$, the following hold:*

i. $p_i \in Q^H \iff$ *there is a step in H, in which p_i has completed line $E3b$ and has not completed $D3b$ or $D7b$ subsequently since its last execution of $E3b$ in H; and*

ii. Q^H *contains at most one instance of p_i.*

Proof.

Part (i). The proof proceeds by an induction on $|H|$.

Base Case: $|H|= 0$. In such case, every process is in its initial state, has never executed line $E3b$, $D3b$ or $D7b$; and the queue is an empty sequence, i.e., $Q^H = \langle\,\rangle$. Therefore Lemma 6.2.4-(i) holds trivially.

Induction Hypothesis (IH): For any $k > 0$, assume that Lemma 6.2.4-(i) holds for all histories in $\mathcal{H}$, such that $|H| < k$.

Induction Step: Now it is required to prove that Lemma 6.2.4-(i) holds for all H such that $|H| = k$. Let σ be the last step in H. Perform a case analysis on σ.

Case 1: σ is an execution of some line of code other than $E3b, D3b,$ or $D7b$. Then Lemma 6.2.4-(i) follows directly from the IH because the state of Q does not change by the action of σ, even if σ is a crash-recovery step by p_i.

Case 2: σ is an execution of $E3b$. Then $p_i \in Q^H$, because p_i is appended to Q at line $E3b$, and this implies Lemma 6.2.4-(i) since p_i has not executed either line $D3b$ or $D7b$ in H since its last execution of $E3b$, which occurred in step σ itself. And for every $p_j \in \mathcal{P}$ where $j \neq i$, Lemma 6.2.4-(i) follows directly from the IH, since p_j is neither appended nor deleted from Q by the action of σ.

Case 3: σ is an execution of $D3b$ or $D7b$. This implies Lemma 6.2.4-(i) because if p_i had previously completed line $E3b$ in H, then it has completed $D3b$ or $D7b$ in step σ after its last execution of $E3b$. And for every $p_j \in \mathcal{P}$ where $j \neq i$, Lemma 6.2.4-(i) follows directly from the IH, since p_j is neither appended nor deleted from Q by the action of σ.

This completes the case analysis for part (i) of Lemma 6.2.4.

□

Part (ii). Proof by induction on $|H|$.

Base Case: $|H| = 0$. In such case, every process is in its initial state, has never executed line $E3b$, $D3b$ or $D7b$; and $Q^H = \langle\,\rangle$. Therefore Lemma 6.2.4-(ii) holds trivially.

Induction Hypothesis (IH): For any $k > 0$, assume that Lemma 6.2.4-(ii) holds for all histories in $\mathcal{H}$, such that $|H| < k$.

Induction Step: Prove that part Lemma 6.2.4-(ii) holds for all H such that $|H| = k$. Let σ be the last step in H and let G satisfy $H = G \circ \sigma$. Proceed by a case analysis on σ.

Case 1: σ is not an execution of line $E3b$ by p_i. Then σ does not append anything to Q. In particular, either σ has no effect on Q or σ removes p_i from Q at line $D3b$ or $D7b$. Suppose for contradiction that Lemma 6.2.4-(ii) is false at the end of H. Then Q^H contains at least two instances of some process p_j. Since σ does not add anything to Q, all instances of p_j are also present in Q^G. This contradicts the IH, which implies that Lemma 6.2.4-(ii) holds for G, which has length $k-1$.

Case 2: σ is an execution of line $E3b$. Then σ appends p_i to Q. It follows from IH that Lemma 6.2.4-(ii) holds for H unless $p_i \in Q^G$. Suppose for contradiction that $p_i \in Q^G$. Then Q^G contains exactly one instance of p_i by the IH. In such case, by Lemma 6.2.4-(i), p_i completed line $E3b$ in G and did not subsequently complete $D3$ or $D7$ before executing σ. Since σ is another execution of $E3b$, this contradicts Lemma 6.2.3.

This completes the case analysis for part (ii) of Lemma 6.2.4.

□

Observation 6.2.6. For a given state of the history variable at the end of some finite history H i.e., Q^H, the following functions are defined:

$$QProcs(Q^H) := \{p_i \mid p_i \in Q\}$$

$$Qempty(Q^H) := \begin{cases} \text{true} & if\ Q = \langle\ \rangle \\ \text{false} & otherwise \end{cases}$$

$$Qhead(Q^H) := \begin{cases} Q[1] & if\ |Q| > 0 \\ null & otherwise \end{cases}$$

$$Qtail(Q^H) := \begin{cases} Q[|Q|] & if\ |Q| > 0 \\ null & otherwise \end{cases}$$

$$Qpred(Q^H, p_i) := \begin{cases} p_j & if\ \langle p_j, p_i \rangle\ is\ a\ subsequence\ of\ Q^H \\ null & otherwise \end{cases}$$

$$Qsucc(Q^H, p_i) := \begin{cases} p_j & if\ \langle p_i, p_j \rangle\ is\ a\ subsequence\ of\ Q^H \\ null & otherwise \end{cases}$$

Following Lemma 6.2.4-(ii), every process $p_i \in QProcs(Q^H)$ has a unique predecessor and successor (unless $null$) in Q^H, i.e., the values of $Qpred(Q^H, p_i)$ and $Qsucc(Q^H, p_i)$ are uniquely determined.

6.3 Recoverable Mutual Exclusion

Invariant 6.3. *Let $\boldsymbol{H}$ be any finite history in $\boldsymbol{\mathcal{H}}$ and $\boldsymbol{p_i}$ be some process in $\boldsymbol{\mathcal{P}}$. Define $\boldsymbol{lastPred(H, p_i)}$ as the last process appended to $\boldsymbol{Q}$ before $\boldsymbol{p_i}$'s last execution of line $\boldsymbol{E3}$ in $\boldsymbol{H}$, or $\boldsymbol{null}$ if no such process exists; and $\boldsymbol{lastSucc(H, p_i)}$ as the process appended to $\boldsymbol{Q}$ immediately after $\boldsymbol{p_i}$'s last execution of line $\boldsymbol{E3}$ in $\boldsymbol{H}$, or $\boldsymbol{null}$ if no such process exists. Then the following statements hold, collectively denoted Invariant 6.3-$\boldsymbol{(H, p_i)}$:*

a) *if* $p_i \notin QProcs(Q^H)$ then
   $q_i.ahead^H =$ null
   $q_i.chk^H = 0 \; or \; 1 \; or \; 3$
b) if $p_i \in QProcs(Q^H) \wedge p_i \neq Qhead(Q^H)$ then
   $q_i.ahead^H \neq$ null
   $q_i.chk^H = 1$
c) if $p_i \in QProcs(Q^H) \wedge p_i = Qhead(Q^H)$ then
   $q_i.ahead^H = null$
   $q_i.chk^H = 1 \; or \; 2 \; or \; 3$
d) $L^H = null$ if and only if $Q^H = \langle \, \rangle$
e) $RMEQ^H$ contains exactly $|Q^H|$ elements, and the elements of $RMEQ^H$ are the qnodes of the processes in Q^H, in that order.

Theorem 6.3.1. *For any finite $H \in \mathcal{H}$, Invariant 6.3 holds for H.*

Proof. The theorem is proved by induction on $|H|$.

Basis: $|H| = 0$. In such case, every $p_i \in \mathcal{P}$ is in its initial state, $RMEQ^H$ is empty, i.e., $L^H = null$ and $Q^H = \langle \, \rangle$. Hence, parts (a) and (d) of *Invariant 6.3*-(H, p_i) hold, because for every $p_i \in \mathcal{P}$, $q_i.ahead^H = null$ by initialization, and $Q^H = \langle \, \rangle$. Since $Qempty(Q^H) = true$, parts (b) and (c) hold trivially because their antecedents are false. Part (e) follows since $|Q^H| = 0$ and $RMEQ^H$ is empty.

Induction Hypothesis: For any $k > 0$, assume Theorem 6.3.1 holds for all histories $H \in \mathcal{H}$, such that $|H| < k$.

Induction Step: Prove that Theorem 6.3.1 holds for all H such that $|H| = k$. Let σ be the last step in H and let G satisfy $H = G \circ \sigma$. Let p_i be any process executing σ. By the IH, *Invariant 6.3*-(G, p_i) holds for all $p_i \in \mathcal{P}$. Let s be the last crash-recovery step taken by p_i in the current passage, if such a step exists, or $\bot$ otherwise. Define *critical operation* as any step by p_i that modifies the state of chk and $ahead$ fields on a qnode, or the state of L, Q and $RMEQ$. If σ is not a critical operation, then *Invariant 6.3*-(H, p_i) holds by the IH since the state of L, Q, and $RMEQ$ are unmodified by the action of σ. Therefore, it suffices to show that the invariant holds when σ is a critical operation. The proof proceeds by the following case analysis on σ.

Case 1: σ is an execution of line $E2$ by p_i. Observe that $p_i@E3$ holds at the end of H, by the action of σ. Therefore, if $p_i \in QProcs(Q^G)$ holds, then by extending the history H, it follows that p_i would eventually complete $E3$. Consequently, there would be two instances of p_i in Q and this contradicts Lemma 6.2.4-(ii). Hence, $p_i \notin QProcs(Q^G)$ holds. Then since p_i is not appended to Q^G in step σ, $p_i \notin QProcs(Q^H)$ holds. Also, $q_i.chk^H = 1$ by the action of σ. Therefore, part (a) of *Invariant 6.3*-(H, p_i) holds. Parts (b) and (c) follow trivially since $p_i \notin QProcs(Q^H)$. Moreover, since the state of L^G, Q^G and $RMEQ^G$ remain unmodified by the action of σ, parts (d) and (e) follow by the IH. Additionally, for every $p_j \in \mathcal{P} \backslash \{p_i\}$, *Invariant 6.3*-$(G, p_j)$ immediately implies *Invariant 6.3*-(H, p_j).

Case 2: σ is an execution of line $E3$ by p_i. Note that for every $p_j \in \mathcal{P} \backslash \{p_i\}$, *Invariant 6.3*-$(G, p_j)$ implies *Invariant 6.3*-(H, p_j). It remains to show *Invariant 6.3*-(H, p_i). Now, $p_i@E3$ at the end of G implies p_i completed $E2$ in its last step in G. Let F be a prefix of G up to and including the step in which p_i completed $E2$ for the last time in G. Then as explained in Case 1, $q_i.chk^F = 1$ and $p_i \notin QProcs(Q^F)$ hold. Also note that p_i is appended to Q only via p_i's completion of line $E3$, which did not occur in any step in G after F. Hence, $q_i.chk^G = 1$ and $p_i \notin QProcs(Q^G)$ hold by the IH. From the algorithm, the SAS instruction at line $E3a$ sets the lock pointer L to qnode q_i by the action of σ, and p_i is atomically appended to Q^G within the same step at $E3b$. Therefore part (d) of *Invariant 6.3*-(H, p_i) holds since $Q^H \neq \langle\,\rangle$ and $L^H = q_i$ by the action of σ. It remains to show parts (a), (b), (c) and (e). Consider the following subcases for the state of the $q_i.ahead$ at the end of H by the action of σ.

Subcase 2.1: $q_i.ahead^H = null$. Intuitively, $L^G = null$ holds since the SAS instruction at $E3a$ stored $null$ in the $q_i.ahead$ field in σ. Then it follows from *Invariant 6.3*-(G, p_i)-(d) that $Q^G = \langle\,\rangle$. Therefore, since $Q^H = Q^G \circ \langle p_i \rangle$ by the action of σ, $Qpred(Q^H, p_i) = null$ holds, i.e., $p_i =$

$Qhead(Q^H)$ and implicitly $p_i \in QProcs(Q^H)$. Hence, part (c) of *Invariant 6.3*-(H, p_i) holds as $q_i.chk^G = 1$ is unmodified in σ. Parts (a) and (b) follow trivially since $p_i \in QProcs(Q^H)$ and $p_i = Qhead(Q^H)$ respectively. And since $Qpred(Q^H, p_i) = null$, $|Q^H| = 1$ holds as p_i is the only process appended to $Q^G = \langle\,\rangle$ in σ. As $q_i \in RMEQ^H$ and $L^H = q_i$ by the action of σ, the $|Q^H|$ elements in $RMEQ^H$ and Q^H are q_i and p_i respectively, which implies part (e) of *Invariant 6.3*-(H, p_i).

***Subcase 2.2*:** $q_i.ahead^H \neq null$. Intuitively, $L^G \neq null$ holds since the SAS instruction at $E3a$ stored some non-null value in the $q_i.ahead$ field in σ. Then by *Invariant 6.3*-(G, p_i)-(d), $Q^G \neq \langle\,\rangle$ holds, which implies that $Qpred(Q^H, p_i) \neq null$ since $p_i \in Qprocs(Q^H)$. Recall that $p_i \notin QProcs(Q^G)$ holds, which implies $Qpred(Q^H, p_i)$ is not p_i itself. Therefore $p_i \neq Qhead(Q^H)$ holds. Hence part (b) of *Invariant 6.3*-(H, p_i) holds since $q_i.chk^H = 1$ holds. Parts (a) and (c) follow trivially since $p_i \in QProcs(Q^H)$ and $p_i \neq Qhead(Q^H)$ respectively. *Invariant 6.3*-(G, p_i)-(e) implies that the $|Q^G|$ elements of $RMEQ^G$ are the qnodes of the processes in Q^G, in that order. And since $p_i \notin QProcs(Q^G)$, *Invariant 6.3*-(G, p_i)-(e) implies that q_i is not in the elements of $RMEQ^G$. Therefore, since $Q^H = Q^G \circ \langle p_i \rangle$ and $L^H = q_i$ by the action of σ, the $|Q^H|$ elements in $RMEQ^H$ and Q^H are q_i appended to elements in $RMEQ^G$, and p_i appended to elements in Q^G respectively, which implies part (e) of *Invariant 6.3*-(H, p_i) as no other elements in Q^G and $RMEQ^G$ are modified by σ.

Case 3: σ is an execution of line $E7$ by p_i. As in the previous case, for every $p_j \in \mathcal{P} \backslash \{p_i\}$, *Invariant 6.3*-$\left(G, p_j\right)$ immediately implies *Invariant 6.3*-$\left(H, p_j\right)$ and it remains to show *Invariant 6.3*-(H, p_i). Now, $p_i@E7$ at the end of G implies p_i completed line $E4$ or $E6$ by reading $q_i.ahead = null$ in its last step in G. As per the lines of code in the algorithm, notice that only p_i can delete itself from Q by its completion of either $D3$ or $D7$, which did not occur in any step in G after its last completion of line $E3$, i.e., $p_i \in QProcs(Q^G)$ holds. Then $q_i \in RMEQ^G$ holds by *Invariant 6.3*-(G, p_i)-(e). And since only p_i can overwrite $q_i.ahead$ to a non-null value by completing $E3$, which did not occur in any step after p_i's last step in G, $q_i.ahead^G = null$ holds. Hence, q_i is the head of $RMEQ^G$, by the definition of a head qnode in $RMEQ$. Therefore, it follows from part (e) of *Invariant 6.3*-(G, p_i) that $p_i \in QProcs(Q^G)$ holds and also $p_i = Qhead(Q^G)$. Thus, part (c) of *Invariant 6.3*-(G, p_i) applies. Since σ does not modify the $q_i.ahead$ field, $q_i.ahead^H = null$ holds by the IH. Also $p_i = Qhead(Q^H)$ holds since p_i is not deleted from Q^G in step σ. Hence, part (c) of *Invariant 6.3*-(H, p_i) holds as $q_i.chk^H = 2$ by the action of σ. Parts (a) and (b) follow trivially since $p_i \in QProcs(Q^H)$ and $p_i = Qhead(Q^H)$ respectively. Parts (d) and (e) hold by the IH, since

$|Q^H| = |Q^G|$ and since the elements in $RMEQ^H$ and Q^H are exactly the same as in $RMEQ^G$ and Q^G respectively.

Case 4: σ is an execution of $RR3$ by p_i. Note that $p_i@RR3$ at the end of G implies p_i completed $RR2$ in its last step in G, reading $L = null$. Let F be a prefix of G up to and including the step in which p_i completed $RR2$ for the last time in G. Then $L^F = null$ implies $q_i \notin RMEQ^F$. Therefore $q_i \notin RMEQ^G$ also holds at the end of G since only p_i can append q_i to $RMEQ$ by completing line $E3$, which did not occur in any step in G after F. Then by part (e) of *Invariant 6.3*-(G, p_i), $p_i \notin QProcs(Q^G)$ holds. And since $p_i \notin QProcs(Q^G)$, *Invariant 6.3*-(G, p_i)-(a) implies $q_i.ahead^G = null$. Therefore $q_i.ahead^H = null$ and $p_i \notin QProcs(Q^H)$ also hold since σ does not modify the $q_i.ahead$ field nor does it append p_i to Q^G. Then part (a) of *Invariant 6.3*-(H, p_i) holds since $q_i.chk^H = 0$ by the action of σ. Parts (b) and (c) follow trivially since $p_i \notin QProcs(Q^H)$. Parts (d) and (e) follow by the IH, since $Q^H = Q^G$ and $RMEQ^H = RMEQ^G$.

Case 5: σ is an execution of $RR9$ by p_i. Note that $p_i@RR9$ at the end of G implies that either the conditions at $RR2$ and subsequently at $RR5$ and $RR7$ were not satisfied when p_i executed those lines for the last time in G after its last completion of $E3$ (i.e., q_i and p_i were already removed from $RMEQ$ and Q respectively before p_i invoked the $recoverRelease$ procedure), or that p_i has returned after the completion of the $release\_lock$ procedure invoked at either $RR6$ or $RR8$ (if the condition at $RR5$ does not hold) after its last completion of $E3$ in G. Let F be a prefix of G up to and including the step in which p_i completed $D3/D7$ for the last time. Then $p_i \notin QProcs(Q^F)$ holds. And as per the lines of code in the algorithm, since p_i does not append itself to Q by completing $E3$ in any step in G after F, $p_i \notin QProcs(Q^G)$ holds, which implies $q_i \notin RMEQ^G$ by part (e) of *Invariant 6.3*-(G, p_i). The analysis from this point is as in Case 4.

Case 6: σ is an execution of line $W5$ by p_i. Note that for every $p_j \in \mathcal{P}\backslash\{p_i\}$, *Invariant 6.3*-$(G, p_j)$ immediately implies *Invariant 6.3*-(H, p_j) and it remains to show *Invariant 6.3*-(H, p_i). Notice that p_i invokes $recoverBlocked$ after its last crash step s in G only if $q_i.chk = 1$ immediately after s, which implies p_i completed $E2$ but did not subsequently complete $E7$ or $W5$ before s. And since $q_i.chk$ is not modified in any line of code in the execution path of $CR3 - RB5/RB9/RB11 – W5$, $q_i.chk^G = 1$ holds. Furthermore, if $p_i@W5$ at the end of G via $RB5$, then since the condition at $RB4$ is satisfied only if $L = q_i$, $q_i \in RMEQ^G$ holds as q_i is not deleted from $RMEQ$ in any step in G after p_i's last execution of $RB4$. Similarly, if $p_i@W5$ at the end of G via $RB9$, then the fact that the condition at $RB8$ is satisfied implies that the $findMe$ method found q_i is reachable in the sequence of qnodes starting from L, i.e., $q_i \in RMEQ$.

And since q_i is not deleted from $RMEQ$ in any step in G after p_i's last completion of $RB8$, i.e., in the execution path of lines $CR3 - RB5/RB9/RB11 – W5$, $q_i \in RMEQ^G$ holds. On the other hand, if $p_i@W5$ via $RB11$, i.e., if $q_i.next \neq null$ holds when p_i executes $RB7$ for the last time in G, it follows that some process $p_j = lastSucc(G, p_i) \neq null$ has set $q_i.next$ to a non-null value in some step, say s', in G after p_i's last completion of $E3$. Let F be a prefix of G up to and including s'. Then $p_i \in QProcs(Q^F)$ holds. Next, $q_i.chk^G = 1$ implies that p_i did not complete either $E7$ or $W5$ in any step in G after F. Therefore, $p_i \in QProcs(Q^G)$ holds since p_i does not complete $D3/D7$ from the exit protocol unless it completes $E7$ or $W5$ first, which did not occur in any step in G after F. Hence, $q_i \in RMEQ^G$ holds by *Invariant 6.3*-(G, p_i)-(e).

Since p_i is enabled to execute $W5$ at the end of G, it follows that p_i completed either $W1$ or $W4$ in its last step in G by reading $q_i.ahead = null$. Observe that the $q_i.ahead$ field can subsequently be overwritten to a non-null value only via p_i's completion of $E3$, which did not occur in any step in G after p_i's last step in G. Hence, $q_i.ahead^G = null$ holds by the IH. Moreover, $q_i.ahead^G = null$ implies q_i is the head of $RMEQ$ and $p_i = Qhead(Q^G)$ holds by *Invariant 6.3*-(G, p_i)-(e). Since σ does not modify Q or $RMEQ$, it follows that q_i is the head of $RMEQ^H$ and $p_i = Qhead(Q^H)$. Hence, *Invariant 6.3*-(H, p_i)-(c) holds as $q_i.chk^H = 2$ by the action of σ. Parts (a) and (b) follow trivially since $p_i \in QProcs(Q^H)$ and $p_i = Qhead(Q^H)$ respectively. Parts (d) and (e) follow by the IH since Q^G and $RMEQ^G$ are unmodified in σ.

Case 7: σ is an execution of line $D1$ by p_i. Note that p_i is enabled to execute line $D1$ in σ if p_i invoked the $release\_lock$ procedure at either $FF3$, $RH2$, $RR6$ or $RR8$ at the end of G. From the structure of the algorithm, if $p_i@D1$ at the end of G via $FF2$ or $RH2$ then $q_i.chk^G = 2$ holds, or $q_i.chk^G = 3$ otherwise. Intuitively, p_i appended itself to Q for the last time in some step s' in G, since line $E3$ always precedes the line of code in which $q_i.chk$ is set to 2 or 3. Now, suppose for contradiction that $q_i \notin RMEQ^G$. Then by *Invariant 6.3*-(G, p_i)-(e), $p_i \notin QProcs(Q^G)$. However, since only p_i can delete itself from Q, $p_i \notin QProcs(Q^G)$ implies that p_i has completed $D3/D7$, say in step s'', after its last completion of $E3$ in G. Let F be a prefix of G up to and including s''. Then $q_i.chk^F = 3$ holds since p_i is in its exit protocol in step s''. And as per the lines of code in the algorithm, p_i can subsequently overwrite $q_i.chk$ to 2 by the end of G only if it executes either $E7$ or $W5$ in G after F. And since p_i's completion of $E3$ always precedes its execution of $E7$ or $W5$, $q_i.chk^G = 2$ contradicts step s'. Then as p_i does not reset the $q_i.chk$ value to 2 in any step in G after s'', $q_i.chk^G = 3$ holds. Now if $p_i@D1$ at the end of G via $FF3$ or $RH2$, then it contradicts p_i's completion of s'' in G after s', since $q_i.chk$ is always 2 when $p_i@FF3/RH2$. On the other hand, if

$p_i@D1$ at the end of G via $RR6$ or $RR8$, then the fact that the condition at $RR5$ or $RR7$ is satisfied contradicts the hypothesis that q_i is deleted from $RMEQ$ by the action of s'' and remains deleted till the end of G.

Therefore, $q_i \in RMEQ^G$ holds and $p_i \in QProcs(Q^G)$ also holds by *Invariant 6.3*-(G, p_i)-(e). And since $q_i.chk^G$ is 2 or 3, *Invariant 6.3*-(G, p_i)-(c) implies that $p_i = Qhead(Q^G)$ and $q_i.ahead^G = null$. Then since p_i is not removed from Q^G and since the $q_i.ahead$ field is unmodified in step σ, $p_i = Qhead(Q^H)$ and $q_i.ahead^H = null$ hold, which implies *Invariant 6.3*-(H, p_i)-(c) since $q_i.chk^H = 3$ by the action of σ. Parts (a) and (b) follow trivially since $p_i \in QProcs(Q^H)$ and $p_i = Qhead(Q^H)$ respectively. Since σ does not modify the state of Q and $RMEQ$, parts (d) and (e) hold by the IH. Additionally, for every $p_j \in \mathcal{P}\backslash\{p_i\}$, *Invariant 6.3*-$(G, p_j)$ immediately implies *Invariant 6.3*-(H, p_j).

Case 8: σ is an execution of line $D3$ by p_i. Since $p_i@D3$ at the end of G succeeds p_i's completion of $D1$, it follows from the analysis of Case 7 that $p_i \in QProcs(Q^G)$, $p_i = Qhead(Q^G)$, $q_i \in RMEQ^G$, $q_i.ahead^G = null$, and $q_i.chk^G = 3$ hold since p_i and q_i are not removed from Q and $RMEQ$ respectively, in any step in G after p_i's last completion of $D1$. Note that for every $p_j \in \mathcal{P}\backslash\{p_i\}$ *Invariant 6.3*-(G, p_j) immediately implies *Invariant 6.3*-(H, p_j). It remains to show *Invariant 6.3*-(H, p_i). Since σ does not modify the $q_i.ahead$ and $q_i.chk$ fields, $q_i.ahead^H = null$ and $q_i.chk^H = 3$ hold. Now, the state of Q and $RMEQ$ at the end of H depends on the result of the CAS operation at $D3a$ in σ. Consider the following subcases for the same.

> ***Subcase 8.1*****:** CAS returns $true$. Then the CAS is followed by the delete operation on Q within the same step at $D3b$. Therefore, part (a) of *Invariant 6.3*-(H, p_i) holds since $p_i \notin QProcs(Q^H)$ by the action of σ and since $q_i.ahead^H = null$. Parts (b) and (c) follow trivially as $p_i \notin QProcs(Q^H)$. Observe that the CAS at $D3a$ in σ succeeds only if $L^G = q_i$, i.e., if $lastSucc(G, p_i) = null$. Then since q_i is the tail of $RMEQ^G$, it follows from *Invariant 6.3*-(G, p_i)-(e) that p_i is also the tail element of Q^G. Also, by Lemma 6.2.4-(ii), Q^G contains at most one instance of p_i. Therefore, parts (d) and (e) of *Invariant 6.3*-(H, p_i) follow since $L^H = null$ and $Q^H = \langle\,\rangle$ hold, as σ removes the only element from both the sequences $RMEQ^G$ and Q^G, and thus $|Q^H| = 0$.
>
> ***Subcase 8.2*****:** CAS returns $false$. In such case, the subsequent delete operation at line $D3b$ is not executed, i.e, the state of Q is not modified, which implies $p_i \in QProcs(Q^H)$ holds by the IH since $p_i \in QProcs(Q^G)$. Since $p_i = Qhead(Q^G)$ and as p_i is not deleted by the action of σ, $p_i =$

$Qhead(Q^H)$ holds, which also implies $Q^H \neq \langle\,\rangle$. Therefore part (c) of *Invariant 6.3*-(H, p_i) holds since $q_i.chk^H = 3$ and $q_i.ahead^H = null$. Parts (a) and (b) follow trivially since $p_i \in QProcs(Q^H)$ and $p_i = Qhead(Q^H)$, respectively. And since $RMEQ^G$, L^G and Q^G are unmodified by the action of σ, parts (d) and (e) hold by the IH.

Case 9: σ is an execution of $D4$ by p_i. Since p_i reaches $D4$ in step σ only if it completed $D3$, and particularly if the CAS at $D3a$ succeeds, in its last step in G, it follows from the analysis of Case 8 (specifically from Subcase 8.1) that $p_i \notin QProcs(Q^G)$ and $q_i.ahead^G = null$. Also note that p_i is added to Q only via p_i's completion of $E3$, which did not occur in any step after p_i's last step in G. Therefore, $p_i \notin QProcs(Q^H)$ and $q_i.ahead^H = null$ hold. Hence, part (a) of *Invariant 6.3*-(H, p_i) holds since $q_i.chk^H = 0$ by the action of σ. Parts (b) and (c) hold trivially since $p_i \notin QProcs(Q^H)$. Parts (d) and (e) hold by the IH since the state of Q^G, L^G and $RMEQ^G$ remain unmodified in σ. Additionally, for every $p_j \in \mathcal{P}\backslash\{p_i\}$ *Invariant 6.3*-$\left(G, p_j\right)$ immediately implies *Invariant 6.3*-$\left(H, p_j\right)$.

Case 10: σ is an execution of line $D7$ by p_i. Note that $p_i@D7$ at the end of G implies that p_i invoked the $release\_lock$ procedure at either $FF3$, $RH2$, $RR6$ or $RR8$ after its last completion of $E3$ in G and as explained in Case 7, $q_i \in RMEQ^G$ holds. Moreover, $p_i@D7$ at the end of G holds only if p_i did not successfully complete $D3$ (if ever reached) in G after its last completion of $E3$, which also implies $q_i \in RMEQ^G$. Therefore by *Invariant 6.3*-(G, p_i)-(e), $p_i \in QProcs(Q^G)$ holds. Also, $q_i.chk^G = 3$ holds since $q_i.chk$ is not overwritten by any process in any step in G after p_i's last completion of $D1$, and hence, by part (c) of *Invariant 6.3*-(G, p_i), $p_i = Qhead(Q^G)$ and $q_i.ahead^G = null$ hold. Furthermore, $p_i@D7$ implies p_i completed either $D2$ or $D6$ in its last step in G, reading $q_i.next^G \neq null$. Since the $q_i.next$ field is set to a non-null value only by $lastSucc(G, p_i)$ when it completes $E5$ or $W3$ in its own passage, $p_j = lastSucc(G, p_i) \neq null$ holds. And since $Qsucc(Q^G, p_i) = p_j$ applies by the order of the append operations executed by p_i and p_j in G, $q_j.ahead = q_i$ holds by *Invariant 6.3*-(G, p_i)-(e), i.e., p_j sets $q_i.next = q_j$ when it completes $E5$ or $W3$ in its own passage, for the last time in G. As p_i overwrites $q_i.next$ to $null$ only when it completes $E1$, which did not occur in any step in G after p_i's last completion of $E3$, $q_i.next^G = q_i$ holds. Furthermore, $p_j \in QProcs(Q^G)$ implies $Q^G \neq \langle\,\rangle$, and hence $L^G \neq null$ by *Invariant 6.3*-(G, p_i)-(d).

Now, observe that p_i promotes p_j by the operation at $D7a$ in σ and deletes itself from Q^G at $D7b$. Hence, $p_i \notin QProcs(Q^H)$ holds by the action of σ. And since the $q_i.ahead$ and $q_i.chk$ fields are unmodified in

σ, $q_i.ahead^H = null$ and $q_i.chk^H = 3$ hold, which implies part (a) of *Invariant 6.3*-(H, p_i). Parts (b) and (c) follow trivially since $p_i \notin QProcs(Q^H)$. Note that $lastSucc(H, p_i) = Qhead(Q^H)$ holds and specifically, $Q^H \neq \langle\ \rangle$. Moreover, since $L^G \neq null$ is unmodified by σ, $L^H \neq null$ holds, which implies part (d) of *Invariant 6.3*-(H, p_i). Finally, since each process has at most one instance of its process ID in Q^H (by Lemma 6.2.4-(ii)), the sequences Q^H and $RMEQ^H$ are the same as Q^G and $RMEQ^G$ with their head elements p_i and q_i removed by the action of σ, respectively. Hence part (e) of *Invariant 6.3*-(H, p_i) holds.

Next, consider *Invariant 6.3*-$\left(H, p_j\right)$. Recall that by the action of σ, $p_j = Qhead(Q^H)$ and $q_j.ahead^H = null$ hold. Therefore part (b) of *Invariant 6.3*-$\left(G, p_j\right)$ applies as $p_j \neq Qhead(Q^G)$, and hence $q_j.chk^G = 1$ holds. Then by the result of the operation in σ, part (c) of *Invariant 6.3*-$\left(H, p_j\right)$ holds, since $q_j.chk^H = 1$ is the same as $q_j.chk^G$. Parts (a) and (b) follow trivially since their antecedents are false. And the analysis for parts (d) and (e) of *Invariant 6.3*-$\left(H, p_j\right)$ is similar to the analysis for parts (d) and (e) of *Invariant 6.3*-(H, p_i). Additionally, for every $p_k \in \mathcal{P} \backslash \{p_i, p_j\}$, *Invariant 6.3*-$(G, p_k)$ immediately implies *Invariant 6.3*-(H, p_k).

Case 11: σ is an execution of line $D8$ by p_i. Note that $p_i@D8$ at the end of G implies p_i completed $D7$ in its last step in G. Then following the analysis of Case 10, $p_i \notin QProcs(Q^G)$ holds since p_i is not appended to Q in any step in G after p_i's last completion of $D7$. And since p_i not appended to Q^G by the action of σ, $p_i \notin QProcs(Q^H)$ holds. Therefore, part (a) of *Invariant 6.3*-(H, p_i) holds since $q_i.chk^H = 0$ by the action of σ. Parts (b) and (c) follow trivially since their antecedents are false and parts (d) and (e) hold by the IH as the state of Q and $RMEQ$ remains unmodified in σ.

Case 12: σ is a crash-recovery step by p_i. By the definition of a crash-recovery step, only the program counter of p_i is reset to line $CR1$ at the end of H by the action of σ and hence the state of p_i's qnode, $RMEQ$ and Q remain unmodified in σ. Since no shared variable is modified by σ, *Invariant 6.3*-(H, p_i) holds by the IH.

□

Corollary 6.3.2. *The RGLock algorithm satisfies Mutual Exclusion.*

Proof. Say process $p_i \in \mathcal{P}$ is in the CS in passage m at the end of a finite history $H \in \mathcal{H}$. Then according to the algorithm, if p_i is enabled to execute the CS step at $FF2$ or $RH1$ at the end of H, it follows that p_i

completed either $E7$ or $W5$ in its last step in H. Therefore, $q_i.chk^H = 2$. Moreover, p_i is in the CS at the end of H if its qnode q_i is the head of $RMEQ^H$, i.e., if $q_i \in RMEQ^H$ and $q_i.ahead^H = null$. Then by part (c) of *Invariant 6.3*-(H, p_i), $p_i = Qhead(Q^H)$ holds. This implies the corollary, since no two processes can simultaneously be the head of the sequence Q^H.

□

Corollary 6.3.3. *The RGLock algorithm satisfies First-Come First-Served.*

Proof. Suppose for contradiction that there exists a finite history $H \in \mathcal{H}$ in which process $p_i \in \mathcal{P}$ completes line $E3$ in passage m and process $p_j \in \mathcal{P}\backslash\{p_i\}$ completes line $E3$ in passage n such that $lastSucc(H, p_i) = p_j$, and at the end of which, p_j is in the CS in passage n but p_i has not completed the CS step in passage m. In particular, p_i has not executed line $D7$ in passage m in H. In such case, based on the order of completion of line $E3$ by the processes, $Qpred(Q^H, p_j) = p_i$ holds since the sequence Q^H contains the elements $\langle p_i, p_j \rangle$ appended in that order. Whereas, by Theorem 6.3.1 and *Invariant 6.3*(H, p_i)-(c), if p_j is in the CS at the end of H, then $Qhead(Q^H) = p_j$, which contradicts with the order of elements in Q^H since p_i did not complete line $D7$ in any step in H.

□

Lemma 6.3.4. *In a crash-recoverable execution, the $findMe$ method terminates in a finite number of steps.*

Proof. In a crash-recoverable history $H \in \mathcal{H}$ in which some process p_i begins executing the $findMe$ method in some passage m, the scanning loop in $F3 - F9$ either terminates when the qnode of the invoking process is found as identified by the condition $temp = q_i$ at $F7$, or when the loop reaches the head qnode in $RMEQ$ as identified by the condition at $F5$, or when the loop performs a maximum of $N - 1$ iterations as identified by $run < N$ at $F3$. Hence, the lemma holds.

□

Theorem 6.3.5. *The RGLock algorithm satisfies Starvation Freedom.*

Proof. Suppose for contradiction that there is an infinite crash-recoverable history $H \in \mathcal{H}$ starting from an initial state, in which some process p_i begins the entry protocol in some passage m and never reaches the

CS step in passage m. This hypothesis is denoted by ‡, for convenience. Consider the following cases for the type of passage p_i is in at the end of H.

Case 1: m is a failure-free passage for p_i in H. Then according to the hypothesis, p_i loops forever at line $E6$ in passage m, reading $q_i.ahead \neq null$ repeatedly. Say p_i completed its doorway (line $E3$) for the last time in H in step s_i and line $E4$ in $s_i{}'$. Let E be a prefix of H up to but not including step s_i and F be a prefix of H up to and including step $s_i{}'$. Choose p_i so that $|E|$ is minimal. If $q_i.ahead$ in step $s_i{}'$ is $null$, then p_i branches to line $E7$ in H immediately after $s_i{}'$. As p_i does not execute line $E6$ in such case, passage m contradicts ‡. Now suppose that $q_i.ahead^F = q_j$, where q_j is the qnode of process $p_j = lastPred(F, p_i)$ as per *Invariant 6.3-(F, p_i)*-(e). Note that since p_i loops at $E6$ as per the hypothesis, p_i has already completed $E5$ in H after s_i and thus p_j does not loop forever at $D6$ in its own passage. Also note that p_j's last execution of $E1$ precedes step s_i and hence $q_j.next$ field is not overwritten by p_j unless p_j subsequently completes $D7$ first after its last completion of $E3$. Then, since E is minimal and H is fair, p_j eventually executes line $D7$ in its own passage, say in step s_j in H. Now let G be a prefix of H up to and including step s_j, such that $F \preccurlyeq G \preccurlyeq H$. Then $q_i.ahead^G = null$ by the action of s_j, which contradicts p_i looping forever at line $E6$ in passage m in H after the prefix G.

Case 2: m is a crash-recoverable passage for p_i in H. Then by the structure of the algorithm, p_i may loop forever at either line $E6$, or $W4$ in passage m, depending on which line in the entry protocol (lines $E1 - E7$) p_i crashes at. Note that by the result of Lemma 6.3.4, the $findMe$ method invoked within the $recoverBlocked$ method terminates in a finite number of steps and hence p_i does not loop forever at $F3$ if it ever invoked the $findMe$ method after its last completion of $E3$ in H. Moreover, since the $recoverHead$ and $recoverRelease$ methods are invoked in passage m immediately after its last crash-recovery step in H only if p_i crashes after completing line $E7$ or $W5$, and line $D1$ respectively, it follows that p_i is either enabled to execute the CS step (as $FF2$ is always executed immediately after $E7$, and $RH1$ is always executed after $W5$ in the algorithm), or that the CS step has already been taken in passage m (as either $FF2$ or $RH1$ is always executed immediately before $D1$ in the algorithm). Hence, cases where the $recoverHead$ and $recoverRelease$ methods are invoked immediately after p_i's last crash are not considered in this section as they contradict the original hypothesis. In particular, p_i must loop forever either at $E6$ in $acquire\_lock$ or at $W4$ in $waitForCS$ invoked via $recoverBlocked$. Say p_i completed its first step in passage m in s, and s' is the last crash-recovery step taken by p_i in H. The proof proceeds by a case analysis on p_i's execution before s'.

***Subcase 2.1*:** p_i did not complete line $E2$ in $H[s..s']$. Then since $q_i.chk$ is not set to 1 in any step in $H[s..s']$, the $recoverBlocked$ method is not invoked even if p_i crashed in any step before s'. Therefore, since $q_i.chk = 0$ at the beginning of passage m, the $q_i.chk$ value remains unmodified in $H[s..s']$ since p_i did not complete $E2$ in any step in H after s and moreover, p_i cannot execute any of the lines $E7$, $D1$, $D4$, $D8$, $W5$, $RR3$ or $RR9$ in passage m unless it completes $E2$ first. Hence the $failureFree$ procedure is invoked immediately after s'.The analysis from this point is as in Case 1.

***Subcase 2.2*:** p_i completed line $E2$, but did not complete $E3$ in $H[s..s']$. Note that since p_i did not complete line $E3$ before its last crash, its qnode is not appended to $RMEQ$. Hence, the $recoverBlocked$ method invoked immediately after s' returns $false$ either when the condition at line $RB2$ is satisfied or when the $findMe$ method invoked at $RB8$ returns false. In particular, since p_i did not complete $E3$ in H before s', and since q_i is not appended to $RMEQ$ after s' at any line in the $recoverBlocked$ procedure, the $findMe$ scan (if ever invoked in H after s') cannot locate q_i in $RMEQ$. Consequently, p_i then executes the $failureFree$ procedure invoked at $CR4$. Therefore, the analysis from this point is as in Case 1.

***Subcase 2.3*:** p_i completed line $E3$, but did not complete $E7$ in $H[s..s']$. Then the $q_i.chk$ value immediately after s' is 1 and the $recoverBlocked$ method invokes $waitForCS$, say in step s'', when the condition at either line $RB4$ or $RB8$ is satisfied or when the condition at $RB7$ is not satisfied and $RB11$ is reached. Let F'' be a prefix of H up to but not including s''. If the last step by p_i in F'' is an execution of $RB4$, the condition at $RB4$ holds only if q_i is the tail of $RMEQ^{F''}$. Consequently, when $waitForCS$ is invoked in s'', if $q_i.ahead^{F''} = null$, then p_i branches to $W5$. And since H is fair, p_i is eventually enabled to execute the CS step at $RH1$ in the $recoverHead$ procedure invoked at $RB12$, which contradicts ‡. Whereas, if the condition at $W1$ is satisfied, i.e., if $q_i.ahead^{F''} \neq null$, it follows that $lastPred(F'',p_i) \neq null$ (by *Invariant-6.3*(F'',p_i)-(e)) and particularly, $lastPred(F'',p_i)$ has not completed $D7$ in its own passage in F'' since s' because if $lastPred(F'',p_i)$ completed $D7$, then subsequently p_i would not loop forever at $E6$ or $W4$ in H, which contradicts ‡. And since H is fair, p_i loops at line $W4$ only until $lastPred(F'',p_i)$ executes line $D7$ in its own passage (observe that p_i completes $E5$ or $W3$ as per the hypothesis and hence $lastPred(F'',p_i)$ does not loop forever at $D6$ in its own passage), in which the $q_i.ahead$ field is set to $null$, contradicting the hypothesis that p_i loops forever at line $W4$ in passage m in H after the prefix F''. Similarly, if p_i invoked $waitForCS$ at $RB9$ or $RB11$, then p_i reaches line $W4$ only if the condition at $W1$ is satisfied, which implies that q_i has a non-null predecessor in $RMEQ$, and hence p_i loops at $W4$ only until that predecessor completes line $D7$ in its own passage, which also contradicts the hypothesis that p_i loops forever at $W4$ in passage m after F'' in H.

□

Corollary 6.3.6. *The RGLock algorithm satisfies Livelock Freedom.*

Proof. The result follows directly from Theorem 6.3.5.

□

Theorem 6.3.7. *The RGLock algorithm satisfies Terminating Exit.*

Proof. Suppose for contradiction that there is an infinite crash-recoverable history $H \in \mathcal{H}$ in which some process p_i begins executing its exit protocol and never completes the exit protocol in some passage m. Specifically, since the exit protocol contains only atomic operations other than the waiting loop at $D6$, it follows from the hypothesis that p_i loops forever at line $D6$ because p_i crashes only a finite number of times in the crash-recoverable history H. Say p_i executed line $D1$ for the last time in H in step s and let s' be the first subsequent execution of line $D6$ by p_i in passage m. Let G be a prefix of H up to and including the step in which p_i completed $D2$ for the last time in H (if $q_i.next \neq null$ at $D2$) or up to and including the step in which p_i executed $D3$ for the last time in H (if $q_i.next = null$ at $D2$). And if p_i executed $D2$ or $D3$ in its last step in G, then by the result of Theorem 6.3.1 and *Invariant-6.3*(G, p_i)-(c) that $p_i = Qhead(Q^G)$. Therefore $q_i \in RMEQ^G$ holds by *Invariant-6.3*(G, p_i)-(e).

Intuitively, p_i executed step s' as a result of the condition at $D2$ not holding, or as the CAS instruction at $D3$ (if ever executed in $H[s..s']$) returning $false$, which implies that $L^G \neq q_i$, i.e., $lastSucc(G, p_i) \neq null$. Let $p_j = lastSucc(G, p_i)$. Then the SAS instruction at $E3a$ sets $q_j.ahead = q_i$ when p_j completed $E3$ after p_i's last execution of $E3$ in G. Since H is fair, p_j eventually executes line $E5$ in some step s'' in H, in which it sets $q_j.ahead.next = q_j$, which contradicts the hypothesis that p_i repeatedly reads $q_i.next = null$ forever at $D6$ after step s'' in H. Moreover, note that p_i resets $q_i.next$ back to $null$ at $E1$ after p_j's completion of $E5$, only when it begins executing the entry protocol in a new passage, which contradicts the original hypothesis.

□

Theorem 6.3.8. *The RGLock algorithm satisfies Finite Recovery.*

Proof. Suppose for contradiction that in some crash-recoverable history $H \in \mathcal{H}$, some process p_i crashes for the last time in H in step s while executing a passage m and never reaches NCS at line $CR8$. Then following the result of Lemma 6.3.4, Theorem 6.3.5, and Theorem 6.3.7, p_i completes any of the crash-recovery procedures invoked immediately after s and eventually reaches line $CR8$ in a finite number of steps without looping forever at any of the lines $E6$, $W4$, $F3$, or $D6$, which contradicts the original hypothesis.

□

Theorem 6.3.9. *The RGLock algorithm incurs $\mathcal{O}(1)$ RMRs per process per failure-free passage.*

Proof. Note that since any write operations on the $ahead$, chk, and $yielded$ fields on a qnode incur at most one RMR each, and the number of such operations outside of any loops in a failure-free passage is $\mathcal{O}(1)$. Therefore, it suffices to show that a process incurs at most $\mathcal{O}(1)$ RMRs in a failure-free passage in the busy-waiting loops at lines $E6$ and $D6$ in the entry and exit protocols respectively. A process $p_i \in \mathcal{P}$ executing the busy-wait loop (if ever) at line $E6$ incurs one RMR for establishing a local cached copy of $q_i.ahead$ and an unbounded number of local reads of the cached value until some process $p_j \in \mathcal{P} \backslash \{p_i\}$ overwrites the $q_i.ahead$ field in the main memory by completing $D7$ in its own passage and invalidates p_i's cached copy. Therefore a subsequent cache miss for p_i implies that the value in the main memory has been modified. Note that p_j can write only to its own $q_j.ahead$ field when it executes line $E3$ and does not affect $q_i.ahead$ field. Hence, p_i terminates the busy-waiting loop after at most two RMRs in total. Similarly, in the exit protocol, a process waiting (if ever) at line $D6$ for its successor, say p_k, to update the $q_i.next$ field by completing $E5$ in its own passage, incurs at most two RMRs in total. Note that p_k only writes to the $next$ field on its own qnode in $E1$.

□

Chapter 7

Conclusion and Future Work

In this thesis, we introduced RGLock, an innovative locking mechanism for guaranteeing mutual exclusion, fairness and liveness in presence of crash failures in a crash-recovery model of emerging shared memory systems that incorporate a dedicated non-volatile main memory. Earlier attempts at making spin-locks recoverable suffer from a variety of shortcomings, such as a lock-holder losing ownership of the lock when crashed within the Critical Section, violation of starvation freedom in lock acquisitions and/or the inability to maintain the first-come-first-serve ordering. To the extent of our knowledge, our work is the first of its kind to formalize 'Recoverable Mutual Exclusion' as a novel correctness property for lock data structures whose state can be recovered from the non-volatile main memory following the failure of one or more of the contending processes in the system.

Despite their simplicity and having only a small number of lines of code, several existing mutex lock designs for multiprocessor computing contain very subtle aspects that make them difficult to prove correctness [52]. Our work presented a comprehensive proof of correctness for the proposed locking mechanism based on the assumed execution model. We believe that the ability to recover a spin lock is particularly valuable in transaction processing systems, which are intended to run indefinitely. Especially in long-running high performance computing applications, it may add a distinct advantage to exploit the benefits of recoverable in-memory computing. This chapter summarizes our contributions and identifies some potential avenues for further research.

7.1 Summary

In Chapter 1, we began by motivating the need for finding alternatives to the existing mutual exclusion algorithms when implementing highly concurrent data structures in a crash-recovery model. We give an account of spin-lock designs in existing literature and recount the benefits the scalable queue-based local-spin algorithm of MCS lock provides. Furthermore, we reiterate the limitations of the original MCS lock algorithm in a crash-recovery model when implemented “out of the box” in a system augmented with a non-volatile main memory, and finally set forth our objectives for a new crash-recoverable mutex.

In Chapter 2, we discussed the related work in the field of mutual exclusion algorithms as well as the advancements in development of media, concurrent data structures, and application interfaces which can exploit the persistence offered by non-volatile random access memory. We also recapitulate the shortcomings in prior efforts to design a crash-recoverable mutex, showing that they are impractical for our objectives due to the unrealistic constraints they place on the operating system and due to an inability to avoid a violation of at least one of the fairness or liveness properties in presence of crash failures. We highlight the significance of the performance potential of non-volatile main memory systems as a catalyst for inventing a crash-recoverable mutex that overcomes the limitations in prior work.

In Chapter 3, we present our execution model for the emerging multiprocessor systems that incorporate non-volatile main memory on a cache-coherent platform. We state our assumptions for the layout of the shared memory and the atomic primitives that are used by the processes in the system for accessing the shared objects, followed by a discussion of the terminology and definitions used in the failure-free and crash-recoverable execution histories. The uniqueness in our model is in the reliance on a persistent storage medium that preserves the shared state of each process irrespective of the number of crash failures occurring in multiple processes in the system.

In Chapter 4, we formalize the correctness properties of a recoverable mutual exclusion algorithm and highlight the importance of any crash-recoverable algorithm conforming to each of these properties. Specifically, we do not lay any constraints with respect to crash failures being restricted to occur only outside the critical section of the program. This distinguishes our correctness property from almost every prior attempt at creating a recoverable mutex lock in the existing work, as outlined in Chapter 2.

In Chapter 5, we present the main contribution of this thesis: the first crash-recoverable mutual exclusion algorithm for modern multiprocessor architectures with non-volatile main memory. We propose a special atomic instruction called *swap_and_store* (SAS) as an alternative to the existing *fetch_and_store* operation used in the doorway instruction, to facilitate the contending processes in joining a linked-list of per-process lock access structure as well as atomically registering either their ownership of the lock or their position in the sequence of processes lined up for acquiring the lock within a single step. Finally, in Chapter 6, we prove the correctness of our algorithm and justify the objectives laid out in the thesis for a crash-recoverable mutual exclusion lock.

7.2 Future Research

RGLock creates a scope to think beyond the traditional algorithms to maintain mutual exclusion and liveness properties in the context of crash-recovery. Nevertheless, there is limitation in our model that may have to be addressed to create more readiness among the practitioners to implement the algorithm in their architectures. It is not yet clear that the atomic SAS instruction will allow efficient implementation of a richer set of data structures, keeping in mind the advancements in software and hardware transactional memories. Specifically, the transactional semantics of the SAS instruction places an unusual requirement on contemporary multiprocessor architectures, and practitioners might favor implementations with transactional memories combined with some fault-tolerance mechanism such as write-ahead-logging, checkpointing, etc., over having to modify existing atomic primitives supported by the hardware in current generation computers. Further investigation of programming techniques and analysis of suitable schemes is required to find an appropriate implementation for the proposed atomic instruction and presenting these abstractions to programmers without much dependence on transactional memories, for that dissolves the very purpose of lock-based concurrency control.

Finally, in our initial quest to find one of the best known mutual exclusion algorithms that is also suitable for a crash-recovery model, we have considered Lamport’s Bakery algorithm [11] as a potential candidate solution for the desirable liveness and fairness properties it offers. Particularly, the Bakery algorithm is one of the first mutual exclusion algorithms that considered failures in any individual components. The notion of fault-tolerance in this algorithm is that any process that fails, halts in its NCS, and may eventually restart with its shared variables reset to their default values. At the outset, this lock might seem trivially recoverable since the state of any process with respect to the lock following a crash failure may be determined immediately by inspecting the state information of its spin variable (‘number’, as labeled in [11]), provided that the shared variables are stored in the non-volatile memory. However, restricting a process from leaving the procedure it executes each time it crashes requires a carefully designed recovery protocol that is beyond trivial. Moreover, the Bakery algorithm also uses unbounded registers for ‘number’. Nonetheless, for its sheer elegance, the Bakery algorithm is indeed an interesting solution to pursue in a crash-recoverable context, given that it satisfies all of the desired properties such as starvation freedom, doorway-FIFO, and wait-free exit, besides guaranteeing mutual exclusion.

References

[1] F. M. David and R. H. Campbell, “Building a self-healing operating system,” *Proceedings - DASC 2007: Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pp. 3–10, 2007.

[2] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, “Microreboot-A Technique for Cheap Recovery.,” *OSDI*, pp. 31–44, 2004.

[3] K. Bailey and L. Ceze, “Operating system implications of fast, cheap, non-volatile memory,” *Proceedings of the 13th USENIX conference on Hot topics in operating systems. USENIX Association*, pp. 2–2, 2011.

[4] E. W. Dijkstra, *Co-Operating Sequential Processes*. 1968.

[5] E. W. Dijkstra, “Solution of a problem in concurrent programming control,” *Communications of the ACM*, vol. 8, no. 9, p. 569, 1965.

[6] L. Lamport, “The mutual exclusion problem: part I---a theory of interprocess communication,” *Journal of the ACM*, vol. 33, no. 2, pp. 313–326, 1986.

[7] M. Michael and M. Scott, “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,” *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. ACM*, pp. 267–275, 1996.

[8] P. Magnusson, A. Landin, and E. Hagersten, “Queue locks on cache coherent multiprocessors,” *Proceedings of 8th International Parallel Processing Symposium*, pp. 165–171, 1994.

[9] T. Johnson and K. Harathi, “A prioritized multiprocessor spin lock,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 9, pp. 926–933, 1997.

[10] J. H. Anderson, Y. J. Kim, and T. Herman, “Shared-memory mutual exclusion: Major research trends since 1986,” *Distributed Computing*, vol. 16, no. 2–3, pp. 75–110, Sep. 2003.

[11] L. Lamport, “A new solution of Dijkstra’s concurrent programming problem,” *Communications of the ACM*, vol. 17, no. 8, pp. 453–455, Aug. 1974.

[12] J. Goodman, “Using cache memory to reduce processor-memory traffic,” *ACM SIGARCH Computer Architecture News*, 1983.

[13] L. Rudolph and Z. Segall, *Dynamic decentralized cache schemes for MIMD parallel processors*. 1984.

[14] T. E. Anderson, “Performance of spin lock alternatives for shared-memory multiprocessors,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, 1990.

[15] J. Mellor-Crummey and M. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.

[16] C. P. Kruskal, L. Rudolph, and M. Snir, “Efficient synchronization of multiprocessors with shared memory,” *ACM Transactions on Programming Languages and Systems*, vol. 10, no. 4, pp. 579–601, 1988.

[17] G. Graunke and S. Thakkar, “Synchronization algorithms for shared-memory multiprocessors,” *Computer*, vol. 23, no. 6, pp. 60–69, 1990.

[18] T. E. Anderson, E. D. Lazowska, and H. M. Levy, “Performance implications of thread management alternatives for shared-memory multiprocessors,” *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1631–1644, 1989.

[19] T. S. Craig, “Queuing spin lock algorithms to support timing predictability,” *1993 Proceedings Real-Time Systems Symposium*, 1993.

[20] M. L. Scott and W. N. Scherer, “Scalable queue-based spin locks with timeout,” *ACM SIGPLAN Notices*, vol. 36, no. 7, pp. 44–52, 2001.

[21] T. S. Craig, “Building FIFO and Priority-Queueing Spin Locks from Atomic Swap,” Technical Report 93-02-02, University of Washington, Seattle, 1993.

[22] H. Lee, “Local-spin mutual exclusion algorithms on the DSM model using fetch&store objects,” 2004.

[23] B. H. Lim and A. Agarwal, “Reactive synchronization algorithms for multiprocessors,” *ACM SIGOPS Operating Systems Review*, vol. 28, no. 5, pp. 25–35, 1994.

[24] S. Boyd-wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, “Non-scalable locks are dangerous,” *Proceedings of the Linux Symposium*, 2012.

[25] R. Myers, “Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask,” *Legal Information Management*, vol. 11, no. 01. pp. 19–23, 2011.

[26] Y.-I. Y. Chang, M. Singhal, and M. T. M. T. Liu, “A Fault Tolerant Algorithm for Distributed Mutual Exclusion,” *Proceedings Ninth Symposium on Reliable Distributed Systems*, pp. 146–154, 1990.

[27] D. Agrawal and A. El Abbadi, “An efficient and fault-tolerant solution for distributed mutual exclusion,” *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 1–20, 1991.

[28] P. Jayanti, T. Chandra, and S. Toueg, “Fault-tolerant wait-free shared objects,” *Journal of the ACM (JACM)*, 1998.

[29] M. K. Aguilera, W. Chen, and S. Toueg, "Failure Detection and Consensus in the Crash Recovery Model," *Distributed Computing*, vol. 13, no. 2, pp. 99–125, 2000.

[30] E. Elnozahy and W. Zwaenepoel, "Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit," *IEEE Transactions on Computers*, 1992.

[31] M. Prvulovic, Z. Z. Z. Zhang, and J. Torrellas, "ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," *Proceedings 29th Annual International Symposium on Computer Architecture*, pp. 111–122, 2002.

[32] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. a. Wood, "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," *Proceedings 29th Annual International Symposium on Computer Architecture*, pp. 123–134, 2002.

[33] G. Janakiraman and Y. Tamir, "Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers," *Proceedings of IEEE 13th Symposium on Reliable Distributed Systems*, pp. 42–51, 1994.

[34] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems*, vol. 17, no. 1. pp. 94–162, 1992.

[35] E. N. (Mootaz) Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[36] D. Patterson and J. Hennessy, *Computer organization and design: the hardware/software interface*. 2013.

[37] M. Michael and Y. Kim, "Fault tolerant mutual exclusion locks for shared memory systems," US Patent 7,493,618, 2009.

[38] U. Abraham, S. Dolev, T. Herman, and I. Koll, "Self-stabilizing l-exclusion," *Theoretical Computer Science*, vol. 266, no. 1–2, pp. 653–692, Sep. 2001.

[39] P. J. Denning, *Mutual exclusion*. 1991.

[40] P. Bohannon, A. Si, S. Sudarshan, J. Gava, T. B. Laboratories, M. Avenue, and M. Hill, "Recoverable User-Level Mutual Exclusion," *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on. IEEE,* pp. 293–301, 1995.

[41] P. Bohannon, D. Lieuwen, and a. Silbershatz, "Recovering scalable spin locks," *Proceedings of SPDP '96: 8th IEEE Symposium on Parallel and Distributed Processing*, 1996.

[42] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[43] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, “The missing memristor found.,” *Nature,* vol. 453, no. 7191, pp. 80–83, 2008.

[44] S. Tehrani, J. M. Slaughter, M. Deherrera, B. N. Engel, N. D. Rizzo, J. Salter, M. Durlam, R. W. Dave, J. Janesky, B. Butcher, K. Smith, and G. Grynkewich, “Magnetoresistive random access memory using magnetic tunnel junctions,” *Proceedings of the IEEE,* vol. 91, no. 5, 2003.

[45] G. R. Fox, F. Chu, and T. Davenport, “Current and future ferroelectric nonvolatile memory technology,” *Journal of Vacuum Science & Technology B: Microelectronics and Nanometer Structures,* vol. 19, no. 5, p. 1967, Sep. 2001.

[46] R. Weiss, “Flash memory takes over,” *Electron Des,* pp. 56–64, 2001.

[47] M. Raynal, “Algorithms for mutual exclusion,” 1986.

[48] J. H. Anderson, “Lamport on mutual exclusion,” *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing - PODC ’01,* pp. 3–12, 2001.

[49] L. Lamport, “A fast mutual exclusion algorithm,” *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 1–11, 1987.

[50] J. H. Anderson and Y.-J. Kim, “Nonatomic mutual exclusion with local spinning,” *Distributed Computing,* pp. 3–12, 2002.

[51] L. Lamport, “Proving the Correctness of Multiprocess Programs,” *IEEE Transactions on Software Engineering,* vol. SE-3, no. 2, 1977.

[52] P. A. Buhr, D. Dice, and W. H. Hesselink, “High-performance N -thread software solutions for mutual exclusion,” *Concurrency and Computation: Practice and Experience,* 2014.

[53] W. Pierce, *Failure-tolerant Computer Design*. 1965.

[54] B. Randell, *System structure for software fault tolerance,* vol. 10, no. 6. 1975.

[55] H. Plattner and A. Zeier, *In-Memory data management: Technology and applications*. 2012.

[56] P. A. Bernstein and N. Goodman, “An algorithm for concurrency control and recovery in replicated distributed databases,” *ACM Transactions on Database Systems*, vol. 9, no. 4. pp. 596–615, 1984.

[57] G. Barnes, “A method for implementing lock-free shared-data structures,” *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures, ACM,* pp. 261–270, Aug. 1993.

[58] M. Herlihy, “Wait-free synchronization,” *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 13, no. 1, pp. 124–149, 1991.

[59] M. Herlihy, V. Luchangco, and M. Moir, “Obstruction-free synchronization: double-ended queues as an example,” *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on. IEEE*, pp. 522–529, 2003.

[60] M. Herlihy, J. Eliot, and B. Moss, *Transactional Memory: Architectural Support For Lock-free Data Structures*, 2nd ed. 1993.

[61] N. Shavit and D. Touitou, “Software transactional memory,” *Distributed Computing*, 1997.

[62] I. Finocchi, F. Grandoni, and G. Italiano, “Designing reliable algorithms in unreliable memories,” *Computer Science Review*, 2007.

[63] D. B. Lomet, “Process structuring, synchronization, and recovery using atomic actions,” *ACM SIGSOFT Software Engineering Notes*, vol. 2, no. 2. pp. 128–137, 1977.

[64] K. L. Wu, W. K. Fuchs, and J. H. Patel, “Error recovery in shared memory multiprocessors using private caches,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 231–240, 1990.

[65] N. Neves, M. Castro, and P. Guedes, “A checkpoint protocol for an entry consistent shared memory system,” *PODC ’94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pp. 121–129, 1994.

[66] R. E. Ahmed, R. C. Frazier, and P. N. Marinos, “Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems,” *[1990] Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*, pp. 82–88, 1990.

[67] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz, “Application-level checkpointing for shared memory programs,” *ACM SIGOPS Operating Systems Review*, vol. 38, no. 5, p. 235, 2004.

[68] L. D. Molesky and K. Ramamritham, “Recovery protocols for shared memory database systems,” *ACM SIGMOD Record*, vol. 24, no. 2, pp. 11–22, 1995.

[69] R. Chopra, “Fault Tolerant Distributed Lock Manager,” US Patent 20,130,174,165, 2013.

[70] U. Helmich, A. Kohler, K. Oakes, M. Taubert, and J. Trotter, “Method and system to execute recovery in non-homogenous multi processor environments.,” US Patent 7,765,429. 27, 2010.

[71] A. Kumar and D. Stein, “Robust and recoverable interprocess locks,” US Patent 6,301,676, 2001.

[72] S. Lee, B. Moon, C. Park, J. Hwang, and K. Kim, “Accelerating In-Page Logging with Non-Volatile Memory.,” *IEEE Data Eng. Bull.*, vol. 33, no. 4, pp. 41–47, 2010.

[73] M. Wu and W. Zwaenepoel, “eNVy: a NonVolatile main memory storage system,” *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*, pp. 25–28, 1993.

[74] R. Bez and A. Pirovano, “Non-volatile memory technologies: Emerging concepts and new materials,” *Materials Science in Semiconductor Processing*, vol. 7, no. 4–6 SPEC. ISS., pp. 349–355, Jan. 2004.

[75] R. H. Katz, D. T. Powers, D. H. Jaffe, J. S. Glider, and T. E. Idleman, “Non-Volatile Memory Storage of Write Operation Identifier in Data Storage Device,” 1993.

[76] I. Moraru, D. Andersen, M. Kaminsky, and N. Binkert, “Persistent, protected and cached: Building blocks for main memory data stores,” *Work*. 2011.

[77] B. Lee, P. Zhou, J. Yang, Y. Zhang, and B. Zhao, “Phase-change technology and the future of main memory,” *IEEE micro*, 2010.

[78] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, “A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology,” *ACM SIGARCH computer architecture news. ACM.*, vol. 37, no. 3, pp. 14–23, 2009.

[79] J. Zhao, “Rethinking the Memory Hierarchy Design With Nonvolatile Memory Technologies,” The Pennsylvania State University, 2014.

[80] S. Lai, “Current status of the phase change memory and its future,” *Electron Devices Meeting, 2003. IEDM’03 Technical Digest. IEEE International. IEEE*, pp. 10–1, 2003.

[81] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne : Lightweight Persistent Memory,” *Asplos*, pp. 1–13, 2011.

[82] M. Baker, S. Asami, E. Deprit, J. Ouseterhout, and M. Seltzer, “Non-volatile memory for fast, reliable file systems,” *ACM SIGPLAN Notices*, vol. 27, no. 9, pp. 10–22, 1992.

[83] J. Ouyang, C.-W. Chu, C. R. Szmanda, L. Ma, and Y. Yang, “Programmable polymer thin film and non-volatile memory device.,” *Nature materials*, vol. 3, no. 12, pp. 918–922, 2004.

[84] J. D. Coburn, “Providing Fast and Safe Access to Next-Generation, Non-Volatile Memories,” UNIVERSITY OF CALIFORNIA, SAN DIEGO, 2012.

[85] S. Venkataraman and N. Tolia, “Redesigning Data Structures for Non-Volatile Byte-Addressable Memory,” in *Proceedings of USENIX Conference on File and Storage Technologies (FAST’10)*, 2010.

[86] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, “Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory.,” in *Proceedings of the 9th USENIX Conference on File and Storage Technologies - FAST’*, 2011.

[87] D. Li, J. S. Vetter, G. Marin, C. Mccurdy, C. Cira, Z. Liu, and W. Yu, “Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications,” *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IPDPS 2012*, pp. 945–956, 2012.

[88] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson, “Moneta: A high-performance storage array architecture for next-generation, non-volatile memories,” *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 385–395, 2010.

[89] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson, “Onyx: a protoype phase change memory storage array,” *HotStorage’11 Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, p. 2, 2011.

[90] A. N. Jacobvitz, R. Calderbank, and D. J. Sorin, “Coset coding to extend the lifetime of memory,” *Proceedings - International Symposium on High-Performance Computer Architecture*, pp. 222–233, 2013.

[91] W. Enck, K. Butler, T. Richardson, P. McDaniel, and A. Smith, “Defending against attacks on main memory persistence,” in *Proceedings - Annual Computer Security Applications Conference, ACSAC*, 2008.

[92] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, “Better I/O Through Byte-Addressable, Persistent Memory,” *Proc. ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 133–146, 2009.

[93] X. Wu and a. L. N. Reddy, “SCMFS: A file system for Storage Class Memory,” *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–11, 2011.

[94] J. Coburn, A. Caulfield, and A. Akel, “NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories,” in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011.

[95] J. C. Mogul, “Operating System Support for NVM + DRAM Hybrid Main Memory,” *HotOS*, pp. 1–5, 2009.

[96] S. Pelley, P. M. Chen, and T. F. Wenisch, “Memory persistency,” *Proceedings - International Symposium on Computer Architecture*, pp. 265–276, 2014.

[97] I. Jeremy, C. Edmund, B. N. Doug, and B. Thomas, “Dynamically Replicated Memory : Building Reliable Systems from Nanoscale Resistive Memories,” *Memory*, vol. 38, no. 1, pp. 3–14, 2010.

[98] S. Park, T. Kelly, and K. Shen, “Failure-Atomic msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data,” *Proceedings of The European Professional Society on Computer Systems (EuroSys)*, pp. 225–238, 2013.

[99] R. Danek and W. Golab, “Closing the complexity gap between FCFS mutual exclusion and mutual exclusion,” *Distributed Computing*, vol. 23, no. 2, pp. 87–111, Mar. 2010.

[100] N. A. Lynch and M. R. Tuttle, “An Introduction to Input / Output Automata,” *CWI Quaterly*, vol. 2, no. September 1989, pp. 1–30, 1988.

[101] T. Harris, “A pragmatic implementation of non-blocking linked-lists,” *Distributed Computing. Springer Berlin Heidelberg*, pp. 300–314, 2001.

[102] W. Golab, “Deconstructing Queue-Based Mutual Exclusion,” Oct. 2012.